Detecting Rootkits With the RAI Runtime Application Inventory

Shabnam Aboughadareh, Christoph Csallner
Computer Science and Engineering Department
The University of Texas at Arlington, Arlington, TX 76019, USA
shabnam.aboughadareh@mavs.uta.edu, csallner@uta.edu

ABSTRACT

Remotely determining which precise code is running on which machines is hard. This is especially true if the monitored machines lack modern security features and may be under malware attack, since in such a scenario the malware may have already manipulated applications and operating systems. Existing approaches to this problem are heavy-weight and have a large attack surface, which is frequently attacked by both applications and malware.

To address this problem, this paper introduces RAI, a light-weight code monitoring tool that is especially well-suited for legacy systems. While potentially useful for many software maintenance tasks, this paper applies RAI for detecting ongoing rootkit attacks. Specifically, in our experiments on several user and kernel mode rootkits, our approach achieved with moderate overhead and a relatively low false positive rate a 100% rootkit detection rate.

CCS Concepts

•**Security and privacy** → **Intrusion/anomaly detection and malware mitigation;** *Distributed systems security;* •**Software and its engineering** → *System administration;* •**Applied computing** → Surveillance mechanisms;

Keywords

RAI; TDOIM; remote runtime code monitoring; rootkit detection; legacy system maintenance

1. INTRODUCTION

Remotely determining which precise code binaries are running on which machines is hard. This challenge has several aspects. First (1), the monitored binaries may change frequently in main memory at runtime. These changes often originate from dynamically generated or self-modifying code, e.g., for dynamic performance optimization or dynamic code obfuscation. A more recent reason are malware attacks. For example, an attacked program's in-memory binaries, as well as the binaries of any other program on the machine, may be changed frequently by an advanced persistent threat (APT).

Monitoring *legacy binaries* poses additional problems as (2) legacy applications are often tightly bound to legacy systems that lack modern security infrastructure. Moreover, (3) restarting legacy applications is often costly and should be avoided, to prevent business interruption.

This problem is significant in practice, since legacy applications are used widely in all industries, including banking, transportation, and health care. Legacy applications are, e.g., susceptible to malware attacks. Among others, a malware attack may lead to a data breach. In 2014 each data breach cost the affected company in the U.S. on average over five million dollars [46].

Reliable data on where which code is running would be valuable, on one hand, for software maintenance—for example, to prioritize reverse engineering, program understanding, and testing. On the other hand, it would also be valuable for detecting ongoing malware attacks. In this case the data may show that due to malware infection, on some machines an executing code part differs slightly from what is currently executing on other machines.

Malware may manipulate and subvert the underlying operating system and any monitoring tool. We define *legacy systems* as systems that lack the hardware-based root of trust that is necessary for ensuring that a remote machine has not been taken over by malware [44]. A common example of such a hardware-based root of trust is the Trusted Platform Module (TPM) [1]. Without such hardware extensions, there will never be absolute protection from attacks for legacy systems. But we propose an approach that gets closer to this impossible ideal than existing techniques.

Such malware take-overs are made easy by the monolithic architecture of many popular operating systems and the many bugs and security vulnerabilities they contain. To minimize the risk of subversion by malware, an important design goal is therefore to minimize the footprint of any monitoring logic deployed on any monitored machine.

Closely related existing approaches to determining which code is running on legacy machines are programmable debuggers and profilers. These approaches provide valuable runtime information but are not sufficient.

Most closely related, (programmable) debuggers such as Dalek and IDA Pro expose much low-level information about running processes, including memory contents and control over the execution [42]. However, being so powerful, debuggers are frequently attacked by both applications and

SSPREW '16, December 5-6, 2016, Los Angeles, CA, USA
© 2016 ACM. ISBN 978-1-4503-4841-6/16/09...$15.00
DOI: http://dx.doi.org/10.1145/3015135.3015138

malware. Malware attacks on debuggers are made easy by debuggers being large and complex multi-layer systems with a large attack surface [55, Chapter 16].

To prevent debugging, many applications employ sophisticated tricks to prevent debuggers from exposing runtime information [19]. Also, to observe both user and kernel mode programs and to detect ongoing malware attacks, a kernel-mode debugger would be needed, which may not be available on all running legacy machines. Finally, debuggers have a relatively high overhead, which can alter the control flow of the monitored application.

Modern profilers such as DTrace use low-overhead dynamic instrumentation to monitor given program locations, both in the kernel and in user-space [7]. However, such profilers have a big footprint (e.g., DTrace places among others an entire VM in the operating system kernel) and are therefore big malware targets. Low-level assembly instruction execution tracing has a very high overhead. Higher-level tracing (e.g., at function boundaries) is typically not available because it requires static instrumentation, which for many legacy applications is effectively impossible since they are shipped without debug symbols [59].

To determine which code is executing on which machine, we present RAI, a Runtime Application Inventory. At a high level, RAI is a client-server scheme that periodically monitors the physical memory contents. RAI does this with a minimal agent deployed on client machines, which may even be installed during an ongoing malware attack.

We evaluate RAI by applying it to detect malware in a scenario where the monitored application is running in many identical (or homogeneous) instances. Such a setup is often the case in a distributed application (e.g., in a data center) or in widely deployed consumer software.

Specifically, the Tiny Distributed On-demand Integrity Monitor (TDOIM) extends the RAI server to compare the current memory contents of the operating system and applications across machines. When some instances start to diverge from the rest, this is an indication that the diverging instances may have been manipulated by malware. In other words, the server periodically infers and updates a white-list directly from the monitored application instances and checks all clients against this dynamic white-list.

To evaluate TDOIM, we implemented TDOIM for Linux and conducted several small experiments. For the experiments we used different user-mode and kernel-mode rootkits that perform code injection, hooking, and in-line patching to infect applications. In our experiments TDOIM could always pinpoint the compromised systems and the infected memory regions in these systems. In our experiments the runtime overhead on the client machines was moderate. The number of false positives was relatively low for kernel modules (4% of the modules' pages) and zero for the OS kernel and user-space applications.

To summarize, this paper makes the following major contributions.

- This paper describes RAI, the first approach to monitoring which code is running on machines that is designed to work even on legacy systems under malware attacks.

- The paper describes TDOIM, a RAI application that is more effective at detecting rootkits in legacy applications than the state of the art.

- The paper provides an initial empirical evaluation of TDOIM on several Linux user and kernel mode rootkits, where with moderate overhead and a relatively low false positive rate our TDOIM prototype implementation achieved a 100% rootkit detection rate and scaled to a highly distributed setup of dozens of monitored application instances.

2. BACKGROUND

This section provides necessary background information on code management and layout techniques that are common across many platforms and operating systems. We also describe how malware, and especially rootkits, inject code and how existing anti-malware approaches often increase the malware attack surface of the machines the anti-malware approaches are supposed to protect.

2.1 Code Location at Runtime

While details differ among the various platforms and operating systems, there are two broad categories of memory address space, kernel and user. Most widely used operating systems are monolithic and thus both the kernel and its extensions (such as device drivers) have full access to all memory. Each user-mode application has its own memory address space and cannot directly access the code or data of other applications or the kernel.

