Slick: An Intrusion Detection System for Virtualized Storage Devices

Andrei Bacs
Vrije Universiteit Amsterdam
Netherlands
a.bacs@vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
Netherlands
giuffrida@cs.vu.nl

Bernhard Grill
TU Vienna
Austria
bgrill@seclab.tuwien.ac.at

Herbert Bos
Vrije Universiteit Amsterdam
Netherlands
h.j.bos@vu.nl

Abstract

Cloud computing is rapidly reshaping the server administration landscape. The widespread use of virtualization and the increasingly high server consolidation ratios, in particular, have introduced unprecedented security challenges for users, increasing the exposure to intrusions and opening up new opportunities for attacks. Deploying security mechanisms in the hypervisor to detect and stop intrusion attempts is a promising strategy to address this problem. Existing hypervisor-based solutions, however, are typically limited to very specific classes of attacks and introduce exceedingly high performance overhead for production use.

In this paper, we present Slick (Storage-Level Intrusion ChecKer), an intrusion detection system (IDS) for virtualized storage devices. Slick detects intrusion attempts by efficiently and transparently monitoring write accesses to critical regions on storage devices. The low-overhead monitoring component operates entirely inside the hypervisor, with no introspection or modifications required in the guest VMs. Using Slick, users can deploy generic IDS rules to detect a broad range of real-world intrusions in a flexible and practical way. Experimental results confirm that Slick is effective at enhancing the security of virtualized servers, while imposing less than 5% overhead in production.

1. INTRODUCTION

Server virtualization is the key enabling technology of modern cloud computing platforms. Not surprisingly, recent studies demonstrate that the number of virtualized server workloads is on the rise—already surpassing 70% on common x86 platforms [25]—and so is the server consolidation ratio—with a steady increase of the number of VMs deployed per physical host over recent years [3]. The ever-growing number of virtualized server platforms is crucial to implement efficient resource management strategies and increasingly lower infrastructure costs, but, at the same time, also introduces new unprecedented security challenges for users.

First, the growing number of virtualized server platforms under control of the same organization naturally increases the exposure to intrusions. Second, deploying security hardening and/or monitoring mechanisms inside the guest VMs is difficult or impossible since the administrator may not have access to them (e.g., in public Infrastructure as a service (IaaS) environments). Even in cases where the guest platform is prepared in advance (e.g., in Platform as a service (PaaS) environments), the administrator may still have no control over the actual software and security configuration deployed inside the VM during its execution. Finally, as consolidation ratios increase, malware implanted by intrusions inside a VM may have access to more precious resources shared with other colocated VMs. For example, an infected cloud VM may become part of a botnet involved in a DDoS attack [24], subtracting precious bandwidth to colocated VMs. Even worse, an implanted piece of malware may rely on sophisticated cross-VM side-channel attacks to steal confidential data from other colocated VMs [33].

Despite the challenges, virtualization also offers new opportunities to deploy effective IDS capabilities on server platforms. Since the hypervisor is not freely accessible from the guest VM, even sophisticated malware that can subvert security solutions deployed in the guest (like OS-level protection or antivirus products) cannot easily bypass a hypervisor-hosted IDS. Existing hypervisor-based solutions leverage this intuition, but typically target very specific classes of attacks. For example, a number of solutions specifically focus on kernel rootkit detection [11, 26, 28, 31], which can, however, be easily bypassed by modern bootkits [7]. In addition, these solutions typically incur nontrivial overhead, limiting their applicability in production.

In this paper, we present Slick, a lightweight storage-level intrusion detection system for virtualized environments. Slick monitors low-level I/O operations on virtualized storage devices and alerts users of an intrusion when policy violations are detected. The policies consist of simple IDS rules that

SAC'16, April 4-8, 2016, Pisa, Italy
Copyright 2016 ACM 978-1-4503-3739-7/16/04…$15.00
http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

apply to low-level I/O operations, similar, in spirit, to firewall rules that users frequently configure for infrastructure equipment. For example, a user can easily configure SLICK rules to be alerted whenever the startup code (MBR or bootloader) of a guest VM is modified. SLICK's monitoring and intrusion detection components run entirely sandboxed inside the hypervisor and operate with no interference from and to the guest VMs. This prevents malware with unrestricted access to a guest VM from subverting SLICK, but also allows the user to deploy fully guest-agnostic IDS capabilities when a guest VM is not under their control or not accessible at all. For example, in public cloud environments, deploying an intrusive guest-aware introspection component might raise privacy concerns or even be illegal.

Contributions. We make the following contributions:

- We present SLICK, a hypervisor-based IDS for virtualized storage devices. SLICK incurs low overhead and supports flexible and easy-to-use rules to detect a broad range of real-world intrusions.
- We present a SLICK implementation for kvm [15] and QEMU [6] environments. Our implementation is fully guest-agnostic and requires minimal and isolated modifications to commodity hypervisors.
- We evaluate SLICK on common server workloads and demonstrate that our solution enhances system security, while incurring low overhead even for heavily consolidated production servers.

