Active File Integrity Monitoring Using Paravirtualized Filesystems

Michael Velten[1], Sascha Wessel[1], Frederic Stumpf[1], and Claudia Eckert[2]

[1] Fraunhofer Research Institution for Applied and Integrated Security
Munich, Germany
{michael.velten,sascha.wessel,frederic.stumpf}@aisec.fraunhofer.de
[2] Technische Universität München, Computer Science Department
Munich, Germany
claudia.eckert@in.tum.de

Abstract. Monitoring file integrity and preventing illegal modifications is a crucial part of improving system security. Unfortunately, current research focusing on isolating monitoring components from supervised systems can often still be thwarted by tampering with the hooks placed inside of Virtual Machines (VMs), thus resulting in critical file operations not being noticed. In this paper, we present an approach of relocating a supervised VM's entire filesystem into the isolated realm of the host. This way, we can enforce that all file operations originating from a VM (e.g., read and write operations) must necessarily be routed through the hypervisor, and thus can be tracked and even be prevented. Disabling hooks in the VM then becomes pointless as this would render a VM incapable of accessing or manipulating its own filesystem. This guarantees secure and complete active file integrity monitoring of VMs. The experimental results of our prototype implementation show the feasibility of our approach.

Keywords: File Integrity Protection, Active File Integrity Monitoring, Paravirtualized Filesystem.

1 Introduction

Protecting the integrity of file objects is a fundamental security objective for building trustworthy systems and for counteracting malware threats. A prominent example of achieving file integrity protection is the Host-based Intrusion Detection System (HIDS) Tripwire [1], which detects manipulations to filesystem objects by comparing their hash values to reference hash values in periodic intervals. However, the problem with Tripwire and similar approaches, including Linux Security Modules (LSM) based approaches like SELinux [2], is that critical security components (e.g., the monitoring components) are not encapsulated from the supervised system. This allows malware to attack and disable the monitoring components in order to conceal attack traces or to hide their presence altogether.

R. Bloem and P. Lipp (Eds.): INTRUST 2013, LNCS 8292, pp. 53–69, 2013.

Researchers have proposed architectures utilizing virtualization to encapsulate the critical security components from the supervised system. The supervised system is moved into a separate VM while the monitoring components are isolated and placed outside of the VM [3, 4]. This prevents malware located in the VM from attacking and disabling the external monitoring components. In order to bridge the semantic gap introduced by the virtualization layer, Virtual Machine Introspection (VMI) techniques are being deployed for monitoring the VMs. Security tools such as [5, 6] build upon VMI and similar techniques for supervising guest VMs. However, these tools realize only *passive* monitoring. This means that security-relevant events occurring within a VM will be recognized after they have happened. In particular, passive monitoring is unable to intercept on events and prevent them from happening. To overcome this problem, researchers have proposed *active* monitoring where hooks are placed inside the monitored VMs. These hooks allow to interrupt the control flow within a VM and give control to the hypervisor before a critical event actually happens [7–9]. However, malware can often circumvent active monitoring by tampering with the hooks placed inside the VMs, thus resulting in critical file operations not being noticed on the hypervisor-level.

In this paper, we present our approach of relocating a supervised VM's entire filesystem into the isolated realm of the host. The only way of accessing and manipulating a VM's filesystem is by communicating with a privileged component located in the hypervisor which has exclusive access to the VM's filesystem. Therefore, the hypervisor is guaranteed that all file operations originating from a VM (e.g., read and write operations) are necessarily routed through the hypervisor. This allows us to actively monitor all file I/O operations within a VM in real-time from "outside of the box" and to possibly prevent them from happening. Furthermore, this approach solves the aforementioned problem of having malware disable hooks in the VM as this would render the VM (and as such the malware itself) incapable of accessing or manipulating the VM's filesystem. The communication between guest VMs and the hypervisor is done over the paravirtualized Plan 9 filesystem protocol [10], which has the advantage of efficiently bridging the semantic gap and preserving all relevant file operation information. Finally, we build upon and improve the work done in [11] to securely measure all executed binaries of all VMs and store these measurements in a single, multiplexed Trusted Platform Module (TPM). This allows for attesting the integrity of individual VMs in the course of a remote attestation. Another key feature of our approach is that we enable regular users of VMs to autonomously install and upgrade software packages in a secure and controlled manner without the need of requiring the intervention of the administrator of the physical system.

The rest of this paper is organized as follows. Section 2 states our assumptions and attacker model. Section 3 explains in detail our concept for active monitoring of guest VMs. Section 4 describes our prototype implementation. Section 5 presents our performance evaluation results. Section 6 gives the security analysis. Section 7 discusses related work. Section 8 concludes this paper.