Code and its data exist in two main forms, on disk in files and loaded in main memory. The compiler typically places both code and data in a number of segments or sections. While the terms have well-defined meanings that differ across platforms, in this paper we use *segment* and *section* interchangeably to refer to a chunk of either code or data either on disk or in main memory. For each segment the compiler can set access rights (read, write, and execute), which most operating systems enforce via the platform's memory management hardware support. For example, code is typically placed in executable non-writable segments and constant data values are typically placed in non-executable non-writable segments.

On most platforms, each application is allocated in a contiguous block of virtual memory, which the platform maps to a likely non-contiguous set of pages in physical (main) memory. The physical address of a given page therefore varies across program executions. Virtual addresses may also differ across executions, e.g., due to address space layout randomization (ASLR). A program can run with only some of its pages in physical memory, the remaining pages are swapped out to secondary storage, such as a disk. Kernel addresses are often an exception, as several OSs such as Linux make sure that their kernel code and read-only data segments are both in contiguous chunks of physical memory and never swapped out.

Most platforms have mechanisms for dynamic code loading, e.g., to support a device the user plugged in at runtime or to handle various input values. This code is loaded into an existing address space and adds additional code and data segments. Common examples in the kernel are kernel-level device drivers. In user-mode, many platforms have mechanisms to dynamically load code libraries.

2.2 Malware: Code-Injecting Rootkits

The term "rootkit" has traditionally been used for software that is used in an initial attack to elevate a user to root access on a victim system. However in the current literature rootkit refers to software that an attacker uses after gaining the desired level of access through some other attack such as a zero-day exploit. In this new definition, which we use in this paper, a rootkit uses the existing level of access to create a more persistent backdoor to retain access in the long term and possibly hides itself and malicious payloads from anti-malware tools.

Many rootkits work by injecting code into the victim system [31]. To inject code, rootkits use their high OS privilege level to overwrite code segments on disk or in main memory or load additional (malicious) code segments (e.g., as a dynamically linked library) and change read-only data segments (i.e., that contain function pointers) to link to them. A rootkit may inject code at any level, i.e., in the kernel, in kernel extensions such as device drivers, and in user-mode applications.

For example, a code-injecting rootkit may be planted by a buffer overflow attack, which overwrites parts of the call stack with a rootkit and transfers control there. In this scenario the new call stack injects or overwrites code in the heap and jumps there. This paper focuses on code-injecting rootkits, since they are common and hard to detect, especially in legacy applications.

2.3 Current Anti-malware Limitations

Many current anti-malware techniques have a large attack surface and thus are themselves vulnerable to many attack vectors [60, 37, 36], which in turn makes the machines they are running on more vulnerable. Due to their large number and complexity, we do not enumerate all such attack vectors. Instead we list a few of the attack vectors that are common in existing techniques but are absent from our work.

As an example functionality, current antivirus tools store application white-lists and malware black-lists on each monitored client. (1) First, such list stores can be manipulated on disk or in memory. (2) Second, the process of adding new elements to the lists can be intercepted or subverted, at the source server, during transmission, or at the client-side antivirus tool destination. (3) Finally, retrieving elements from the lists can be intercepted.

In addition to these direct attack vectors, there are also the following well-known indirect attacks. Existing anti-malware tools have a lot of features and therefore contain a lot of code, which presents many opportunities for code vulnerabilities and zero-day exploits. The functionality is also spread across many places such as disk, memory, and registry, with their own attack vectors. To access such system resources, current anti-malware tools use a large number of operating system APIs, such as system calls, which malware frequently manipulates [22].

3. RAI APPROACH AND DESIGN

This section gives an overview of RAI's main assumptions and the resulting architecture. For example, RAI monitors efficiently code that can be loaded dynamically and code that is partially swapped out to disk.

On each platform running the monitored application we install a tiny client-side agent. We keep the agent's functionality minimal to minimize the attack surface added to the monitored machines. This agent does not require special hardware or virtualization, which together with its minimal functionality allows deployment on a wide variety of platforms. This client-side agent can be installed on a platform during a malware attack and does not require recompilation or restart of the monitored application.

Each client-side agent periodically computes hash values of the physical main memory that is currently used for kernel, device drivers, and user-mode applications. Each agent then sends the hashes to the RAI server. The RAI server thereby keeps track of which machine is currently running which parts of the operating system and which parts of which applications.

3.1 Assumptions and Threat Model

A monitored application may run on a user's system in user-mode, kernel-mode, or both. A user system is the operating system (OS) and all applications running on the OS. The OS may run directly on hardware, in a virtual machine (VM) instance, or in a container [18].

We assume a user system may be under malware attack. As common in rootkit-type attacks, we assume the adversary has full access to the whole attacked machine, including the file system and all memory address spaces. The adversary may exploit this access and continuously hack a victim machine's operating system and the application running on the machine. As part of such an attack, the attacker injects code for both malicious payloads and to hide its trace.

The adversary may carry out the attack by manipulating binaries on the disk, by infecting loaded images in memory, or both. For code injection into the different kinds of software running on the victim's machine, a concrete attack may include a combination of the following common rootkit techniques [22].

For injecting code into the kernel or kernel extensions, the adversary may obtain a higher privilege level (such as root access) and inject the malicious payload by patching the binaries on disk, hooking the system call table, or patching (overwriting) code sections and read-only data sections in memory. For user-mode applications, the adversary may patch binaries on disk or inject a malicious library in the address space of a running legitimate user-mode process or service, by manipulating the code section in memory.

Similar to related tools, if malware is aware of the RAI agent on the victim systems, malware could attack the agent or intercept its network communication. For example, since RAI relies heavily on hashing memory contents, an attacker could carefully craft a manipulation such that both the original and the manipulated memory yield the same hash value. To minimize the possibility of such attacks, RAI hides its trace in the system (e.g., by hiding the driver code and its network activity) and uses a minimal set of APIs that are not common targets of malware manipulation.

3.2 RAI Architecture Overview

Since the RAI client component resides in the address space of the OS that runs the monitored application, there is a risk that a rootkit attacks the RAI client. To minimize this risk, the RAI client has a tiny feature set and therefore a tiny attack surface. In RAI many key features are located on the server.

Figure 1 gives a high-level overview of RAI's architecture, using an example setup of m machines running a mix of applications. The RAI back-end is a user-mode application that resides outside the monitored machines, i.e., on a remote host. Each of the m monitored machines has installed

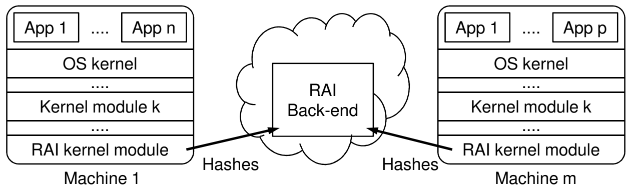

Figure 1: Example configuration of RAI, monitoring m application instances.

a RAI client component, i.e., the RAI agent. The RAI agent is a kernel module that sends at short random intervals to the RAI back-end server its machine's configuration. The configuration includes its processor type, OS version, and a hash of all code segments and all read-only data segments of all processes that are currently in physical main memory.

