AN EVENT-DRIVEN, INCLUSIONARY AND SECURE APPROACH TO KERNEL INTEGRITY

Satyajit Grover, Divya Naidu Kolar Sunder, Samuel O. Moffatt and Michael E. Kounavis
Corporate Technology Group, Intel Corporation

Keywords: Rootkits, Kernel, Security, Virtualization, Hypervisor.

Abstract: In this paper we address the problem of protecting computer systems against stealth malware. The problem is important because the number of known types of stealth malware increases exponentially. Existing approaches have some advantages for ensuring system integrity but sophisticated techniques utilized by stealthy malware can thwart them. We propose Runtime Kernel Rootkit Detection (RKRD), a hardware-based, event-driven, secure and inclusionary approach to kernel integrity that addresses some of the limitations of the state of the art. Our solution is based on the principles of using virtualization hardware for isolation, verifying signatures coming from trusted code as opposed to malware for scalability and performing system checks driven by events. Our RKRD implementation is guided by our goals of strong isolation, no modifications to target guest OS kernels, easy deployment, minimal infrastructure impact, and minimal performance overhead. We developed a system prototype and conducted a number of experiments which show that the performance impact of our solution is negligible.

1 INTRODUCTION

In this paper we address the problem of protecting computer systems against stealth malware. By the term 'stealth malware' we mean malicious code that uses rootkit mechanisms to install and actively hide itself on OS kernels. The problem is important because the number of known types of stealth malware has been increasing exponentially. There were less than 500 types of stealth malware known before the end of 2005, whereas this number increased to 5500 by the end of 2006. The problem is also important because applications running on computer systems rely on services provided by OS kernels. Applications assume that the kernel services are uncompromised, an assumption which is not always true. Among the many types of attacks performed by stealth malware, prevalent attacks are based on modifying code and static data, import tables, system call tables, interrupt tables and exception handlers.

Several system integrity mechanisms have been proposed in the past (e.g., [3, 6, 7, 9]). To some extent, prior art is not completely adequate for addressing several types of stealth malware threats. In what follows we justify our claim. Some detection mechanisms examine kernel data structures to find malicious activity (e.g., data structures accessed by the process explorer, netstat etc.). Such solutions do not always protect against static code and data modifications. They also do not address transient attacks. Other solutions perform heuristic-based analysis for identifying known malware signatures in memory. These approaches have some advantages over the former for ensuring system integrity. However, sophisticated techniques utilized by stealthy malware can thwart them. The main disadvantage of these approaches is that they are reactive as opposed to proactive solutions to malware detection. Other approaches check the correctness of signatures coming from trusted code as opposed to malware and are thus more scalable but their security depends on the OS they are trying to protect.

We propose Runtime Kernel Rootkit Detection (RKRD) (pronounced 'record'), a hardware-based, event-driven, secure and inclusionary approach to kernel integrity that addresses some of the limitations of earlier approaches. RKRD is designed to ensure the integrity of kernel code as well as critical kernel data structures and the flow of execution of kernel modules. By 'event-driven' we mean that RKRD detects the introduction/activity of rootkits and other malicious kernel code at runtime,

Grover S., Naidu Kolar Sunder D., O. Moffatt S. and E. Kounavis M. (2008).
AN EVENT-DRIVEN, INCLUSIONARY AND SECURE APPROACH TO KERNEL INTEGRITY.
In *Proceedings of the International Conference on Security and Cryptography*, pages 411-420
DOI: 10.5220/0001916004110420

specifically at the moment when measured code and data structures are altered. By 'secure' we mean that RKRD defends itself against attacks using virtualization technology. Virtualization technology provides isolation between virtual machines and is used to protect the RKRD components that measure the integrity of guest OS kernels. By 'inclusionary' we mean that RKRD checks that the signatures coming from known and trusted code have the values they should have. It does not scan code looking for signatures of malicious code, as typical anti-virus tools do. An inclusionary approach is therefore proactive and scalable.

We implemented a RKRD prototype based on the afore-mentioned design principles. Our system development was guided by the goals of strong isolation, no modifications to the target guest OS kernels, easy deployment, minimal infrastructure impact, and minimal performance overhead. We conducted experiments with real implementations of stealthy malware and were able to detect attempts to compromise system integrity. The experiments showed that the performance impact of our solution is negligible.