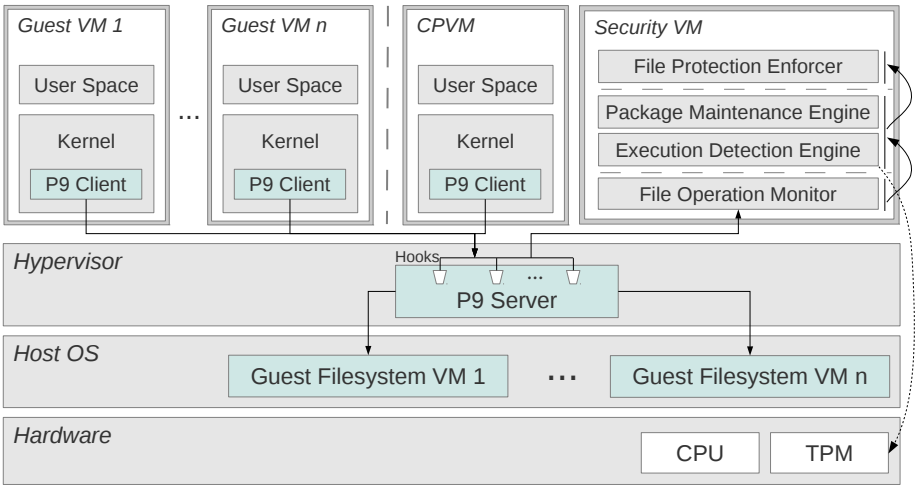


Fig. 1. Paravirtualized monitoring architecture with externalized guest filesystems

2 Assumptions and Attacker Model

We assume a virtualized platform where attackers have full access to their respective guest VMs, but no direct physical hardware access. We consider remote attackers as well as legitimate users of guest VMs that try to compromise the guest VM and gain control of the guest user space and kernel. We focus on preventing malicious file manipulations, which includes temporary as well as persistent file modifications that survive reboots (e.g., the installation of malware). Another objective is to prevent the execution of unknown and malicious executables, respectively. We do not consider runtime attacks, e.g., in-memory modifications, buffer overflow attacks, and code injection.

3 Active Monitoring of Guest VMs

The key aspect of our concept is that we relocate a guest VM's entire filesystem from the guest VM to the isolated realm of the host. We then grant only a privileged component, located in the hypervisor, exclusive access to the guest filesystems. This means that for all guest VMs, the only way of accessing and manipulating their own filesystems is by communicating with this privileged component located in the hypervisor. Therefore, the hypervisor is guaranteed that all file operations originating from a VM (e.g., read and write operations) are necessarily routed through the hypervisor. This allows the hypervisor to actively monitor all file operations of all guest VMs and to possibly prevent them before they actually happen. Furthermore, this makes sure that it is impossible for an attacker to bypass the hypervisor (and as such circumvent the monitoring), even in the event of a completely compromised VM – since otherwise there is no way

of accessing the VM's filesystem. This is an advantage over other approaches (e.g., [7] and [8]) where disabling hooks in the VM still allows for manipulation of filesystem objects.

For our concept, we make use of the Plan 9 (P9) filesystem protocol in order to relocate a guest VM's filesystem to the host. The P9 protocol is designed as a distributed filesystem protocol that may be used over the network and which operates on a file-based granularity. The client-server protocol uses messages that reflect ordinary file operations (for example, messages originating from read or write system calls). In our case, a P9 client resides in each guest VM and cooperates with the P9 server located in the hypervisor. The actual communication between the P9 clients and the P9 server is done over virtio [12], which is the de-facto standard of a paravirtualizing framework for Linux. This allows us to efficiently bridge the semantic gap and to preserve all relevant file operation information.

Our paravirtualized monitoring architecture is shown in Fig. 1. The hypervisor runs one or more guest VMs, which are subject to monitoring. Each guest VM contains a P9 client that translates ordinary file operation requests originating from within the VM to P9 request messages. These messages will be forwarded by the respective P9 client to the P9 server located in the realm of the hypervisor. The P9 server has exclusive access to the filesystems of the guest VMs. The guest filesystems are located on the filesystem of the host. The P9 server processes the P9 requests accordingly, for example, by reading a file (and providing it to the P9 client) or by writing to the filesystem. Note that we prohibit the loading of kernel modules within VMs in order to prevent attacks utilizing filesystem caching (cf. Section 6). In particular, we prevent the loading of kernel modules supporting other filesystems, including virtual and stacked filesystems, as well as modules enabling filesystems in userspace (e.g., FUSE).