RAI-like monitoring tools are susceptible to *scrubbing attacks*, in which a memory resident malware predicts monitoring time and attempts to conceal its trace in memory at the time the monitoring agent computes a memory hash [38]. To address such attacks, RAI obtains its memory hashes in random intervals, which are by default 30–90 seconds. To further address the threat of scrubbing attacks and to reduce the RAI client's attack surface, RAI places its random interval generator on its back-end component. To prevent an attacker from inferring hashing activity from network activity, the server's instruction includes a random delay value, which the client uses to decouple the times of incoming network traffic from the start of hashing.

3.3 De-Relocating Virtual Addresses

In any two execution scenarios, a given kernel module or shared library may be loaded at different virtual addresses. For example, this may be due to two machines loading code in different orders. Since some addresses may differ across machines and invocation scenarios, compilers express these addresses relative to an assumed base address. Many recent user-level libraries implement this with position independent code (PIC). PIC adds a level of indirection and replaces addresses in the code segment with a lookup of the actual virtual target address. The lookup is placed in the (writable) data segment and is therefore not hashed by RAI. If a library is compiled to PIC it will therefore yield the same RAI hash value regardless of where it is loaded.

Instead of position independent code, kernel modules and legacy user-level libraries use the traditional technique of load-time relocation using the relocation tables that are available in on-disk binaries, but not in in-memory (loaded) binaries.

When loading such code the operating system fixes the code's base address and replaces ("relocates") each relative virtual address with the then-known absolute virtual target address. After relocation, the same library or kernel module may therefore contain different (absolute) virtual addresses on different machines, which would yield different hash values for the same piece of code.

A straightforward solution to this problem would be to find all such addresses in kernel modules and legacy shared libraries and replace them with zero. While this approach would ensure equal hash values for different relocations, it may also produce false negatives if a rootkit only manipulates such addresses.

To address this problem, RAI "de-relocates" addresses for hashing. RAI replaces each absolute virtual address that points to a library with the corresponding relative virtual address. Specifically, for each absolute virtual address, RAI determines the library and memory segment the address points to and subtracts from the address the segment's base address, thus yielding the de-relocated address.

To find addresses in a library, the current RAI prototype implementation customizes the Distorm disassembler [13], which supports 32-bit and 64-bit Intel processors. Disassembling x86 is undecidable in general [59], so no disassembler will produce perfect results in all cases. However in our experience Distorm produced reasonable results in practice. Distorm also provides convenient access to opcodes and operands via its decomposer feature. While Distorm contains some Linux header files that can only be used in user space it is written in POSIX C (without using OS-dependent APIs). We could thus remove user-mode specific header files from Distorm and use it in RAI's kernel agent.

3.4 Dynamically Loaded Code

De-relocation ensures that library hashes do not differ based on their virtual addresses. However an application's hashes may still differ across executions if the executions use different library load orders. This may happen when two executions solve different tasks, each requiring its own libraries or its own library load order.

Mapping an entire application to a single hash value would lead to frequent problems. For example, the two application instances in Figure 2 have not been manipulated. But they would yield different hash values (Hash_T1 $\neq$ Hash_T2), as only one of them has loaded the lib2.so dynamic library.

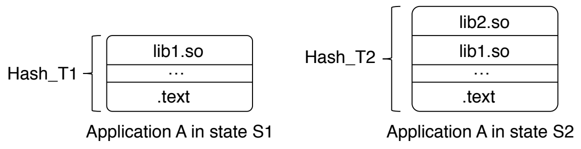

Figure 2: Two application instances yield different "hash of hashes" (Hash_T1 $\neq$ Hash_T2), as only one has loaded the lib2.so dynamic library.

To distinguish such cases from rootkit attacks, RAI associates each page hash with the name of the segment the page belongs to. RAI also computes one hash per segment. This way, RAI can quickly compare segments across machines and locate manipulated pages, even if different libraries are loaded or they have been loaded in different orders.

3.5 2-Level Hash

When characterizing a program's code and read-only data segments as hash values, it is important to determine which parts of the code and read-only data segment to include in the hash. Since at any time some pages of the program's segments may be swapped out to disk, the brute force solution would be to first swap all segments back into main memory and then compute the hash values. However this approach would be very inefficient, as each swap can be time intensive and displace pages other processes may need, triggering further system slowdown. Furthermore, pages on disk are less interesting as they are currently not in use, by neither the application nor any malware.

To hash efficiently in the presence of partially swapped out code, previous work such as SVV just computes a single hash value of a given application using the pages that currently happen to be in memory [50]. However this approach may produce false positives, as on different machines different pages may be swapped out to disk.

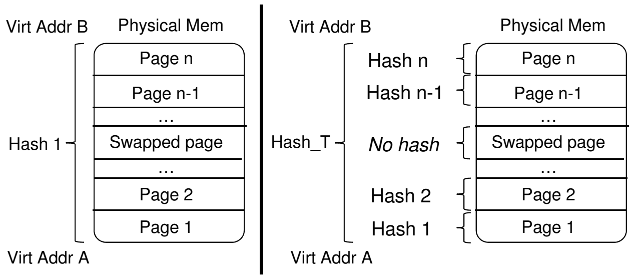

Figure 3: Single-hash of available pages (left) vs. hash of hashes (Hash_T) plus individual page hashes 1 to n (right).

To efficiently deal with swapped-out pages, RAI uses the following multi-level hashing scheme. Figure 3 shows an example of hashing the N pages in virtual address range A to B. On the left, SVV computes a single hash of all the pages that are currently in physical memory. On the right, RAI first locates all the pages of a given program's code and read-only data sections that are currently in physical memory and therefore not swapped out. RAI then computes the hash of each of these pages and associates each hash value with the page's virtual address. Further RAI computes the hash Hash_T of these hash values.

3.6 RAI Agent's Attack Surface

The RAI client-side component is designed to have a minimal feature set and therefore a minimal attack surface. Specifically, RAI's client-side agent calls APIs and macros only to access and hash memory, manipulate strings, communicate via the network, and perform OS-level synchronization.

As a concrete example of the RAI agent's small footprint, all OS functions RAI's agent calls provide essential and widely available low-level OS features. Implementing RAI for other Linux versions and OSs such as Windows is thus straight-forward, since these other OSs all provide the features RAI uses.

Besides its small footprint, all standard techniques apply for hiding RAI's presence from malware on client machines. Example techniques include unlinking the RAI agent from the list of running kernel modules, randomizing the name of the RAI agent before deploying it, and similarly randomizing its code and therefore its own hash signature.

3.7 RAI's Server Component

At a high level the server's main tasks are communicating with RAI's client-side agents and dealing with the hashes it receives from these clients. To keep the attack surface of RAI's clients as small as possible, the server also takes on such tasks as keeping track of all participating machines, initiating client-server communication, and deciding random communication intervals. For each participating client, the server aims to maintain at least two hash values, the current and the previous one.

After sending a request to its clients, the server waits for a configurable duration for client responses (5 seconds by default). If the server only receives partial (or no) hashes from some clients, it sends another request to these systems. If a client does not respond to server requests for a long time (e.g., 10 requests), the server produces an alert. This alert indicates that the client is shut down, has a network problem, or is possibly under malware attack.

4. EXAMPLE IMPLEMENTATION: LINUX

To evaluate RAI, we have implemented its main components as prototypes for the Linux platform. This section describes key implementation details that allow an initial empirical evaluation.