The paper is structured as follows: In Section 2 we discuss principles that underpin the RKRD design. Section 3 discusses the RKRD threat model along with design assumptions. The system architecture is described in Section 4, followed by details of our prototype in Section 5. We discuss the performance impact of the prototype in Section 6. Section 7 discusses how RKRD addresses the threat model. Following this, in Section 8 we discuss related work in the area. Finally, we present concluding remarks in section 9.

2 DESIGN PRINCIPLES

In what follows we elaborate on the three principles that guide the design of RKRD:

1. **Isolation using Virtualization Hardware.** Integrity detection mechanisms are vulnerable if a circular dependency exists between these mechanisms and the OS. Stealthy malware utilizes that dependency to hide itself from the integrity checking mechanisms. RKRD leverages virtualization hardware to create an isolated and trusted partition in an untrusted environment. This partition is used for running the RKRD service that measures guest OS kernel integrity. The isolation provided by virtualization breaks that dependency. This is essential since malware affecting the kernel cannot get access to the integrity checking solution, thereby mitigating its ability to hide itself.
2. **Verifying the Signature of Trusted Code.** The RKRD approach is based on verifying the signatures of known and trusted code in the OS. This inclusionary approach is unlike many prior approaches to system integrity that scan for malicious code signatures. The trusted code base on a system presents a limited (and thus scalable) footprint for the integrity checking solution. It can be effectively monitored to drastically narrow the opportunities for compromise due to malicious code.
3. **System Checks based on Events.** Our approach continuously monitors kernel integrity through event-driven checking. This mechanism can detect a change in the measured code and data structures at the moment when they are altered. This addresses transient attacks that compromise the system briefly and then revert it to a known good state. Such attacks can prevent existing integrity checkers from discovering their activity by compromise kernel.

3 ASSUMPTIONS AND THREAT MODEL

RKRD assumes immutability of the preboot environment. A preboot environment can be protected by technologies that perform basic checks that ensure the integrity of the boot code. RKRD does not assume that the kernel is immutable, just that its construction can be verified. We believe that this is a fair assumption since the integrity of any piece of code or data can be asserted with appropriate hash value checks regardless of where and when the checks take place. However, protecting the preboot environment represents substantial engineering effort which is out of the scope of this work. Further assumptions made in the system design are stated in Section 4.

We consider a threat model in which the adversary gains complete access to the kernel by utilizing a documented method, or an exploit based on a known bug or vulnerability. This gives the adversary the ability to execute code at kernel privilege, to modify existing code and data, and to insert new code into the control flow. Protecting a system from such an adversary poses various challenges on operating systems. Operating systems typically provide simplicity and performance through the use of a single address space for all

kernel code. In such case, the adversary can modify any crucial system tables, drivers, and also change CPU state by modifying general purpose registers. In addition, an attack can be waged in a transient manner, reverting back to the original state to leave no trace for a detection system. RKRD includes such transient attacks in its threat model

Following are the classes of attacks we considered when designing RKRD. An analysis on how these threats are addressed is presented in Section 7.

- **Import Table Hooking.** Applications utilize functions exported by libraries and other applications. These external calls are maintained in an import table. Rootkits can overwrite the entries in these tables to alter the control flow into malicious code.
- **Code & Static Data Modifications**. In this method, the actual code of a targeted function is overwritten with malicious code. This task is accomplished using documented APIs defined on modern operating systems. For instance, Microsoft has provided APIs such as OpenProcess, ReadProcessMemory and WriteProcessMemory for writing to and reading from another process's memory. These well-documented APIs serve as a toolkit for rootkit installers.
- **IDT/Exception Handler Hooking.** On x86 processors, an Interrupt Descriptor Table (IDT) is used to handle hardware and software interrupts. A malicious user can insert a rootkit by modifying an IDT entry to point to a malicious function instead of the default interrupt handler. This method of hooking will cause the hooked code to be called before the default interrupt handler function. Shadow Walker is a well known rootkit that utilizes this method.
- **System call Table Hooking.** All native system service addresses are listed in a data structure called the system call table. To call a specific system function, the system service dispatcher looks up the address of the function in this table. With complete access to the kernel, an adversary can change the entries in this table to point to malicious code instead.
- **Dynamic Kernel Object Manipulation.** This is a method of altering the dynamic state-keeping structures in the kernel. An example is the system process table that maintains a list of executing processes. By altering such structure, the adversary can hide the existence of malicious code from auditing software that monitors the system.