There are four components responsible for monitoring and enforcing file integrity of the guests. The monitoring components are encapsulated from the guest VMs (and the hypervisor) in a special security VM (cf. Fig. 1). We place hooks in all relevant parts of the request handlers of the P9 server in order to inform the monitoring components of all relevant file operations. This enables the security VM to monitor all requests originating from a VM's P9 client trying to access its guest filesystem. The components process the P9 requests and decide – based on an access control policy – whether a request will be granted or denied. In particular, the monitoring components are:

File Operation Monitor (FOM). Receives and analyzes all hooked P9 request messages from the P9 server. Relevant requests will be forwarded to EDE and PME (see below). The details are described in Section 3.1.

Execution Detection Engine (EDE). Detects when a guest VM is trying to execute a file based on a heuristic approach which is based on recognizing distinct sequences of P9 requests. Execution of files will be securely recorded by storing a corresponding *SHA*1 hash value in a secure element, in particular, a TPM [13]. The details are described in Section 3.3.

Table 1. Critical Requests of the Plan 9 9P2000.L protocol

P9 Request	Potential Impact
`write`	Writing new files or modifying the content of existing files, e.g., altering configuration files or executables
`rename, renameat`	Moving files, thus effectively deleting them from one location within the filesystem and possibly replacing other files with the content of the renamed file
`remove, unlinkat`	Removing files or directories, e.g., changing the behavior of programs by deleting their configuration files or hiding traces by deleting log files
`lcreate, mkdir`	Creating new files or directories; may be misused to truncate existing files
`link, symlink`	Creating a hardlink or symbolic link, e.g., creating a link in a directory like `/bin` to a malicious executable in `/tmp` (where the creation of arbitrary files may be allowed)

Package Maintenance Engine (PME). Detects when a guest VM is trying to install, remove, upgrade, or downgrade software packages, and handles it by utilizing a special VM, called the Complementary Privileged Virtual Machine (CPVM). The details are described in Section 3.4.

File Protection Enforcer (FPE). Decides whether a P9 request will finally be granted or denied. The decision is based on whitelist policy rules. The details are described in Section 3.2.

3.1 Monitoring and Analyzing File Operation Requests

The File Operation Monitor (FOM) is responsible for analyzing P9 request messages forwarded by the P9 server. In particular, FOM scans for all *critical requests* of the utilized 9P2000.L[1] protocol [14]. A request is considered critical if it has the potential to impact the integrity of the guest's filesystem. Table 1 lists all critical P9 requests that are handled by FOM along with a description of their potential impacts.

Note that Table 1 does not list the P9 `read` request since it cannot be used to affect a file's integrity. However, `read` requests still play an important part in our concept as they occur as a distinct sequence of P9 requests whenever a file in the guest VM is going to be executed. Since the P9 filesystem protocol does not incorporate a dedicated `execute` request itself, we take advantage of this sequence of `read` signature requests in order to come up with a heuristic to detect the execution of files. The details are described in Section 3.3.

Shadow Copy Write. The P9 `write` request requires further consideration. A special case of the `write` request is that it may exceed the message size of

[1] 9P2000.L includes the core 9P2000 requests as a subset.

the P9 client or the P9 server implementation (or both). The reason is that the entire (to be written) payload data has to be sent from the P9 client to the P9 server. In such cases, the P9 client splits up a `write` request $w[f, d]$ (containing the payload data d for a file f) into several sub-requests $w_1[f, d_1], \ldots, w_n[f, d_n]$ [14]. This poses a problem for monitoring `write` requests because FPE may not be able to decide upon the partial information of a sub-request $w_i[f, d_i]$ (in particular, the first sub-request $w_1[f, d_1]$) on whether the overall request $w[f, d]$ should be granted or not. In particular, if FPE only allows a file f to be written if its future content (i.e., the content of f after applying the `write` operation $w[f, d]$) matches a certain hash value (cf. Section 3.2), knowledge of the entire future content of f is required in order to be able to calculate the hash value of f. Note that in this regard, it is not sufficient to only consider the payload data d. The reason is that a `write` request may only partially write a file f. In this case, the payload data d differs from the content of the resulting file f.