The goal of this prototype implementation is not to provide a highly efficient and secure large scale distributed software, but to allow rapid evaluation. We therefore took a few shortcuts in the implementation. Notably, in the prototype the client-side component is a kernel module that listens directly on the network for server commands, which poses obvious security problems. This is not a fundamental limitation, though, and can easily be replaced with handling network communication entirely in user-mode.

Similarly, the prototype implementation is missing advanced features such as failure recovery, security and authentication, high availability, scalability and performance management. As another example, our current prototype implementation does not actively prevent denial of service attacks on the server component. Addressing these issues is part of future work.

4.1 Obtaining Virtual Address Ranges

The RAI agent is a kernel module and therefore has full access to the user system's memory. When the agent starts executing it creates a single-threaded work-queue [5] that listens to a predefined port and operates based on the commands it receives from the RAI back-end.

To obtain the virtual address range of Linux kernel code and read-only data, existing rootkit detection and protection tools consult the Linux kernel's symbol table [47], which is stored in the *system.map* file. As usual, for each kernel function and variable the symbol table maps between name and address. However this approach does not work if the kernel uses run-time address randomization [17].

Instead, memory dumping tools traverse the I/O memory resources to find the kernel code's physical address range [57]. The Linux kernel keeps track of the I/O operations that occur within the address range of physical resources such as RAM. For the OS the RAM's child resources include kernel code, data, and uninitialized data sections (bss). However this approach does not work for the kernel's read-only data.

To obtain the virtual address range of Linux kernel code and read-only data, even under run-time address randomization, RAI calls Linux's *kallsyms* function. The kallsyms function extracts all symbols (e.g., functions and variables) from the kernel and is therefore commonly used by Linux debuggers. The resulting file maps between symbol address and name. For instance, a developer can use this file to extract the entry point address of an internal kernel function within the OS's virtual address space.

In most Linux distributions kallsyms is available. The RAI agent calls kallsyms to extract the address of the symbols that mark the start and end addresses of the kernel code and read-only data segments (`_stext`, `_etext`, `__start_rodata`, and `__end_rodata`).

Outside the kernel, RAI extracts code and read-only data address ranges by traversing the kernel's data structures in the kernel's heap. For example, the Linux kernel maintains linked lists of all processes (`task_struct`) and drivers (`module_struct`) and RAI interprets them to find the required virtual address ranges. Specifically, RAI traverses the above heap structures to identify each segment or *virtual memory area* (VMA) of the kernel, each kernel module, and each user-mode process. RAI checks if a VMA is flagged as executable, which indicates a code segment.

4.2 Hashing Physical Memory

With the virtual addresses de-relocated (Section 3.3), hashing many of the physical Linux kernel memory pages is straightforward as the kernel makes sure that its code and read-only data are in contiguous physical addresses and are never swapped out. So the RAI agent takes the virtual address ranges obtained in Section 4.1 and maps them to physical addresses via standard kernel APIs.

An interesting special case are legacy machines that have more physical than virtual memory. For example, to access main memory beyond 4 GB on a 32-bit x86 machine with its 4 GB virtual address space, the Linux kernel dynamically maps a set of virtual addresses to a larger set of physical addresses, which is also called HIGH_MEMORY or the kmap segment [39].

After the kernel maps one of these virtual addresses to a new physical address, the page at the old target address may still be valid. But now no virtual address maps to it. Linux memory-maps code into a process's virtual address space. This means that, despite being currently unreachable from the virtual addresses, such a page may still be in Linux's *page cache*. To solve this issue and obtain the hashes of all in-memory pages of an application, RAI reads those pages directly from the page cache.

The current implementation uses the MD5 hashing algorithm since MD5 is fast. But the choice of hash function is not important to RAI. We could easily replace MD5 with another hash function, if MD5's level of hash collisions are deemed a problem [4]. For example, switching to a stronger hash-function such as SHA-2 or SHA-3 would lower the chance of such hash collisions.

4.3 Sending Hashes to the Back-end

Each RAI agent has a unique *RAI identifier*. In the current implementation this identifier is fixed and assigned before the RAI agent is installed. The RAI agent produces one message to send the hashes to the back-end application. Every message contains the user system's RAI identifier and the names of kernel, kernel extensions, and applications, followed by their respective 16 byte MD5 hashes. RAI names the hashes of the kernel "kernel code" and "kernel data".

As mentioned earlier, RAI sends the hashes of loaded libraries or injected code separately from the code segments of user-mode applications. Thus, after including the hash of text segment of a process, RAI writes names and hashes of loaded libraries or injected codes.

5. BUILT ON RAI: TDOIM

RAI could be used for several software maintenance tasks. However, in this paper we focus on a proof-of-concept application that detects ongoing rootkit attacks.

Detecting code-injection rootkit attacks conducted with new malware is notoriously hard, even on modern systems [8]. Such attacks may be launched, e.g., by zero-day attacks. New malware may inject malicious behavior into a trusted application, may be persistent on the infected system, and may manipulate the infected system to hide itself effectively. Recent examples of code-injection rootkit attacks include Stuxnet, Duqu, and Flame [34, 30, 56].

For such attacks, existing malware detection approaches are not effective for legacy systems. For example, a widely accepted best practice for detecting code-injecting rootkits in legacy applications is running host-based third-party antivirus tools and they are in wide use [9]. However, current antivirus tools are essentially complex high-privilege extensions of the underlying operating system. Such an extension dramatically increases the malware attack surface—many attacks on antivirus tools have been described [60, 37, 36].

Also, it may take weeks before a current antivirus tool detects a new malware attack, since most antivirus tools rely at least partially on comparing application files to a blacklist of known malware signatures. For a new malware attack it takes time for antivirus vendors to discover the malware, distill it into signatures, and push the signatures to protected hosts. For example, in a 2007 study on some 8k malware samples, young (less than one week old) malware samples went undetected at a rate from over 20% to over 60%, depending on the tool vendor [40]. A 2014 study had similar results, e.g., antivirus tools could still not reliably identify malware samples that were several months old [14].

Cloud-based antivirus approaches [40] remain blacklist-based, which may leave high-value applications exposed to new malware attacks for weeks. To support legacy applications, extensions of host-based antivirus tools catalog the files of all applications into a whitelist[1]. But this approach still has a large attack surface and may whitelist malware that is already on the host.

Other anti-malware work also does not adequately support legacy applications, since it places strong assumptions on the monitored applications. For example, recent techniques assume virtualization [53, 47, 48, 11, 21, 27, 58], certain VMs [53], or special hardware such as TPM or PCI add-in cards [45, 32, 38, 16, 51, 25].

5.1 Many Homogeneous Instances

In addition to RAI's assumptions (Section 3.1), TDOIM also assumes that many application instances are running at the same time. This assumption is met, for example, by large distributed applications running many application instances (perhaps in data centers or in the cloud) or by many users running instances of the same stand-alone client-side application.

As an example of this assumption, Figure 2 shows a false warning if lib2.so is a benign library. However a real-world TDOIM setup has typically many instances and therefore more than two instances running such a benign library. Since rootkits typically spread relatively slowly across machines, TDOIM has a time window in which a rootkit is only present on a relatively small number of machines. Thus TDOIM uses a configurable threshold value (by default 10%). An occurrence above the threshold indicates a benign library, whereas

[1]For example: http://www.mcafee.com/GTITurnItOn

a below-threshold occurrence indicates a rootkit infection.