It is important to note that there are several classes of attacks that RKRD does not address. First, RKRD assumes that the user is not the adversary. If the user was the adversary, the user could perform more aggressive attacks which are beyond the scope of this work. Second, RKRD does not address buffer/integer overflow/underflow attacks that can support control flow changes, data modifications, and other malicious activities. There are auxiliary methods for handling these attacks like execute disable methods, use of canary values etc. Third, RKRD does not address attacks resulting from conversions to/from canonical formats, input verification errors and string formatting. In addition it does not defend against race conditions used for privilege escalation within the guest OS kernel or privilege escalation into the VMM. Another group of attacks not addressed by RKRD are DMA device-based attacks. Virtualization mechanisms can protect a kernel from DMA-based attacks by removing the kernel and its key data structures from access by devices. Another issue not addressed in the threat model is attacks against manifests. These attacks occur when the software is transported from the manufacturer but before it is delivered to the user. One big problem that needs to be addressed is the management of a potentially large number of manifest signing keys.

4 SYSTEM ARCHITECTURE

4.1 Virtualization-based Environment

RKRD is designed to monitor the integrity of the kernel to assure that there is no malware inserted into any of the OS execution paths. It provides continuous integrity verification without interfering with the legitimate operations of system patching and module installation The RKRD design does not require any modification of the guest OS. An overview of the system is provided in Figure 1. The system control flow is monitored using the hypervisor and the integrity verification is done by a software component executing in a trusted environment that is isolated from the OS as described in section 4.3. Integrity verification is performed whenever the control flow monitor detects a change in the system and in a measured component.

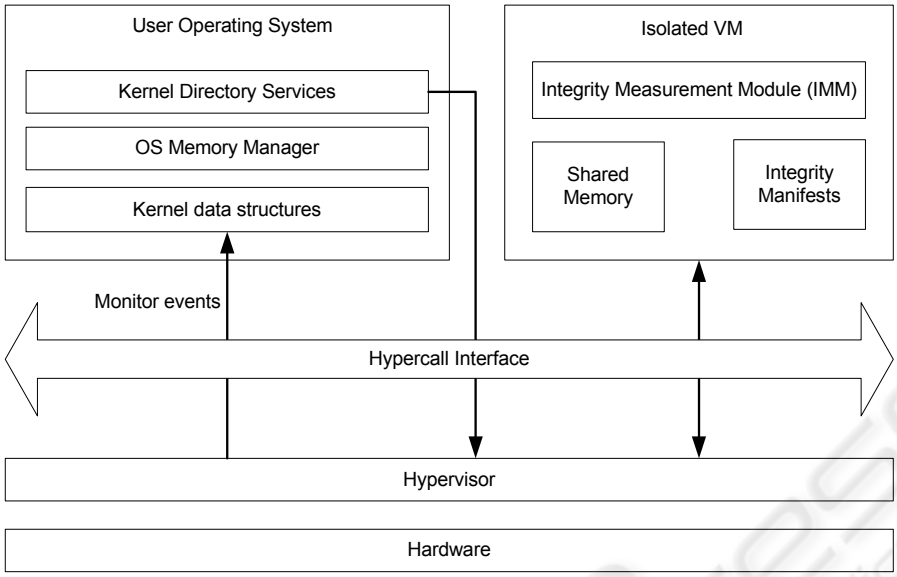

Figure 1: System Architecture.

In order to describe our architecture we present a brief overview of a virtualization-based system that leverages hardware virtualization. On our platform we utilized the Intel® Virtualization Technology or VT-x; similar mechanism can provide virtualization on other processors. Virtualization refers to the technique of partitioning the physical resources of a processor or a chipset into Virtual Machines (VMs) and inserting a higher privilege executive under the OS. This executive is known as a hypervisor and the privilege level is known as VMX-root mode in VT-x.

A control transfer into the hypervisor is called a VMExit and transfer of control to a VM is called a VMEntry. A VM can explicitly force a VMExit by using a VMCALL instruction. Such instruction is also called 'hypercall'. A guest OS runs in VMX-non-root mode which ensures that critical OS operations cause a VMExit. This allows the hypervisor to enforce isolation policies. The hypervisor manages the launching/shutdown of VMs, accesses to memory, and accesses to microprocessor registers. It also manages interrupts and instruction virtualization and can protect against DMA-based attacks.