Though we integrated our implementation with kvm/QEMU, it should be straightforward to port SLICK to other virtualization platforms. To facilitate this and make SLICK immediately available, we will open source our current prototype.

2. THREAT MODEL

We consider powerful attackers who are capable of compromising any part of a guest environment and obtain root privileges. We do assume the integrity of the isolation offered by the virtualization technology itself. In other words, while malicious code may compromise all components of the guest's software stack, including the kernel, it cannot escape the virtual environment itself. In addition, we assume that the attacker uses persistent storage to ensure the system remains compromised. For instance, to survive reboots the attack may compromise the virtual environment's bootstrap code directly (e.g., by rewriting the bootloader), or create an additional account in the `/etc/passwd` file. Both scenarios are listed in Figure 2 which we will use as running examples.

3. MONITORING STORAGE DEVICES

In this section, we first motivate why monitoring storage devices is important (Section 3.1) and then describe how such monitoring can be done (Section 3.2).

3.1 Malware and storage

Data in computer systems is kept on storage devices managed by the operating system using abstractions known as files that are managed by filesystems. However, filesystems do not occupy all sectors of a storage device. For instance, some sectors may contain startup code or other metadata. We refer to all sectors that are inside a filesystem as *light regions*, and to sectors outside any filesystem as *dark regions*.

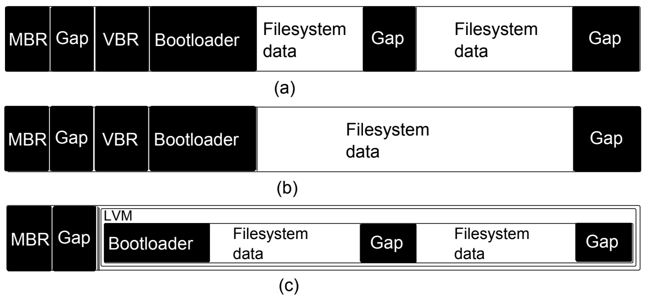

Figure 1: Typical disk layouts for virtualized guests

Dark regions. In a non-virtualized BIOS-based system, the CPU boots in real mode and executes the BIOS[1]. The BIOS reads and passes control to the code of the Master Boot Record (MBR) located within the first 512 bytes of the system's hard disk. The MBR code parses the partition table (PT) to find a "bootable" partition (containing the OS) and typically hands over control to the Volume Boot Record (VBR). The VBR resides within the bootable partition's first 512 bytes and contains further information necessary for booting such as the filesystem parameters and the bootloader's disk location. The VBR code then loads the bootloader's first stage into memory and passes control to it. From the filesystem's perspective, all of these disk regions are dark. Finally, the bootloader loads further code from the disk, switches to protected mode, and executes the kernel.

In a virtualized environment, each guest has its own MBR, bootloader, and filesystem(s). For instance, Figure 1 shows typical disk layouts in virtualized guests. At the top (a), we see an MBR (Master Boot Record) partitioning scheme with two partitions, each containing a filesystem. For example, a guest running Windows 7 with one system partition, plus the Windows 7 hidden partition which is not visible to the user; or a guest running Linux with root and swap partitions. Below that (b), we see an MBR partitioning scheme with a single partition containing a filesystem. Finally, at the bottom (c), we see an MBR partitioning scheme with an LVM [17] volume group that contains two LVM partitions. The dark regions denoted as 'gaps' in the figure are mainly created by the convention to align the data on the physical disk, as implemented in most partitioning tools. The disk layouts in Figure 1 are deliberately not at scale to emphasize the existence of disk regions that are located outside the filesystems maintained by the OS and thus inaccessible by regular users during normal operations.

Even though these regions are small in size, it is precisely because they provide storage beyond the reach of the filesystem (and all software on top of it) that they are attractive to malware authors. For instance, malicious code frequently

[1]The process is conceptually similar for UEFI systems.

Scenario 1: Compromising dark regions. A web server in a Windows guest is compromised. After taking control over the server's execution and elevating its privileges, the malicious code downloads more malicious code (a bootkit) and writes it into the MBR and other dark regions. The malicious code restarts the guest VM to activate the bootkit and join a botnet. Assuming the disk layout of the guest VM is similar to Figure 1(a), the layout after the infection may be the one shown in Figure 3 (malicious code in red).

Scenario 2: Compromising light regions. This time, after compromising a Linux server and elevating its privileges, the malicious code creates an additional entry in `/etc/passwd` with root privileges and no password. Doing so, allows the attacker to enter the system at will at some later time. In addition, we assume that the attacker modifies a user's private key.

Figure 2: Attacks compromising different regions

uses them to save itself and survive across system reboots. In a healthy systems, the regions that are 'dark' from a filesystem perspective (the MBR, the gaps, and so on) either contain startup code, or are empty. In a compromised system, however, these regions often contain malicious code or data. Bootkit infections, in particular, store malicious code in the disk regions that contain the MBR, VBR, or bootloader. Such malware seizes control even before the OS kernel executes, permitting it to hook interrupts and subvert later kernel activities. Scenario 1 in Figure 2 describes such an attack, while a possible disk layout of an *TDL4* [29] infected system is shown in Figure 3 (malicious data in red).