TDOIM also works best if there are relatively few application variants. Specifically, we assume that the applications are relatively homogeneous in terms of their version and how they have been compiled. This is often the case, as developers tend to use the same compiler for long periods of time and only a few versions of a given legacy application are in wide use.

5.2 TDOIM Use-Cases

TDOIM has two main use-cases, which differ in the type of monitored applications and their owners. (1) First, an administrator of many legacy application instances, which are possibly distributed over several data centers, suspects a malware attack on some instances. So the admin installs the client-side agent on each machine, e.g., remotely as a kernel extension or device driver. The server component detects outliers, which triggers the admin to either inspect or shut down the outliers.

In the second use-case (2) an end-user may suspect a malware infection on her machine and installs the TDOIM client on her local machine. TDOIM will then compare her hash values with those of other end-users. This use-case has stronger privacy requirements for transmitting hashed memory regions to the server and a lower tolerance for false warnings, as the end-user can only inspect a single instance for possible malware infection.

Both use-cases have in common that upon completion the TDOIM client may be uninstalled from the affected machines. Neither installation nor uninstallation require recompilation or application restarts.

5.3 TDOIM Architecture

To detect ongoing rootkit attacks, we built on RAI a tiny distributed on-demand integrity monitor (TDOIM). Based on the data received from RAI clients, the TDOIM back-end divides the user systems into groups. Within a group, each member has the same combination of architecture and OS version. Within each group, TDOIM compares the hashes of the various applications to detect outliers. For example, the two application instances in Figure 2 have different hash of hashes (Hash_T1 $\neq$ Hash_T2). TDOIM thus compares the segments' hashes (.text, libc.so, ld.so, lib1.so, and lib2.so) and identifies lib2.so as an outlier.

The TDOIM server detects outliers only within the group of reported hashes received by the RAI server and thereby does not require any prior knowledge about original binaries, file signatures, blacklists, or whitelist. More importantly, our server detects potential malware attacks *immediately* and does not have to wait until a third party has released corresponding malware signatures.

Our voting-based scheme works because a malware attack usually spreads relatively slowly across the various locations running the monitored application. While the malware has only infected a minority of the monitored application instances, our scheme can detect the malware infection as outliers.

TDOIM aims for efficient communication. For example, by default a RAI agent only sends a single hash (of all hashes). Only if the server flags a hash as an outlier, the server requests more detailed hash values, to compare the hashes of all pages in physical memory across different systems. In addition to removing the false warnings of SVV-type approaches, RAI's hash of hashes technique can therefore also pinpoint the pages that have been manipulated, if any.

6. EVALUATION

To evaluate RAI we evaluate its proof-of-concept application TDOIM. To evaluate TDOIM, we ask if TDOIM shows promise for online rootkit detection. This question has two main facets, runtime overhead and true vs. false positives. True and false negatives are less relevant, since rootkit detection is heuristic in nature and TDOIM can be combined with other rootkit detection approaches. We therefore investigate the following four research questions (RQ), expectations, and hypotheses (H).

RQ1: Does its runtime overhead preclude TDOIM from detecting rootkits online? We do not expect TDOIM to be applicable for all settings, because, for example, TDOIM performs a relatively expensive analysis to compare kernel addresses across clients to compensate for code load order and OS address space layout randomization. H1: TDOIM can be useful in settings in which client machines still have significant available computational resources.

RQ2: Does its false positive rate preclude TDOIM from being used in production? Since disassembly of x86 binaries is undecidable, we cannot expect zero false positives. H2: TDOIM's false positive rate is typically greater than zero but on average can remain below 10%.

RQ3: Can TDOIM detect common types of kernel and user level rootkit attacks? We expect TDOIM to detect common rootkit attack types as they occur. H3: TDOIM can detect common rootkit attack types online.

RQ4: Does TDOIM scale to geographically widely distributed deployments? We expect TDOIM can scale to setups that have realistic communication delays. H4: TDOIM can detect rootkit attacks within a few minutes, even if the TDOIM-monitored applications are running on geographically widely distributed machines.

We compare RAI and TDOIM qualitatively to competing approaches throughout the paper. A full evaluation with a quantitative comparison is subject to future work.

6.1 Subjects: Kernel and User Level Rootkits

Table 1 lists the rootkits used in the experiments. They are a mix of third-party samples and our own development. The rootkits operate both in user and kernel mode and manipulate both disk contents and main memory. The rootkits perform a variety of attacks—they exchange libraries, inject code, divert execution, and change both kernel code and data. Following is a high-level description of each rootkit.

Exchanging libraries via LD_PRELOAD: In this approach the rootkit exchanges a program's libraries at load time with malicious libraries, i.e., to divert some of the program's library function calls such as system calls (which are calls to the C standard library libc). Specifically, the rootkit changes the order in which a new process loads libraries. To change the load order, the rootkit sets the LD_PRELOAD environment variable of a new process. Linux loads the libraries listed in LD_PRELOAD before all other libraries, even before libc.

As an example implementation of this rootkit, we set LD_PRELOAD to divert the execution of the *open file* system call to a malicious *open file* function implemented in our injected library. We expect TDOIM to detect this root-

Table 1: Rootkit subjects run either in user (u) or in kernel (k) mode; KV = Linux kernel version; CPU = 32 vs. 64 bit; Loc = main memory (m) vs. disk (d).

Rootkit		KV	CPU	Loc	Attack
LD_PRELOAD	u	3/2.6	32/64	m	Exchange libraries
Jynxkit	u	2.6	32	m	Exchange libraries
Patch dynamic loader	u	2.6	32	d	Inject code
Attach to process	u	2.6	32	m	Divert execution
Syscall hooking	k	3	64	m	Change kernel data
Suterusu	k	3	64	m	Change kernel code

kit by comparing the hash code of two program instances, which will differ due to the different loaded libraries.

Jynxkit—Exchanging libraries via ld.so.preload: Beside the process-specific LD_PRELOAD, Linux also checks the contents of the /etc/ld.so.preload file for user-level libraries that should be loaded before libc. Manipulating this file thus changes the library load order of all future user-mode processes.

For example, the third-party Jynxkit rootkit adds its malicious ld_poison.so library to ld.so.preload and thereby diverts several system calls including open. Jynxkit remains undetected by several common anti-rootkit tools, because they only detect library injection attacks via LD_PRELOAD.

Patching the user-mode program loader: The previous approaches leave the OS binaries intact. But a rootkit can also directly rewrite on disk the binary file of the OS's standard ld-linux.so dynamic user-mode program loader, e.g., using ELF hooker [3]. This rootkit adds to the loader binary on disk malicious shell-code. This patched loader sets the entry point of a to-be-loaded program to some malicious code. The loader also adds code that after the malicious code execution transfers execution to the program's originally intended benign entry point. We use the rootkit to inject code into the cat application.

InjectSO—Diverting process execution: Instead of manipulating future processes, this attack uses an OS's standard debugger support to attach to a running process and diverts its execution [10]. This attack typically diverts execution to the OS's loader and dynamically loads a malicious library. The attack then changes some of the program's function pointers to divert function calls to the just injected malicious library.