In what follows we describe the three main components of the RKRD architecture: the Kernel Directory Service, the Integrity Measurement Module, and the Hypervisor. Details regarding their implementation in our prototype are described in Section 5.

4.2 Kernel Directory Service (KDS)

The KDS is designed to execute as a service on the guest OS every time a new module is loaded. It provides the hypervisor with the names and locations (virtual addresses) of modules loaded in the kernel through a hypercall. The hypervisor passes this information to the Integrity Measurement Module (IMM). A compromised KDS will not affect the security of RKRD. If a rootkit is installed through compromising the KDS, the presence of this rootkit can be detected by checks on other critical data structures (i.e., import and export tables, SSDT and IDT tables) as described in the next section. KDS exists to improve the performance of the system by identifying the location of the kernel components that are verified in memory by the IMM

4.3 Integrity Measurement Module (IMM)

The Integrity Measurement Module (IMM) is responsible for checking the integrity of kernel modules and data structures. It executes on a VM isolated by the hypervisor from the guest OS. This isolation satisfies the first design principle as stated in Section 2.

The IMM checks the integrity of kernel modules by comparing their pages in memory against their corresponding integrity manifest (Hardjono and Smith, 2006). These manifests are accessible to the IMM on its file system in the isolated partition. We

assume that the integrity of the manifest database can be guaranteed. However, the security of that approach is beyond the scope of this paper. RKRD assumes that only legitimate kernel code has an associated manifest and that the code is not self-modifying in a non-deterministic way. A manifest contains cryptographic hashes for each page of the code and static data sections of the module. It also contains the structures that are required to revert relocations done by the loader, and the module's import and export tables. Each manifest is created simply by extracting this information from the compiled binary file of a module, e.g. an ELF file on Linux® and a Portable Executable (PE) file for the Windows® XP Operating System. The information in a manifest allows the IMM to reconstruct a module's relevant data structures from a runtime snapshot of the module's memory. This is essential to RKRD because it provides us with a means of determining that a module's runtime memory matches its compile time state. An uncompromised system should exhibit no discrepancies between its runtime and compile time images.

The IMM checks system integrity by executing a three phase 'IMM integrity' algorithm:

- **Phase 1.** For each module, the IMM verifies each page of the code and static data sections by comparing its cryptographic hash against the original stored in the manifest. Also, it extracts the memory location of all the exported and imported symbols. This information is utilized in the following phases and also for future checks.
- **Phase 2.** For each module, the IMM finds all the imported entries and verifies that they are being exported by a module that passed Phase 1.
- **Phase 3.** The IMM extracts data structures that point to essential services, such as system calls and interrupt handlers, and verifies that these services lie within modules that have passed phases 1 and 2. This phase can be extended to other guest OS data structures.

This algorithm satisfies the second design principle as stated in Section 2. A detailed description of our implementation of this algorithm is provided in Section 5. By running this algorithm on all system modules we can verify that they are in accordance with the original intent of the module author as recorded in the manifest. The IMM then marks each module as passed or failed and notifies the hypervisor. All modules that pass are marked as measured and included in the trust boundary. On an uncompromised system, all modules should pass the integrity check since the IMM has their manifests and they should only be interacting with other measured code. If any module fails the test then that event is interpreted as the detection of malware. The hypervisor can then take preventive action, e.g. log the activity and halt the guest OS.

4.4 Hypervisor

The hypervisor serves as a trusted entity in our system, extending the 'root of trust' provided by the hardware platform. The hypervisor has three main functions in the RKRD architecture. First, it maintains system integrity state that includes a list of all modules that are executing as part of the kernel. This list is initiated by the KDS and is reinforced by constant monitoring of the guest OS. Second, it monitors events that may alter system state and triggers integrity verification if necessary. This mechanism satisfies the third design principle as stated in Section 2. The hypervisor monitors the guest OS for the following events and initiates integrity checks as needed:

- **Addition or Removal of Modules.** Each time a module is loaded into or removed from the kernel, the hypervisor performs an integrity check by executing the IMM integrity algorithm.
- **Change in a Measured Data Structure.** Checks are also required when a measured data structure is altered. The hypervisor continuously monitors such structures by marking their shadow pages as read-only and intercepting any changes made to them. Any attempt to alter their content is routed through the hypervisor's page fault handler, which initiates a check to verify it.
- **Paging-in of Measured Pages.** Malware can overwrite a page of a system module when it is written out to the page file on disk. When that page is loaded back in response to a page fault, the module can be infected by the corrupted code on it. In order to prevent such an attack, the hypervisor intercepts all page faults that read a page into memory from the disk. It then checks the integrity of that page as described in phase 1 of the IMM integrity algorithm.