The layout of a storage device is determined when a user employs a partitioning tool that makes low-level changes to the disk. This typically happens before the creation of high-level OS data structures such as filesystems. Of course, the disk layout may legitimately change due to repartitioning, just like the content of the bootsectors may change when the user updates the bootstrap code, but these are rare events.

Light regions. Light regions consist of all regular filesystems' blocks. These blocks are either free or used to store file contents, inodes, and/or other filesystem metadata. It is important to note that free blocks are not 'dark' as they are still for the filesystem to manage. Some files are security sensitive and should only be written by authorized users. Well-known examples include a user's SSH keys, a UNIX system's `/etc/passwd` and `/etc/shadow` files, and executable programs.

The frequency of updates to security-sensitives files varies more than the disk layout and boot records, although some files, like SSH keys, are essentially write-once and never change. Others, like `/etc/passwd` do change, but rarely. Executables may change whenever the user applies an update. In all of these cases, users may want to log the change events. However, since malicious code may be controlling the virtualized environment, it is not safe to do so inside

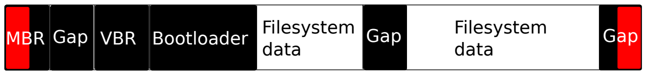

Figure 3: Disk layout of a kvm guest after infection

the environment itself. Scenario 2 of Figure 2 describes an attack that compromises light regions.

3.2 Detecting intrusions via disk accesses

What the example scenarios of Figure 2 have in common is that they deal with (relatively) rare events that are highly relevant from a security perspective. The modification of the startup code or of the layout of the disk indicates either a reinstallation or major upgrade of the guest VM, or a bootkit infection that modifies the startup code to gain control early during system startup. Likewise, an update of certain sensitive files is something that a user might want to hear about also. By monitoring changes to virtual storage devices, we can detect such events and alert the user.

Monitoring disk activity at a low level should not be done from within the guest VM as the malware may already be in control by the time the operating system gets to run. All checks that rely on OS-level services are therefore powerless to detect such attacks. Antivirus (AV) solutions running inside the guest VM are a case in point. They rely on the operating system to scan the storage devices, as well as to provide the semantic knowledge to distinguish malicious activities from benign ones. None of this can be relied upon in case of an intrusion.

Instead, we opt for monitoring at the lowest possible level—the emulated hardware beyond the reach of the guest VM. This has the benefit that malware cannot hide its malicious activity unless an attack manages to escape the virtual environment (which we consider out of scope for this paper). The downside of low-level monitoring is that only transfers of chunks of data are visible, lacking all forms of higher level semantics (like files). This forces us to model virtual storage devices as black boxes which interface with the rest of the guest VM via low-level I/O transfers of chunks of data with no particular semantics. Despite the little or no semantics available on the transferred data, this interface can still be policed to detect even sophisticated intrusions in dark and light regions. In the next section, we will revisit the two examples presented earlier to further substantiate this claim.

4. OVERVIEW

In this section, we present a high-level overview of Slick by outlining the steps a user needs to configure and deploy our storage-level intrusion detection system. We exemplify the workflow with the examples considered in the previous section. For both examples, we assume the disk layout presented in Figure 1(a).

In Slick, an *intrusion* is an I/O (read or write) operation to any region of a monitored virtual storage that violates a predetermined policy. A policy consists of a rule set defined by the user and is enforced throughout the execution of the guest VMs. The rules resemble simple firewall rules that users already employ on commodity platforms. We describe

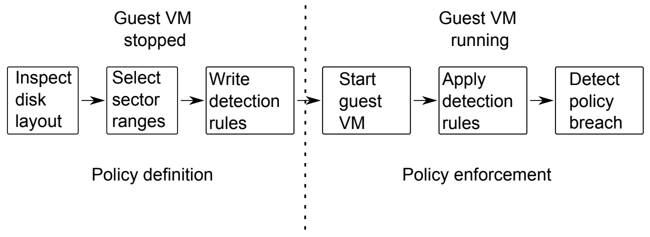

Figure 4: End-to-end user workflow in Slick

the detection rules syntax in more detail in Section 6.2.

Figure 4 depicts the general steps necessary to configure and run SLICK. The process consists of two phases: *policy definition* and *policy enforcement*.

Policy definition. During the *policy definition* phase, the guest VM is stopped. The user *inspects the disk layout*, *selects the sector ranges of interest*, and *writes the detection rules* to configure SLICK. For ease of use (and to avoid human errors), the user only provides file names. All the steps mentioned above are automated by SLICK.