We used the InjectSO rootkit to inject into the address space of a running standard Linux cat application a malicious library. In this type of attack, TDOIM notices that the cat instances running on different machines yield different hash values, due to the rootkit manipulating cat's code segment.

Hooking the system call table: A common rootkit technique first locates kernel data structures such as the system call table that point to important system functions. By changing the pointers, attackers can divert or "hijack" system calls to malicious code. These rootkits thus hijack the kernel execution without changing any kernel code. Here we diverted system calls with the Linux syscall hooker [6]. For this attack we expect TDOIM to detect a difference in the kernels' read-only data segments.

Suterusu—In-line function patching: Besides pointers, a rootkit can of course also directly change the kernel code to change its control-flow. For example, in in-line patching a rootkit may overwrite a function's prologue with a jump to malicious code and thereby divert kernel execution [22]. We evaluate the effectiveness of TDOIM against the Suterusu rootkit [12]. We expect TDOIM to notice a changed hash value of the code segment that contains the patched function.

6.2 Experimental Setup

We conducted two sets of experiments. The first set explored RQ1 to RQ3 on a small-scale setup. While limited, these experiments provided interesting initial insights. The second set explored RQ4 on a geographically distributed setup on Amazon AWS.

All experiments use a crude approximation of a legacy system setup. That is, we mainly make sure that we do not use any modern security features such as TPM.

Setup for RQ1 to RQ3: We set up a testbed of 20 user systems and installed a TDOIM agent in each. These 20 user systems communicated with one TDOIM server. For these experiments we side-stepped (non-local) network requirements and place all components on the same host, each running in its own virtual machine. While this choice does not capture typical application installation scenarios, it is still valuable as a baseline for future experiments.

We further focus the experiments on a baseline via the following experimental choices. First, we run the same version of each application and each kernel module on each client. Second, for these experiments we do not explore situations in which applications dynamically load a large number of libraries during the experiments. On the other hand we do not prevent applications from loading libraries.

Each client ran on an Ubuntu Linux VMWare VM with kernel versions 2.6 or 3, 32-bit or 64-bit Intel processor, and 512 MB RAM. The TDOIM server ran on a 64-bit Ubuntu VMWare VM with a version 3 kernel and 4 GB RAM. The host system was Debian Linux running on a 2.33 GHz Xeon machine with 32 GB RAM. The server and each client used their Ubuntu OS in its default installation and configuration. This meant that running in each OS were 100 user-mode applications and 41 kernel modules.

For each experiment, we divided the 20 clients into two groups. In each group each user system has an identical configuration, i.e., the same hardware and OS configuration. For each experiment we further infected between one and three VMs with a given rootkit.

AWS setup for RQ4: To explore RQ4, TDOIM monitors application instances running across the world. TDOIM can run on both VMs and physical machines. We used VMs in this experiment because it allowed us to quickly deploy TDOIM to machines across the world, via Amazon Web Services (AWS). Specifically, we ran TDOIM on up to 60 virtual machines equally distributed over 10 AWS regions, i.e.,

Table 2: Breakdown of TDOIM's average runtime overhead; Loc = TDOIM client (c) vs. server (s).

TDOIM activity	Target	Loc	Slowdown (%)
Hash	user-mode app	c	8
Hash	kernel code/data	c	3
Hash + de-relocate	kernel modules	c	20
Compare hashes	20 VMs	s	15
Log MA_INFO	20 VMs	s	42

in Virginia, California, Oregon, Ireland, Germany, Japan, South Korea, Singapore, Australia, and Brazil.

Due to the experiment's costs, we focused on a single representative rootkit, i.e., the Section 6.1 one manipulating LD_PRELOAD. Since TDOIM performs the same operations regardless of the rootkit, we do not expect different results for different rootkits.

In this experiment, the TDOIM server and clients are running on Linux Ubuntu 14.04 LTS. Each client is running 90 user-mode applications with 0.6 GB RAM. The server has 30 GB RAM and runs in Oregon. As a baseline, this experiment only communicates the hash of hashes results to the server. To provide a conservative scenario in terms of delay until TDOIM can detect the attack, this experiment has only a single infected client, the client is in Japan, and the server checks its hash last.

6.3 Experimental Results

6.3.1 *RQ1: Moderate Runtime Overhead*

To evaluate the performance of the TDOIM agent, we measure how much time it takes to compute the page hashes of user-mode applications and kernel data and code. For kernel modules, we measure the overhead of page hashing and de-relocation.

To evaluate the performance of the TDOIM back-end application, we calculate the overhead of comparing hashes of the kernel and 141 applications (100 user-mode applications and 41 kernel modules) over 20 virtual machines. Also, we customize the back-end application to produce the log of hashes for each virtual machine and we measure the overhead of the log producing task.

Table 2 summarizes our measurements. The table contains two kinds of slowdown numbers. The server (s) slowdown measurements represent a given task's slowdown of the TDOIM server component. The client (c) measurements are slowdown of the monitored application.

As a baseline for the server measurement, the biggest overhead (42% slowdown) was logging the hashes received from the clients to disk. This task is only turned on when debugging the TDOIM server component. Comparing the hash values resulted in a 15% slowdown. On the client side, the sum of all TDOIM tasks lead to a slowdown of 31% of the monitored client application. While this is a non-negligible slowdown, it is still feasible for many application scenarios. The TDOIM prototype implementation could also be further optimized, possibly leading to smaller slowdown numbers.

6.3.2 *RQ2: Few False Positives*

To explore research questions RQ2 and RQ3 we carefully analyzed each TDOIM-produced alert. Specifically, we dumped the pages that yielded different hash values on different clients. We then manually analyzed the pages to determine if different hash values were caused by a rootkit.

Table 3: The 7 kernel modules that produced false alerts and their number of pages; FH = false hashes; FP = false positives.

Kernel module	Pages	FH	FP (%)
e1000	22	3	13.6
vmwgfx	18	2	11.1
ttm	11	1	9.0
drm	33	1	3.0
bluetooth	55	3	5.4
rfcomm	9	1	11.1
psmouse	16	2	12.5
All 41 modules	314	13	4.1

A false positive occurs when TDOIM produces an alert about different hash values from two or more client agents that are not caused by a rootkit manipulation. Such false positives could occur for different reasons, such as errors in our TDOIM prototype implementation. For our experiments we expect these false positives to stem from the insufficient reverse engineering of memory addresses in client side kernel modules.

Indeed, during our experiments all of TDOIM's false positives came from kernel modules. Recall that kernel modules are relatively challenging to handle due to the OS's address space layout randomization and possibly different module load order.

Of the 41 kernel modules running in our standard 64 bit Ubuntu 12.04 LTS installation, we received in our experiments false positives on 7 kernel modules. These false positives are summarized in Table 3. For example, the e1000 kernel module uses 22 pages for its code and read-only data sections. The size of each page is 4kB. TDOIM received conflicting hashes and thus false warnings on 3 of these 22 pages, which corresponds to a 13.6% ratio. Across all 314 pages of the 41 kernel modules, false warnings occurred in 13 pages or 4.1% of pages. Reducing the false positive rate is subject to future work.