The hypervisor also satisfies requests made by the IMM for copies of pages of guest OS memory that are to be verified.

5 PROTOTYPE

We implemented RKRD as a proof of concept and system prototype. In this section we describe our implementation with respect to the system architecture.

5.1 Software

We implemented a light-weight hypervisor for our prototype. Our hypervisor provides us with the capabilities to monitor system events as required and to create shadow page tables needed to intercept paging events and modifications to data structures. We extended it to provide interfaces for communication with KDS and the IMM and to monitor guest OS events for initiating integrity checks. The guest OS in our prototype is Windows® XP with Service Pack 2. We used this OS to demonstrate that our technology works on a proprietary environment for which we do not have access to the source code (and thus did not modify for the sake of providing kernel integrity). Our architecture works just as well with any other OS, provided we can generate the appropriate manifests. The KDS is implemented as an OS service and leverages the standard EnumDeviceDrivers API (Microsoft Corporation, 2008) for its task of collecting information about system modules.

In addition to the basic software platform described above, our design requires host OS specific manifests. We wrote a tool in Visual C++ for the generation of manifests of OS system modules. The tool requires only a compiled binary file as input. On the Windows® XP operating system the system modules are implemented in the PE file format (Microsoft Corporation, 2006). These modules include executables (including the kernel), DLLs and device drivers. These manifests contain cryptographic hashes based on the SHA-1 algorithm. The security provided by a hash algorithm typically depends on the assumptions made on the computing capability of the attacker. We are aware of recent cryptanalytic work (Wang et al., 2005) which substantially reduced the complexity of finding collisions in SHA-1. If SHA-1 is considered insecure under some attacker assumptions it can be replaced by any other hash function which demonstrates better collision and pre-image resistance. Manifests also contain the relocation fix-ups, Import Address Table (IAT), Export Table (ET), Import Lookup Table (ILT) and other PE file tables.

5.2 System Operation

At boot time, the hypervisor launches the guest OS and the IMM in separate virtual machines. The IMM registers with the hypervisor using a hypercall. This process of registration establishes a shared memory area between the two that is protected by the hypervisor. After launching the IMM VM, the KDS is started on the guest OS. It sends the list of executing system modules to the hypervisor, including the name and virtual address of each. The hypervisor initiates a system integrity check by sending this information to the IMM. The IMM then executes the IMM integrity algorithm for integrity verification as shown in Figure 2.

In phase 1 the IMM executes the following steps for each module:

- **accesses** the corresponding manifest and utilizes it to request the module's memory image from the hypervisor. The hypervisor copies the requested memory pages from the guest OS into the memory shared with the IMM.
- **reconstructs** the compile time image of the module using these pages along with the information from the manifest.
- **verifies** the integrity of each page of the code and static data sections of the module based on the SHA-1 cryptographic hash values in the manifest.
- **stores** the necessary PE file tables for the module.

In phase 2 the IMM executes the following steps for each module:

- **verifies** the links (function pointers) between the module and its dependencies using the ILT information from the manifest along with the ET and IAT entries collected in phase 1.
- **iterates** through all ILT entries and finds the corresponding ET entries.
- **verifies** if the IAT entries match the virtual addresses determined in the previous step for ILT entries

In phase 3 the IMM:

- **requests** the hypervisor to retrieve the System Services Descriptor Table (SSDT) and verifies that all entries in it point to measured code; and
- **requests** the hypervisor to retrieve the Interrupt Descriptor Table (IDT) entries from memory (first 32) and verifies that the they point to measured code.

For each module and each data structure that passes verification, the IMM notifies the hypervisor.