In Scenario 1, the guest VM runs a Windows 7 installation with the disk layout from Figure 1(a):

```
Device    Boot  Start      End  Sectors  Size Id Type
disk.img1 *      2048   718847   716800  350M  7 NTFS
disk.img2      718848 41940991 41222144 19.7G  7 NTFS
```

The virtual server user wants to configure SLICK to detect a potential bootkit installation inside the guest VM. For this purpose, the user defines the following rule set:

```
w 0-0 disk.img
w 1-2047 disk.img
w 2048-2048 disk.img
w 2049-2065 disk.img
```

Applying it, SLICK monitors the sectors involved in the boot process: the MBR (sector 0), the gap used for alignment (sectors 1-2047), the VBR (sector 2048, i.e. the first sector of the bootable partition) and the bootloader (the first 16 sectors of the bootable partition after the VBR).

In Scenario 2, the guest VM runs a Linux installation. The virtual server user wants to configure SLICK to detect modifications to a particular file in the guest VM. To select the relevant sector range, the user must inspect the filesystem data from the guest VM. This operation may require an explicit agreement with the guest users for privacy reasons. Note, however, that the user does not need to access file data, but only the location (sectors) of the file on persistent storage. For example, to monitor a guest's **`/etc/passwd`** file , the user can simply monitor its inode for modifications. Not all changes result in new accounts, but SLICK can easily distinguish between, say, new values for the "last access time", and a modification or addition of a datablock. To monitor the SSH private key of some guest user `jdoe` the user specifies the filename which SLICK automatically matches to the corresponding physical sectors using the SleuthKit tools.

The disk layout is the following (each row is 1 partition):

```
Device    Boot  Start      End  Sectors  Size Id Type
disk.img1 *      2048 16601087 16599040  7.9G 83 EXT4
disk.img2    16601088 16775167   174080   85M 82 swap
```

and SLICK is configured with a simple rule which corresponds to the file **`/home/jdoe/.ssh/id_rsa`**:

```
w 292856-292863 disk.img
```

Policy enforcement. During the *policy enforcement* phase, the guest VM is running, while SLICK enforces the policies defined previously. For every run-time I/O operation SLICK checks against the policy rules and for every match an alert is generated.

Let us now consider the examples introduced earlier. For Scenario 1, the report file is structured as follows:

```
DR: 4 -W 2049 2065 disk.img
DR: 3 -W 2048 2048 disk.img
DR: 2 -W 1 2047 disk.img
DR: 1 -W 0 0 disk.img

W 1 0 0
W 4 2051 2054
```

In the excerpt above, the report file starts with a list of the relevant disk regions corresponding to the policy rules defined by the user. Each region has a given ID, I/O flags (`W` to specify a write operation, in our example), a physical sector range, and the owning storage device. The subsequent lines refer to alerts generated by SLICK. Two intrusions are exemplified above (both for region 1), with policy-violating write operations to sector 0 and sectors 2051-2054 (respectively).

For Scenario 2, the report file exemplifies a single alert for a policy violating write operation to sector 292856— thus illegally modifying the private SSH key:

```
DR: 1 -W 292856 292863 disk.img

W 1 292856 292856
```

5. A SLICK DESIGN

Figure 5 presents the high-level architecture of SLICK. Initially, the virtualization manager creates a guest. The host schedules the guest as a regular process, while the guest makes use of the services offered by the hypervisor.

As in most virtualization solutions, the hypervisor together with a privileged user space process are responsible for emulating a full virtual machine, including all the emulated devices. For isolation, the guest and its controling process have separate address spaces, but the user process has full control over the guest's address space. A single thread of execution of this userspace process runs the guest code in guest mode, while all other threads (also known as worker threads) are responsible for the emulation of devices. In addition, the privileged user process maps regions of memory of the guest address space into its own address space for efficient communication between the guest sandbox and the emulated devices.

Without SLICK, the execution flow for I/O operations proceeds as follows. ① The execution thread executes guest

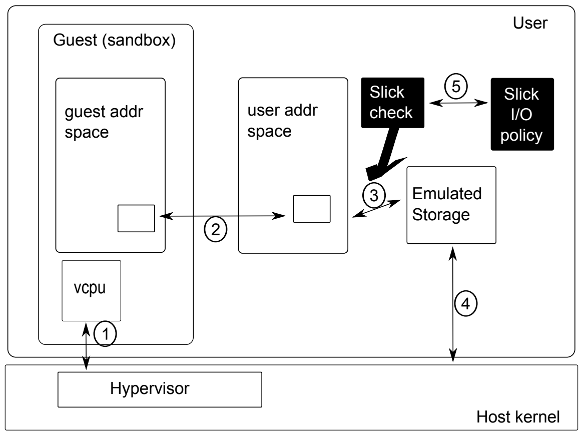

Figure 5: Slick's high-level architecture

code. ② When the guest *vcpu* (virtual CPU) that executes code wants to perform I/O with the disk, it writes to the mapped buffer that it shares with the privileged user process. ③ It then notifies the worker thread responsible for emulation of the device that there is I/O pending by performing an exit from guest mode. Control then passes to the worker thread which executes the request. ④ The worker thread completes the I/O request by calling the *read* or *write* system call using the file descriptor that corresponds to the destination file. Upon completion of the I/O request the user process resumes the execution of the guest(1) which executes more guest code until another external event needs handling. Upon completion of the I/O request the user process resumes the execution of the guest at step ①, to execute more guest code until another external event needs handling.