We solve this problem by introducing a technique called *shadowing*, which works in three phases:

1. FOM detects a `write` sub-request $w_1[f, d_1]$ by inspecting the request's header data. If f already exists on the guest's filesystem, FOM creates a *shadow copy* f' with the same content as f. If f does not exist, FOM creates an empty file f'. The shadow copy f' is located outside of the guest's filesystem and only accessible by FOM. Depending on the size of f, and possibly other factors (e.g., hardware and performance constraints), the shadow copy may be kept entirely in RAM.
2. FOM applies the sub-request $w_1[f, d_1]$ along with all other corresponding sub-requests $w_2[f, d_2], \ldots, w_n[f, d_n]$ exclusively to the shadow copy f'. When all sub-requests $w_1[f, d_1], \ldots, w_n[f, d_n]$ have been processed (which is detected by a terminal `clunk` or `fsync` operation [10]), FOM signals to FPE that there is a new `write` request $w[f, d]$ and passes a pointer to f'.
3. FPE is then able to calculate the hash value of f', which resembles the potential future content of f, and to finally decide whether the overall `write` request $w[f, d]$ should be granted or denied. If it is granted, the P9 server eventually gets signaled to allow and process all sub-requests $w_1[f, d_1], \ldots, w_n[f, d_n]$. Finally, the shadow copy f' gets discarded.

3.2 Enforcing File Protection

The File Protection Enforcer (FPE) is responsible for deciding whether a P9 request will be granted or denied. The decision making is based on Access Control List (ACLs) that define which filesystem operations are allowed within guests and which ones are prohibited. An ACL consists of arbitrarily many Access Control Entries (ACEs) which determine for a given file f whether certain operations on f are allowed or denied. There exists one ACL for each VM. The ACL implements a whitelist approach that prohibits all filesystem operations within a VM unless an operation is explicitly granted by an ACE.

Table 2. Critical requests mapped to policy checks using only predicates

P9 Request	Predicate Policy Check
`write(f,d)`	$f' \leftarrow w[f,d] : W(f) \wedge H(f')$
`rename(`f_1`,`f_2`), renameat(`f_1`,`f_2`)`	$W(f_2) \wedge D(f_1) \wedge H(f_1)$
`remove(f), unlinkat(f)`	$D(f)$
`lcreate(f)`	$W(f)$
`link(`f_1`,`f_2`), symlink(`f_1`,`f_2`)`	$W(f_2) \wedge H(f_1)$
`exec(f)` (*)	$E(f) \wedge H(f)$

(*) virtual request

Policy Predicates and P9 Request Mapping. We define a minimal set of four predicates that may be used to construct an ACE. All predicates evaluate to either true or false. The predicates are:

$W(f)$: (partial) writing of file f allowed?
$D(f)$: deletion of file f allowed?
$E(f)$: execution of file f allowed?
$H(f)$: hash sum of the content of file f matches a reference hash value?

The objective of exclusively using this minimal set of predicates in the ACEs, is to abstract from the actual P9 requests and to come up with simpler, more generic ACEs. This has the advantage that one does not have to create ACEs for each specific P9 request. Instead, it is only required to define for a file f whether writing $W(f)$, deletion $D(f)$, and execution $E(f)$ is allowed or denied (the latter of which is the default) – possibly in conjunction with reference hash values that have to be matched ($H(f)$). In particular, a file may be associated with a list of one or more *reference hash values* $\langle h_1, \ldots, h_n \rangle$. The predicate $H(f)$ evaluates to true iff the content of f matches one of the hash values h_i or if the list of reference hash values contains the wildcard character "$*$". Otherwise, $H(f)$ always evaluates to false.

For example, an ACE for a file f may define that writing of f is allowed (W predicate) if the resulting content matches one of several reference hash values (H predicate). Such an ACE may then evaluate to true not only for P9 `write` requests but also for `rename`, `renameat`, `lcreate`, `link`, and `symlink` requests, respectively, as will be explained in the following.

For the actual policy enforcement, the FPE internally maps all critical P9 requests (cf. Table 1) to corresponding policy checks using only these predicates. The mapping is shown in Table 2 (for clarity, we only illustrate the policy checks for files and omit the checks for directories). If the overall expression of such a policy check evaluates to true, the respective P9 request will be granted by FPE. Otherwise, it will be denied. Note that a `write` request $w[f,d]$ (which might be a partial write) will first be applied to a temporary file f' (denoted by $f' \leftarrow w[f,d]$ in Table 2). This is similar to shadowing as described in Section 3.1. If the content of the resulting file f' matches a valid reference hash value, $H(f')$ evaluates to *true*. Also note that `exec` is not an actual P9 request but a *virtual request* which is propagated by EDE. This is explained in Section 3.3.