During system operation, the hypervisor monitors events as described in Section 4 and initiates integrity checks as needed. If at any time the IMM detects an unknown modification to the system, it marks it as an unexpected change and notifies the hypervisor. The hypervisor can then take corrective action based on system policy, including halting the host OS to prevent damage from malware. We note that a shadow page is marked as 'read-only' before or after checks.

```
IMM_INTEGRITY_ALGO(kernel_module_list l,
                   int list_size)
{
   int i;
   for(i = 0; i < list_size; i++)
   {
    kernel_module m = get_module(l, i);
      manifest f = open_manifest(m);
      image g = get_memory_image(m);
      if((section(f) == CODE) ||
         (section(f) == STATIC_DATA))
    {
       page_list p = get_pages(g);
       int j;
         for (j=0; j < psize(g); j++)
         {
            if(hash_from_manifest(f,j)
            != hash_from_memory(p, j))
            {
               error(BAD_MODULE, j);
            }
         }
      }
      store_tables_for_next_phases(f, g);
   }
   //end of phase 1

   for(i = 0; i < list_size; i++)
   {
    kernel_module m = get_module(l, i);
      imported_entries_list il =
    get_imported_entries(m);
    exported_entries_list el =
    get_exported_entries(il, m);
      int j;
      for (j=0; j < lsize(il); j++)
    {
           if(il[j] != el[j])
       {
            error(BAD_IT_ENTRY, m, j);
         }
      }
      add_module_into_pass_list(m);
   }
   //end of phase 2

   table ssdt = get_SSDT_table();
   table idt = get_IDT_table();
   for(i = 0; i < lsize(ssdt); i++)
   {
    if(!is_in_phase_2_pass_list(ssdt[i]))
    {
        error(BAD_SSDT_ENTRY, i);
    }
   }
   for(i = 0; i < lsize(idt); i++)
   {
    if(!is_in_phase_2_pass_list(idt[i]))
    {
        error(BAD_IDT_ENTRY, i);
    }
   }
}
   //end of phase 3
```

Figure 2: The IMM Integrity Algorithm.

6 PERFORMANCE

The primary goal of RKRD is to develop a system that can verifiably detect many popular exploits of system vulnerabilities. However, it is important that its operation does not have a significant impact on the performance of the measured system. This section provides some analysis on the performance impact of the prototype.

We measured the time taken to execute the three phases of the IMM integrity algorithm. We collected two sets of data. A first set was collected using the prototype to verify four Windows® XP kernel modules: ntoskrnl.exe, hal.dll, kdcom.dll, and bootvid.dll. These modules were selected because they are required for critical system functionality. Another set was collected from all kernel modules. All experiments were performed on an Intel® Centrino platform with a 1.83 GHz Core™ Duo processor and 2GB of DDR2 RAM.

Performance data for the verification of the four critical modules, broken up by each phase of our integrity check algorithm, can be found in Table 1. Phase 1 completes in 71 ms. There were approximately 490 pages worth of data verified and structures extracted, at 0.14 ms/page. Each page is 4KB in size. Phase 2 averaged 6 ms and phase 3 averaged 1 ms. Table 2 provides performance data on the time required to verify all kernel modules that can be identified using standard user-level Windows® APIs. The total number of modules varied between 100 and 102 on our test system. Phase 1 averaged 444 ms, phase 2 averaged 255 ms, and phase 3 averaged 1 ms for all modules.

Table 1: Time taken for 4 critical modules on a hypervisor.

Phase 1	Phase 2	Phase 3	Total
71 ms	6 ms	1 ms	78 ms

Table 2: Time taken for all modules on a hypervisor.

Phase 1	Phase 2	Phase 3	Total
444 ms	255 ms	1 ms	700 ms

We also created a version of the algorithm that ran in a non-virtualized environment. It used a kernel driver to service memory access requests of the application. This was done to calculate the performance overhead of using hypercalls for memory access. The performance data for the four critical modules and for all modules is provided in Tables 3 and 4 respectively. The overhead associated with virtualization is significant. The time

it took to complete the algorithm for the four modules increased by 28% between the virtualized and non-virtualized case, and by 29% for the case of all modules. The overhead of running the IMM on a virtualized system comes from the overhead of using hypercalls and the performance overhead of running in a virtualized environment. The algorithm itself performs well but future work is required to reduce the overhead of virtualization and hypercalls.

Table 3: Time taken for 4 critical modules without hypervisor.

Phase 1	Phase 2	Phase 3	Total
55 ms	5 ms	1 ms	61 ms

Table 4: Time taken for all modules without hypervisor.