To police the I/O requests, Slick interposes its checks at this step (as indicated by ③). The reason is twofold. First, the code here runs isolated from the possibly compromised guest. Second, this is the earliest point at which we know all the necessary information about the I/O to check against the detection rules. For example, prior to this step the existence of I/O requests is known in the guest sandbox by the execution thread, but not in sufficient detail to fully describe the I/O request event. Specifically, while we know the direction and the location (from the point of view of the guest), Slick and the IO worker thread also needs to know which the destination file in the host that contains the emulated device data. This information is available only in the userspace process. Since the I/O thread needs to have all information about the source buffer and destination of the I/O, we perform our checks before the worker thread performs the I/O. For Slick, the intention to do I/O to a disk region is enough to detect an intrusion attempt and there is no need to wait for the successful completion of the I/O operation.

6. A SLICK IMPLEMENTATION

In this section we detail our Slick prototype implemented on top of kvm/QEMU [6] version 2.2.0.

6.1 Kvm/QEMU modifications

QEMU and kvm closely follow the general I/O virtualization

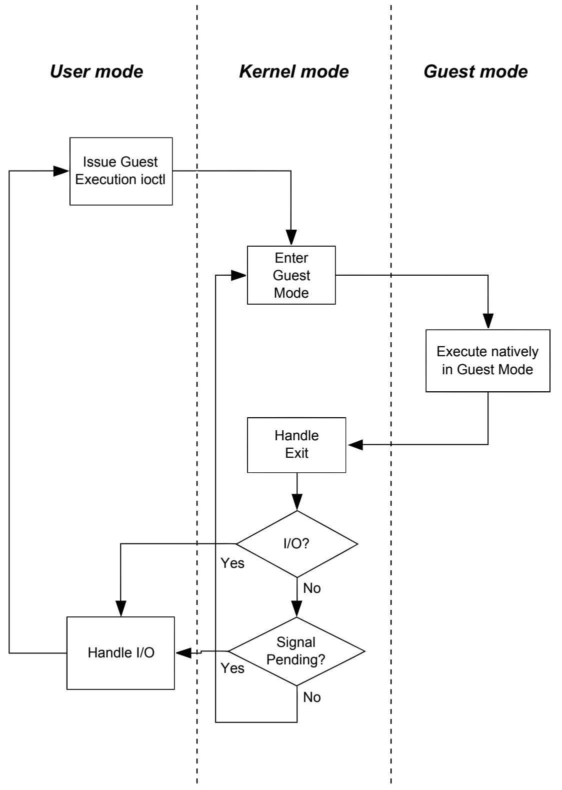

Figure 6: kvm/QEMU guest execution loop [15]

model discussed in Section 5, with all the I/O operations issued by the guest handled by a dedicated user-space I/O thread on the host. Figure 6 depicts the guest execution loop enforced by kvm/QEMU. To issue an I/O operation, a guest kernel thread writes the necessary I/O arguments to a shared buffer and traps into the hypervisor to notify the dedicated I/O thread running in userland.

To intercept I/O thread operations for all the supported storage devices, Slick implements its interposition mechanism in kvm/QEMU's block driver. In detail, Slick places hooks in the existing `bdrv_open_common` and `bdrv_close` functions to perform initialization and cleanup operations when a device is connected and disconnected (respectively). Further, Slick places its detection hooks in all relevant synchronous (`bdrv_rw_co`) and asynchronous (`bdrv_aio_readv`, `bdrv_aio_writev`, and `bdrv_aio_write_zeroes`) I/O functions. No other modification to kvm/QEMU is necessary for Slick to correctly implement its intrusion detection capabilities. Furthermore, the previously documented modifications are fairly small and isolated, ensuring maintainability and encouraging adoption. Slick's policy enforcement core is implemented in C in only 34 lines of code (LOC), with an additional 1966 lines of code to support helpers to ease policy management, disk layout management, memory management, parsing, and reporting.

6.2 Detection rules

Slick's detection rules follow a simple syntax which resembles, e.g., the one in use by common packet filters:

```
<direction> <start sector>-<end sector> <device name>
```

Each rule defines an illegal I/O operation which SLICK should immediately report upon detection. A rule consists of the *direction* of the I/O operation (i.e., `r` or `R` for read, `w` or `W` for write, and `a` or `A` for all the I/O operations) a target *sector range* (i.e., start and end physical sectors in the valid range for the virtual device file), and a *device name* (e.g., the path to the virtual device file on the filesystem). SLICK's detection rules can, in principle, support an arbitrary number of virtual storage devices.

7. EVALUATION

We evaluated our prototype on a Debian Linux 8.0 system running a modified version of kvm/QEMU 2.2.0 (official Debian repositories) with SLICK support. We allowed each guest VM to allocate 2 GB of RAM and use a disk in *raw* format. We experimented with two different guest OS, Debian Linux 7.0 and Windows 7, both 32 bit—with their disk layouts reflecting the ones in Figure1(b) and (a), respectively. We repeated all experiments 5 times and reported the median.