6.3.3 *RQ3: 100% Rootkit Detection Rate*

Besides false positives, the experiments produced many true positives. That is, in each experiment, TDOIM successfully pinpointed the VM, application, and page that were infected by a rootkit, both at the user and kernel level.

So, in summary, in our experiments on several user and kernel mode rootkits, TDOIM achieved with moderate overhead and a relatively low false positive rate a 100% rootkit detection rate.

6.3.4 *RQ4: Scaling to Larger Deployments*

We can break the delay from rootkit attack to attack notification into two parts. The first part is the random communication interval determined by the server. In our experiments this interval ranged from 0 to 90 seconds. More interesting is the remaining delay, which contains the client agent's hash computation, network communication delays, and the server's hash processing.

Table 4 summarizes the second component of the attack to notification delay. Each experiment was run on a server with a set of clients, ranging from 6 to 60 clients equally distributed over 10 AWS regions. This delay increased with the number of clients, from 0.4 to 4.4 seconds.

While this roughly linear delay increase appears to limit

Table 4: Scaling a TDOIM deployment distributed over 10 regions in North and South America, Europe, Asia, and Australia; M = total client machines; D = delay until attack notification in seconds (in addition to TDOIM's randomized communication intervals).

M	6	12	18	24	30	36	42	48	54	60
D	0.4	1.0	1.1	1.6	1.9	2.1	2.9	3.3	3.9	4.4

TDOIM's scalability, parts of this scaling behavior can be explained by our naive server implementation. For example, the server currently creates a separate thread for each client and has not been optimized for scalability or performance. On the positive side, even in its current naive implementation TDOIM scales to a highly distributed setup of dozens of application instances.

7. RELATED WORK

Beyond the related work discussed in Section 1, also related are static instrumentation approaches. However these are not well-suited for legacy applications, as static instrumentation requires recompiling and restarting the application. Besides the business impact of restarts, the lack of source code and debug symbols makes this task undecidable [59]. As another issue, static approaches do not support self-modifying code or malware-induced code changes [24].

For TDOIM, most closely related are cloud-based antivirus approaches, pioneered by CloudAV [40, 41, 61, 26]. These approaches reduce the attack surface for malware on the monitored client, by shifting much of the detection functionality from the client to a cloud-based server, which also makes them attractive for resource constrained (e.g., mobile) devices [41, 61, 26]. However cloud-based antivirus approaches still rely on manually (and thus relatively slowly) curated blacklists.

Similar to TDOIM, a rootkit detector built on Pioneer also periodically computes hashes of the kernel code and read-only data and sends these hashes to a server component [54, 33, 28, 2, 20]. Pioneer does not rely on virtualized OS stacks or special hardware. Instead, Pioneer times its execution and thereby detects rootkits. However, Pioneer requires prior knowledge of the installed software and makes strong assumptions about machine and communication speed. While these assumptions have been partially relaxed, they do not support legacy applications communicating over public networks [33, 28].

Traditional malware detection approaches are similar to antivirus tools, as they compare relevant data against existing white-lists or black-lists and can check for an entire class of attacks [29]. However, they focus exclusively on the integrity of the kernel, have a large attack surface, and cannot be applied in an ongoing malware attack. For example, when the kernel loads a device driver, earlier work analyzes the driver with static symbolic execution, to check if the driver matches given patterns of malicious behavior [29]. Nickle hashes kernel code and prevents execution of code that does not match this hash [47]. Poker uses Nickle to detect rootkit execution at runtime and then captures a trace of the rootkit execution [48].

However detecting kernel-level attacks is not sufficient. For example, by employing user-mode rootkit techniques, attackers can compromise systems, run malicious payloads, and remain undetected from these approaches. By monitoring user-mode and kernel-mode applications as well as the whole operating system, TDOIM is effective against common user- and kernel-mode rootkits.

Several integrity checking techniques such as Nickle and Vigilare rely on modern infrastructure that is not available in legacy systems, such as virtualization (e.g., based on a hypervisor or software-based virtualization) [53, 47, 48, 11, 21] and specific hardware such as TPM or PCI add-in cards [45, 32, 38, 16, 51, 25, 35, 52].

File integrity checkers such as Tripwire rely on an accurate comparison of runtime memory contents with original on-disk binaries [51, 27]. Such a comparison requires either the files' original signatures or a clean system state, to ensure that no binary is patched by malware. In other words, such tools cannot be used in an ongoing attack. For example, the system virginity verifier (SVV) is a cross-view based Windows rootkit detection approach that checks if code sections of important system dlls and system drivers in memory are consistent with their on-disk binaries [50].

8. LIMITATIONS AND FUTURE WORK

Not covered by our threat model are stack manipulations. For example, in return-oriented programming (ROP) attackers hijack the application control flow without code injection [49, 23], by instead overwriting an application's stack.

Detecting ROP attacks with TDOIM would require monitoring and comparing application control-flows across machines. Moreover, several other approaches mitigate ROP attacks by monitoring the control flow or via compiler-level approaches for building less vulnerable binaries [15, 43].

Also not covered by our threat model are manipulations of dynamic data structures in the kernel's heap such as direct kernel object manipulation (DKOM). The most common DKOM attack is manipulating the linked lists the kernel uses to keep track of kernel modules and processes, to hide the presence of malware in the victim system.

While TDOIM does not detect such heap manipulations, many real-world attacks use DKOM only to hide their presence. In addition, such attacks rely on rootkits covered by our threat model to attack applications. TDOIM can thus detect such combined attacks, even if their traces are hidden via DKOM.

9. CONCLUSIONS

Remotely determining which precise code is running on which machines is hard. This is especially true if the monitored machines lack modern security features and may be under malware attack, since in such a scenario the malware may have already manipulated applications and operating systems. Existing approaches to this problem are heavyweight and have a large attack surface, which is frequently attacked by both applications and malware.

To address this problem, this paper introduced RAI, a light-weight code monitoring tool that is especially well-suited for legacy systems. While potentially useful for many software maintenance tasks, this paper applied RAI for detecting ongoing rootkit attacks. Specifically, in our experiments on several user and kernel mode rootkits, our approach achieved with moderate overhead and a relatively low false positive rate a 100% rootkit detection rate.

Acknowledgments

We would like to thank Matthew Elder, Nathan Evans, Azzedine Benameur, and the anonymous reviewers for their helpful feedback. This material is based upon work supported by the National Science Foundation under Grants No. 1117369 and 1527398.

10. REFERENCES

[1] Trusted Platform Module. ISO/IEC 11889, May 2009.

[2] F. Armknecht, A. Sadeghi, S. Schulz, and C. Wachsmann. A security framework for the analysis and design of software attestation. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 1–12. ACM, Nov. 2013.

[3] N. Bareil. ld-linux.so ELF hooker. http://justanothergeek.chdir.org/2011/11/ld-linuxso-elf-hooker/. Accessed Sept. 2016.

[4] J. Black, M. Cochran, and T. Highland. A study of the MD5 attacks: Insights and improvements. In *Proc. 13th International Workshop on Fast Software Encryption (FSE)*, pages 262–277. Springer, Mar. 2006.

[5] D. P. Bovet and M. Cesati. *Understanding the Linux kernel.* " O'Reilly Media, Inc.", 2005.

[6] E. Bradbury. linux-syscall-hooker. https://github.com/ebradbury/linux-syscall-hooker. Accessed Sept. 2016.