Phase 1	Phase 2	Phase 3	Total
300 ms	241 ms	1 ms	542 ms

There are several simple event-driven checks that were highlighted previously. When pages holding module code or static data sections are paged in from disk they must be verified. We monitored the number of page faults in the system to see how often pages are paged in from disk. Figure 3 shows the rate of paging of code section pages on the test system under normal desktop use for the four critical modules when the OS is running on top of the hypervisor. The spikes in the figure happen when a new application set is loaded.

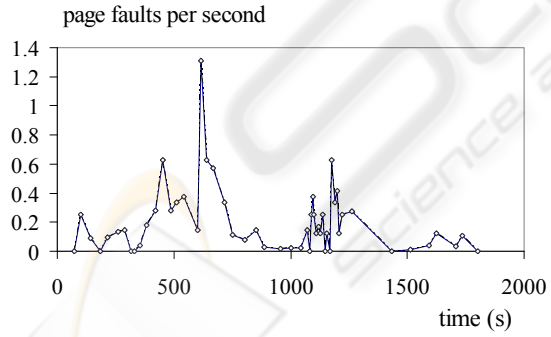

Figure 3: Rate of Paging from Disk.

Another event-driven check involves verifying the integrity of the SSDT and the IDT whenever they are modified. These data structures and the pages they reside on were not modified during normal system operation on our test platform and this check should only need to be performed rarely, such as when a piece of software is loaded into the kernel. With less than 2 page faults per second, the performance impact of the integrity check is less than 0.1%. A third event based check, which is also rare, is to verify the integrity of a page in memory that contains code or static data of a protected module if an alteration is made.

The system only performs a full verification of all the modules the first time or when a new module is loaded. Otherwise the system performs event based checks on occurrences as described above which are rare on our test system. These have minimum impact on the system performance.

7 THREAT ANALYSIS

In this section we revisit the threat model described in Section 3 and show how RKRD addresses the threats discussed there.

- **Import Table Hooking.** RKRD monitors the entire control flow of code executing in the kernel, from memory accesses to higher level OS control structures, providing malware no place to insert itself into the control flow. Specifically, Phase 2 of the algorithm addresses import and export table hooking. It ensures that the links (function pointers) between a system module and its dependencies are unmodified and point to the exact address location provided by the dependency in its export table.
- **Code & Static Data Modifications.** Phase 1 of the algorithm described above ensures that the code and static data sections remain unmodified during the runtime execution of a module. It is invoked whenever the hypervisor intercepts events that may cause the memory image of a module to change.
- **IDT/Exception Handler and System call Table Hooking.** Phase 3 of the algorithm addresses attacks to the system data structures. As the hypervisor continuously monitors the structures and intercepts any modification to them, RKRD can ensure that they are not altered by malicious code, without limiting alterations to them by measured code. This addresses the threat from system call table hooking and interrupt table hooking.
- **Dynamic Kernel Object Manipulation.** RKRD can monitor any structure specified by the programmer and hence it can monitor dynamic kernel objects in addition to the IDT and SSDT data structures specified in the IMM integrity algorithm.

- **Transient Attacks.** The RKRD architecture prevents transient attacks through the use of a hypervisor which monitors events that may alter previously measured code or structures. The monitoring can intercept a modification at the moment it occurs. On interception, control is transferred to the higher privilege of the hypervisor and returns to the guest OS if and only if the modification is found to be valid.

8 RELATED WORK

Copilot (Petroni et al., 2004) is a kernel integrity monitor that runs on a coprocessor system attached to the PCI bus of the host being monitored. This removes the dependence on the host kernel and can thus independently monitor the kernel. Copilot accesses the contents of the host's main memory over the PCI bus. However it has been shown that memory access over the PCI bus can be spoofed to limit or even alter the contents of the memory being examined (Rutkowska, 2007). Such an attempt thwarts the independent monitor. Our technology does not rely on the PCI bus. Instead we utilize hardware virtualization to run our monitor and manager from the highest privilege of a hypervisor. This provides us with direct access to the system memory. Furthermore, Copilot cannot detect transient attacks, those that change the kernel state and then change it back before Copilot rescans the system. In our design, the hypervisor can initiate a system scan the moment measured code or data structure changes from its last measured state.