Our evaluation answers 3 key questions: (i) *Performance*: Does SLICK yield low overhead in production? (ii) *Scalability*: Does SLICK scale efficiently with the number of detection policies? (iii) *Effectiveness*: How effective is SLICK in detecting intrusions for well-known malicious attacks?

7.1 Performance

To assess SLICK's performance overhead, we first extensively evaluated our prototype against common server workloads used in production. Our preliminary experiments reported no measurable overhead. This is expected, given that SLICK adds relatively inexpensive checks (range checking) to relatively expensive operations (I/O operations). To highlight the performance overhead, we used PostMark [14] (version 1.51), a I/O-intensive microbenchmark to stress-test the I/O subsystem while simulating email and web services workloads. We configured PostMark to generate 50000 files with 1000-10000 bytes of data and issue 50000-400000 transactions. We cleared all the caches on the host and disallowed the guest VM to use the host's page cache—using kvm's *cache=none* option.

To establish a throughput baseline (transactions per second), we first ran the PostMark benchmark inside a guest VM with SLICK disabled. Next, we enabled SLICK with a policy to track all non filesystem disk regions that resulted in the following detection rules:

```
w 0-0 disk.img
w 1-2047 disk.img
w 16775168-16777216 disk.img
```

Using the SLICK configuration above, we repeated the same test, measured the resulting throughput, and compared our performance results against baseline. Table 1 reports the resulting throughput degradation induced by SLICK in our experiments. Our results show that the performance overhead introduced by SLICK is low even for I/O-intensive microbenchmarks (4.32% in the worst case), which suggests that SLICK can effectively be deployed in production with a negligible performance impact across several possible real-world workloads.

Transactions	Degradation (%)
50000	1.92
100000	3.40
200000	4.32
400000	3.95

Table 1: Slick-induced throughput degradation

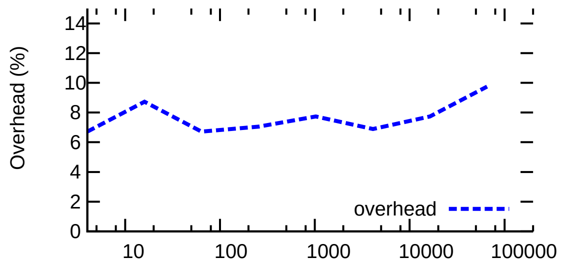

Figure 7: Slick overhead vs. number of rules

7.2 Scalability

To assess SLICK's scalability properties, we first evaluated the impact of the number of configured rules on run-time performance. For this purpose, we ran PostMark multiple times (using the same configuration as before) while varying the number of detection rules configured in SLICK. We used detection rules of the form '`a n-n disk.img`', allowing both read and write operations to a single sector. Figure 7 depicts the resulting throughput degradation for an increasing number of policies. SLICK performance is relatively stable for an increasing number of policies (in the range of 6-9%), yielding a more noticeable increase only for very large rule set sizes. Since we expect a small number of rules in practice (e.g., less than 10 rules typically in use to monitor dark regions with the most common disk layouts, according to Figure 1), we can easily conclude that SLICK scales efficiently to a large number of policies for the practical cases of interest. For Windows 7, a policy covering all exe and dll files contains 9974 rules. At runtime SLICK merges rules which have adjacent locations and the policy has less than 2000 rules. Next, we evaluated SLICK scalability across VMs. We used the configuration from subsection 7.1 with 50000 transactions and ran 2,4 and 8 VMs in parallel on the same host with and without SLICK enabled. Our results show a degradation of 1.91% for 2 VMs, 1.07% for 4 VMs and 2.5% for 8 VMs.

7.3 Effectiveness

To assess the effectiveness of SLICK's intrusion detection strategy, we evaluated our prototype system against a number of well-known malware infections. For our evaluation, we gathered 2400 Windows-based malware samples which emerged between 2010 and 2014. Each sample incorporates a different malware family (*TDL4, Sinowal, Carberp, Pihar, and Trup*) and installs a bootkit to achieve persistence and stealthiness on the system. To detect bootkit-like behavior, we configured SLICK with the following policy:

Malware family	Short description	Writes to dark regions (id)	Detected by policy
Trup	clickjacking trojan	DR1, DR2	Yes
TDL4	considered the most advanced bootkit	DR1, DR5	Yes
Pihar	financial data stealer, related to TDL4	DR1, DR5	Yes
Carberp	banking trojan, source leaked in 2014	DR2, DR4	Yes
Sinowal	partially taken over by researchers in 2009 [30]	DR1, DR5	Yes

Table 2: utilized malware families and example detection results out of the 2400 detected samples

```
w 0-0 disk.img # dark region 1 (DR1)
w 1-2047 disk.img # dark region 2 (DR2)
w 2048-2048 disk.img # dark region 3 (DR3)
w 2049-2065 disk.img # dark region 4 (DR4)
w 41940991-41943040 disk.img # dark region 5 (DR5)
```