[7] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. USENIX Annual Technical Conference*, pages 15–28. USENIX, June 2004.

[8] cert.pl. More human than human: Flame's code injection techniques. https://www.cert.pl/en/news/single/more-human-than-human-flames-code-injection-techniques/, Aug. 2012. Accessed Sept. 2016.

[9] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen. SplitScreen: Enabling efficient, distributed malware detection. In *Proc. 7th USENIX NSDI*, Apr. 2010.

[10] S. Clowes. Injectso: Modifying and spying on running processes under linux and solaris. In *Blach Hat Europe*, Nov. 2001.

[11] M. Conover and T.-c. Chiueh. Code injection from the hypervisor: Removing the need for in-guest agents. Black Hat USA, July 2009.

[12] M. Coppola. Suterusu rootkit: Inline kernel function hooking on x86 and ARM. https://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/. Accessed Sept. 2016.

[13] G. Dabah. Distorm: Powerful disassembler library for x86/amd64. https://github.com/gdabah/distorm. Accessed Sept. 2016".

[14] Damballa. State of infections report: Q4 2014. http://landing.damballa.com/state-infections-report-q4-2014.html. Accessed Sept. 2016.

[15] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proc. 6th ACM Symposium on Information, Computer, and Communications Security*, pages 40–51. ACM, 2011.

[16] L. Duflot, D. Etiemble, and O. Grumelard. Using CPU system management mode to circumvent operating system security functions. *CanSecWest/core06*, 2006.

[17] J. Edge. Kernel address space layout randomization. Oct. 2013.

[18] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. Technical Report RC25482, IBM Research, July 2014.

[19] M. N. Gagnon, S. Taylor, and A. K. Ghosh. Software protection through anti-debugging. *IEEE Security & Privacy*, 5(3):82–84, May 2007.

[20] A. Ghosh, A. Sapello, A. Poylisher, C. J. Chiang, A. Kubota, and T. Matsunaka. On the feasibility of deploying software attestation in cloud environments. In *Proc. 7th IEEE International Conference on Cloud Computing (CLOUD)*, pages 128–135. IEEE, June 2014.

[21] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 279–290. ACM, Mar. 2011.

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

[23] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proc. 18th USENIX Security Symposium*, pages 383–398. USENIX, Aug. 2009.

[24] Y. Hwang, T. Lin, and R. Chang. Disirer: Converting a retargetable compiler into a multiplatform binary translator. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4), Dec. 2010.

[25] T. Jaeger, R. Sailer, and U. Shankar. Prima: Policy-reduced integrity measurement architecture. In *Proc. 11th ACM Symposium on Access Control Models and Technologies*, pages 19–28. ACM, 2006.

[26] C. Jarabek, D. Barrera, and J. Aycock. Thinav: Truly lightweight mobile cloud-based anti-malware. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 209–218. ACM, 2012.

[27] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proc. 2nd ACM Conference on Computer and Communications Security (CCS)*, pages 18–29. ACM, 1994.

[28] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *Proc. IEEE Symposium on Security and Privacy (Oakland)*, pages 239–253. IEEE, May 2012.

[29] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proc. 20th Annual Computer Security Applications Conference*, pages 91–100. IEEE, 2004.

[30] Laboratory of Cryptography and System Security (CrySyS). Duqu: A stuxnet-like malware found in the wild. Technical report, Budapest University of

Technology and Economics, Oct. 2011.

[31] A. Lineberry. Malicious code injection via /dev/mem. Black Hat Europe, Mar. 2009.

[32] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proc. ACM Workshop on Scalable Trusted Computing*, pages 21–29. ACM, Nov. 2007.

[33] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: Tamper-proof code execution on legacy systems. In *Proc. 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 21–40. Springer, July 2010.

[34] A. Matrosov, E. Rodionov, D. Harley, and J. Malcho. Stuxnet under the microscope. *ESET LLC (September 2010)*, 2010.

[35] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. EuroSys*, pages 315–328. ACM, Apr. 2008.

[36] B. Min and V. Varadharajan. A novel malware for subversion of self-protection in anti-virus. *Software: Practice and Experience*, 46(3):289–431, Mar. 2016.

[37] B. Min, V. Varadharajan, U. K. Tupakula, and M. Hitchens. Antivirus security: Naked during updates. *Software: Practice and Experience*, 44(10):1201–1222, Oct. 2014.

[38] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: toward snoop-based kernel integrity monitor. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 28–37. ACM, 2012.

[39] D. Mosberger and S. Eranian. *IA-64 Linux Kernel: Design and Implementation*. Prentice Hall, Feb. 2002.

[40] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-version antivirus in the network cloud. In *Proc. 17th USENIX Security Symposium*, pages 91–106. USENIX, July 2008.

[41] J. Oberheide, K. Veeraraghavan, E. Cooke, J. Flinn, and F. Jahanian. Virtualized in-cloud security services for mobile devices. In *Proc. 1st Workshop on Virtualization in Mobile Computing (MobiVirt)*, pages 31–35. ACM, 2008.

[42] R. A. Olsson, R. H. Crawford, and W. W. Ho. Dalek: A GNU, improved programmable debugger. In *Proc. USENIX Summer Technical Conference*, pages 221–232. USENIX, June 1990.

[43] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proc. 26th Annual Computer Security Applications Conference*, pages 49–58. ACM, 2010.

[44] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *Proc. 31st IEEE Symposium on Security and Privacy (Oakland)*, pages 414–429. IEEE, May 2010.

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

[46] Ponemon Institute. 2014 cost of data breach study: Global analysis, May 2014.

[47] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proc. 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 1–20. Springer, Sept. 2008.

[48] R. Riley, X. Jiang, and D. Xu. Multi-aspect profiling of kernel rootkit behavior. In *Proc. 4th ACM European Conference on Computer Systems (EuroSys)*, pages 47–60. ACM, Apr. 2009.

[49] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2:1–2:34, Mar. 2012.

[50] J. Rutkowska. System virginity verifier. In *Hack in the Box security Conference*, pages 2–25, 2005.

[51] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proc. USENIX Security Symposium*, volume 13, pages 223–238, 2004.

[52] D. Schellekens, B. Wyseur, and B. Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 74(1–2):13–22, Dec. 2008.

[53] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350. ACM, Oct. 2007.

[54] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16. ACM, Oct. 2005.

[55] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, Mar. 2012.

[56] sKyWIper Analysis Team. sKyWIper (a.k.a. flame a.k.a. flamer): A complex malware for targeted attacks. Technical report, Budapest University of Technology and Economics, May 2012.

[57] J. Sylve. Lime: Linux memory extractor. https://github.com/504ensicslabs/lime. September 2016.

[58] J. Torrey. MoRE shadow walker: TLB-splitting on modern x86. In *Black Hat USA Briefings*. UBM Tech, Aug. 2014.

[59] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. M. Thuraisingham. Differentiating code from data in x86 binaries. In *Proc. European Conference on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, pages 522–536. Springer, Sept. 2011.

[60] F. Xue. Attacking antivirus. In *Black Hat Europe*, Mar. 2008.

[61] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders. Secloud: A cloud-based comprehensive and lightweight security solution for smartphones. *Computers & Security*, 37:215–227, 2013.