PatchGuard (Microsoft Corporation, 2007) is a Microsoft technology that monitors and measures essential system data structures and modules in order to detect tampering. The measuring is done by performing a checksum. It is however still dependent on the OS. A kernel rootkit can exploit that dependency to its own advantage. Our technology provides protection to the system data structures and modules by measuring them using a cryptographic hash from a higher privilege entity, i.e. the hypervisor using hardware virtualization. This also enables us to similarly measure any OS without modifying its source code.

SecVisor (Seshadri et al., 2007) is a hypervisor that is designed to ensure code integrity for commodity kernels. It attempts to ensure that code executing at the kernel privilege level has been approved by the user. However, it makes no attempt to ensure that code executing at the kernel privilege does not tamper with system modules or structure. It is well understood that such a scenario is possible using standard driver loading procedures or exploiting bugs in the kernel. With our approach we can protect the trusted areas of the kernel from other areas to detect such attacks. Also, unlike SecVisor, our approach does not need any modification to the OS kernel source.

System Virginity Verifier 2.3 (Rutkowska, 2005) makes sure that a user program or any kernel module does not modify the code sections in the kernel code. It is done by verifying if the code sections of important system DLLs and system drivers (kernel modules) are the same in memory and in the corresponding PE file on disk. Our technology is capable of checking all system drivers loaded, not just the important kernel modules as in SVV. SVV uses a software based approach to compare the code section in the memory and the PE file, unlike our approach which is based on hardware virtualization.

9 CONCLUDING REMARKS

We presented a runtime rootkit detection system that uses hardware virtualization technology to measure and monitor the integrity of a kernel. It ensures system integrity against an adversary with complete access to the kernel. We believe that our work has some importance because it addresses a number of limitations of the state-of-the-art, as discussed in the related work section. Our approach is independent of the OS and does not require any modification to the measured kernel or system modules. We have demonstrated the feasibility of our approach via a prototype implementation that has minimum impact on the performance on the system.

There are several classes of attacks that RKRD does not address. RKRD does not address buffer/integer overflow/underflow attacks. RKRD also does not address attacks resulting from conversions to/from canonical formats, input verification errors and string formatting. In addition it does not defend against race conditions used for privilege escalation and DMA device-based attacks. Another issue not addressed is the management of signed manifest keys and in the threat model, the attacks against manifests. In future work, we propose to enhance the threat model to include some of these attacks as well as more sophisticated malware. We would also like to address the main source of performance impact i.e. the copying of memory pages from the guest OS to the IMM VM.

ACKNOWLEDGEMENTS

The authors would like to thank David M. Durham and Hormuzd Khosravi for their contributions to many of the concepts discussed in this paper as well as spearheading the RKRD project in the lab. They would also like to thank Gaya Nagabhushan for helping with building the RKRD prototype and Jesse Walker, Prashant Dewan and Travis Schluessler for their useful suggestions on how to improve the quality of the paper.

REFERENCES

X. Wang, Y. L. Yin and H. Yu, "Finding Collisions in the Full SHA-1", *Lecture Notes in Computer Science*, Vol. 3621 (November 2005), pp. 17-36

Microsoft Corporation. "Microsoft portable executable and common object file format specification". Available at: http://www.microsoft.com/whdc/system/platform/firm-ware/PECOFF.mspx, 2006.

Microsoft Corporation. "*Kernel enhancements for windows vista and windows server 2008*". Available at: http://www.microsoft.com/whdc/system/vista/kernel-en. mspx, 2007.

Microsoft Corporation. "Enumdevicedrivers function (windows)". http://msdn2.microsoft.com/en-us/library/ms682617(VS.85).aspx, 2008.

T. Hardjono and N. Smith. "TCG infrastructure working group architecture part ii – integrity management. Specification", *Trusted Computing Group*, 2006. https://www.trustedcomputinggroup.org/specs/IWG/IWG ArchitecturePartII v1.0.pdf.

N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor", *In USENIX Security Symposium*, pages 179–194. USENIX, 2004.

J. Rutkowska. "System virginity verifier, defining the roadmap for malware detection on windows system". Kuala Lumpur, Malaysia, September 2005.

J. Rutkowska. "*Beyond the CPU: Defeating hardware based RAM acquisition Tools*", *BlackHat DC* 2007, February 2007.

A. Seshadri, M. Luk, N. Qu, and A. Perrig. "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSs". In Thomas C. Bressoud and M. Frans Kaashoek, editors, *SOSP*, pages 335–350. ACM, 2007.