To detect modern bootkits modifying code residing within the MBR, VBR, or bootloader, the policy above instructs SLICK to detect all the hard disk changes outside the filesystem regions. *DR1* represents the MBR, *DR2* the space between the MBR and VBR, *DR3* the VBR, *DR4* the bootloader, and *DR5* the free space beyond the last partition. While SLICK was able to successfully detect all our malware samples, for sake of brevity, we only report statistics for the first detection result for each malware family here. Table 2 presents our results, with the *Writes to dark regions (id)* column describing the particular dark regions modified by the malware sample, e.g. *DR2* indicates a modification to *dark region 2*, located between disk sectors 1 and 2047 (the space between the MBR and VBR).

Table 2 provides concrete insights into the effectiveness of our policies. For example, the *Sinowal* malware sample modifies dark region 1 and 5 (*DR1, DR5*)—MBR—and unoccupied space beyond the last partition. Since the policy allows SLICK to detect arbitrary modifications to dark regions, SLICK can accurately detect this and many other malware samples that exhibit similar bootkit-like behavior.

8. RELATED WORK

Intrusion detection systems (IDS) span a range of domains from network based (NIDS) [1, 19, 27] to host based (HIDS) [2, 23]. HIDS typically use introspection techniques to detect and better describe intrusions. Many HIDS that use heavyweight introspection are used as honeypots or malware analysis platforms [5, 9, 22, 32].

A common technique used by HIDS is system call monitoring of the guest environment [18, 23]. It uses the sequences of system calls of previous application executions to detect anomalies when new executions diverge from the known sequences. SLICK handles every I/O operation individually without considering sequences of operations or any other dependencies between them.

Storage based IDS [2, 20, 21] examine the I/O requests for suspicious behavior. Tripwire [2] uses cryptographic hashes to protect filesystem objects which it scans at different moments in time. In [20, 21] the checks are made inside an NFS server which monitors the requests made to the filesystem. These systems are located at the filesystem level and do not consider storage regions outside filesystems thus possibly missing bootkit infections altogether. SLICK on the other hand checks all areas of the disk defined in the policies including non filesystem areas.

Banikazemi [4] presents an storage based IDS implemented in a storage area network (SAN) controller. Similar to SLICK it checks for intrusions at block level rather than at filesystem level and requires a mapping between filesystem objects and occupied blocks. Other systems aim to secure the integrity of major OS components or executed applications via hypervisor technology. Systems like SecVisor [28], [13], [31], [11] exploit the separation supplied by a trusted virtual machine to protect the guest operating system.

9. DISCUSSION

Guest escapes. SLICK relies on the fact that an attack may compromise the guest but cannot escape the virtual environment. Guest escapes were demonstrated in [8, 10, 16]. The attacks assume a more powerful threat model where the attacker is fully aware of the underlying hypervisor and its vulnerabilities. Cloudburst [16] exploits bugs in the VMWare video device emulation that leak host memory to the guest and permit arbitrary host memory write from the guest. Virtunoid [8] exploits a bug in the implementation of KVM PCI device hotplugging that allows unplug requests from devices that do not support it (the Intel PIIX4). This leaves behind dangling pointers that can be exploited by use after free. Venom [10] exploits a buffer overflow in the emulated floppy drive controller and achieves code execution in the context of the host hypervisor process. In the above examples the attack surface consists of the memory mappings from guest to host that emulate port I/O and memory mapped I/O. Even though the guest memory is mapped as non executable in the host address space, [8] shows a way to use mprotect to obtain executable pages and inject payloads from guest to host. A strategy to reduce the attack surface of the KVM hypervisor is to move hypervisor functionality to userspace [12].

Detection policies. SLICK allows for different detection policies: log I/O requests, deny policy-violating I/O requests, read decoy data, redirect writes to shadow blocks, etc. Currently SLICK implements the non-intrusive detection policy that only logs policy violations. However, it is easy to add intrusion prevention capabilities, e.g., redirecting write operations to shadow blocks or replacing the write length with 0. For read operations, SLICK can either serve shadow block or random decoy data.

Light region detection. SLICK, due to its non intrusive and semantics-agnostic nature cannot distinguish between benign and malicious accesses. Still, it provides a audit log beyond the reach of malware that is crucial for detection/analysis of malware behavior a posteriori. Furthermore, in PaaS or SaaS environments, many sensitive files never change without the user's explicit permission, effectively removing such false positives. SLICK can detect using its policies backdoor insertions (by modifying configuration files of services, new user creation), infectors of executable files or libraries. SLICK cannot detect, without adding costly introspection, malicious drive-by downloads which drop new executable files that are executed later.

10. CONCLUSION

Most malware persists across reboots by writing code and data on the system's storage devices. The disk blocks used by such malware may reside in regular file-system space, but could also be outside any file system whatsoever. Example of the latter include malware that modifies the MBR, the bootloader, or even the inter-partition gaps. Detecting such malware from inside the OS is typically not possible, because by the time it runs, the malware is already in control. A much better place for such a storage-level intrusion detector is outside the OS altogether, for instance in the hypervisor.

In this paper, we discuss SLICK, an intrusion detection tool that operates outside the guest VMs and allows a user to specify exactly which blocks to monitor for changes. SLICK is versatile enough to detect intrusion inside the file system as well as outside it. Moreover, it has friendly, yet powerful configuration and incures virtually no overhead in practice. SLICK helps users track many kinds of unwanted modification to the file system, but it is especially useful for detecting stealthy bootkits and other malware like TDL4 and Sinowal.

Acknowledgements. The research was sponsored by ERC Project #259108 ("Rosetta"), NWO VICI project #639.023.309 ("Dowser"), and NWO project #628.001.006, ("Re-Cover").

11. REFERENCES

[1] Suricata. http://suricata-ids.org/.

[2] Tripwire. http://sourceforge.net/projects/tripwire/.

[3] Virtualization Statistics. http://blog.eginnovations.com/2012/08/31/interesting-virtualization-statistics.

[4] BANIKAZEMI, M., POFF, D. E., AND ABALI, B. Storage-based intrusion detection for storage area networks (sans). In *MSST* (2005).

[5] BAYER, U., KRUEGEL, C., AND KIRDA, E. Ttanalyze: A tool for analyzing malware. In *EICAR* (2006).

[6] BELLARD, F. QEMU, a fast and portable dynamic translator. In *USENIX ATC* (2005).

[7] BULYGIN, Y., FURTAK, A., AND BAZHANIUK, O. A tale of one software bypass of windows 8 secure boot. In *BlackHat USA* (2013).

[8] ELHAGE, N. Virtunoid: A KVM Guest - Host privilege escalation exploit. In *BlackHat USA* (2011).

[9] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010).

[10] GEFFNER, J. Venom. http://venom.crowdstrike.com/.

[11] GRACE, M., WANG, Z., SRINIVASAN, D., LI, J., JIANG, X., LIANG, Z., AND LIAKH, S. Transparent Protection of Commodity OS Kernels Using Hardware Virtualization. In *SecureComm* (2010).

[12] HONIG, A. KVM Security Improvements. In *KVM Forum* (2014).

[13] JIANG, X., WANG, X., AND XU, D. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS* (2007).

[14] KATCHER, J. Postmark: a new file system benchmark. Tech. Rep. TR-3022, Network Appliance, 1997.

[15] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the Linux Virtual Machine Monitor. In *Linux Symposium* (2007).

[16] KORTCHINSKY, K. CLOUDBURST: A VMware Guest to Host Escape Story. In *BlackHat USA* (2009).

[17] LINUX LVM. http://www.sourceware.org/lvm2/.

[18] MUTZ, D., VALEUR, F., VIGNA, G., AND KRUEGEL, C. Anomalous system call detection. *TISSEC* (2006).

[19] PAXSON, V. Bro: A system for detecting network intruders in real-time. *Comput. Netw.* (1999).

[20] PENNINGTON, A. G., GRIFFIN, J. L., BUCY, J. S., STRUNK, J. D., AND GANGER, G. R. Storage-based intrusion detection. *TISSEC* (2010).

[21] PENNINGTON, A. G., STRUNK, J. D., GRIFFIN, J. L., SOULES, C. A. N., GOODSON, G. R., AND GANGER, G. R. Storage-based intrusion detection: watching storage activity for suspicious behavior. In *USENIX SEC* (2003).

[22] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys* (2006).

[23] PROVOS, N. Improving host security with system call policies. In *USENIX SEC* (2003).

[24] RAGAN, R., AND SALAZAR, O. Cloudbots: Harvesting crypto coins like a botnet farmer. In *BlackHat USA* (2014).

[25] RIGHTSCALE. State of the Cloud Report 2015. http://assets.rightscale.com/uploads/pdfs/RightScale-2014-State-of-the-Cloud-Report.pdf.

[26] RILEY, R., JIANG, X., AND XU, D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *RAID* (2008).

[27] ROESCH, M. Snort - lightweight intrusion detection for networks. In *LISA* (1999).

[28] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP* (2007).

[29] SOUMENKOV, I., AND GOLOVANOV, S. TDL4 -Top Bot. https://securelist.com/analysis/publications/36152/tdl4-top-bot/.

[30] STONE-GROSS, B., COVA, M., CAVALLARO, L., GILBERT, B., SZYDLOWSKI, M., KEMMERER, R., KRUEGEL, C., AND VIGNA, G. Your botnet is my botnet: analysis of a botnet takeover. In *CCS* (2009).

[31] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering kernel rootkits with lightweight hook protection. In *CCS* (2009).

[32] WILLEMS, C., HOLZ, T., AND FREILING, F. Toward automated dynamic malware analysis using cwsandbox. In *IEEE S&P* (2007).

[33] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *CCS* (2012).