Package Policy Rules. We also define predicates to determine which software package maintenance operations may be autonomously issued by legitimate guest VM users (cf. Section 3.4). The predicates are:

$P_i(p)$: installing, upgrading, or downgrading package p allowed?
$P_r(p)$: removing package p allowed?
$P_h(p)$: hash sum of the package p matches a reference hash value?

Whenever PME detects an installation, upgrading, or downgrading attempt of a package p (cf. Section 3.4), respectively, it is propagated to FPE which, in turn, will check whether the predicate $P_i(p)$ evaluates to true. Furthermore, the predicate $P_h(p)$ may be used – analogously to $H(f)$ as described above – to restrict the installation, upgrading, and downgrading of a package p to packages that match a reference hash value. This allows to selectively permit only certain packages (and package versions) while prohibiting others, e.g., older versions with known vulnerabilities. For removing attempts of p, FPE will check whether the predicate $P_r(p)$ evaluates to true.

3.3 Detecting Program Execution

Detecting and possibly preventing the execution of files within VMs is an important part of our concept. Unfortunately, having EDE detect executed programs from outside of the guest VMs is not straight forward due to the fact that P9 does not distinguish between reading a file and executing a file. Instead, in both cases a read request is sent by the P9 client and only the VM decides afterwards whether the read file will be executed. Note that we cannot just extend the P9 clients (and server) such that they distinguish between read and execute requests (e.g., an executable-bit). The reason is that this information would not be trustworthy since an attacker may tamper with it (e.g., setting the executable-bit from 1 to 0) once the VM is compromised. Therefore, we incorporate EDE which is able to detect the execution of a file within a VM by utilizing a heuristic approach. EDE is protected from the aforementioned attacks since it is located in the security VM (cf. Fig. 1) and monitors the VMs from "outside of the box", without relying on auxiliary (untrustworthy) information sent from the VM.

Whenever a program is going to be executed within a VM, there will be a distinct sequence of preceding Plan 9 requests in a defined chronological order, as described in the following. EDE recognizes this sequence of *signature requests* and deduces which file is intended to be executed. FPE may then grant or deny the execution based on policy rules as described in Section 3.2.

For the execution detection, we consider the Executable and Linking Format (ELF) [15], which is the standard binary format for executables on many Unix-like operating systems, including Linux. The heuristic for detecting the execution of ELF files under Linux, consists of the following signature requests (in their chronological order of occurrence):

1. The `execve` system call first reads in 128 bytes to determine the binary type of a file f. Consequently, EDE scans for the corresponding P9 read requests.
2. The ELF loader of the Linux kernel invokes the function `kernel_read`, which reads 224 bytes from f, starting from offset 52.
3. A subsequent invocation of `kernel_read` reads 19 bytes from f, starting from offset 276, which gets treated as the path to an interpreter [15].

The above signature requests are usually followed by multiple read requests that attempt to map the entire file f into memory. Note that EDE is also able to detect the loading of ELF libraries, which generate signature requests similar to that of executed binaries. We consider the detection of executed script files (e.g., shell scripts) out of the scope of this paper. This will be addressed in future research.

Secure Storage of Integrity Measurements. We build upon and improve the work done in [11] to measure all executed binaries of all VMs and store these measurements in a single, multiplexed TPM. This allows for attesting the integrity of individual VMs (*remote attestation*). In [11], each VM runs an adapted version of the Integrity Measurement Architecture (IMA) [16] that monitors the execution of files, calculates integrity measurements, and forwards them to a TPM multiplexing agent located in the hypervisor. We improve this solution by relocating and consolidating the IMA measurement components from the guest VMs to the well encapsulated security VM. This has the advantage that only a single measurement agent is required for monitoring the execution of files of all VMs from "outside of the box" and for storing integrity measurements ($SHA1$ values) in the TPM. Additionally, this prevents attackers from tampering with the monitoring and measuring components, respectively, since they are out of reach of the guest VMs. In our case, measuring all monitored executables and storing them in the TPM is done by EDE.

3.4 Autonomous Software Package Installation and Upgrade

Another key feature of our approach is that it is possible for legitimate users of guest VMs to autonomously install, remove, upgrade, or downgrade software packages without the need of any manual intervention by the administrator of the physical system. However, these *package maintenance operations* are not allowed to be done in an arbitrary manner, but all such operations have to adhere to the policy rules enforced by FPE as described in Section 3.2. Also note that it is not possible for a guest VM user to directly manipulate the package contents as they are write protected. This prevents illegal modifications of the guest VM by attackers – which includes legitimate but maliciously acting VM users. Finally, note that PME may also actively enforce the upgrading of (outdated) packages within VMs.

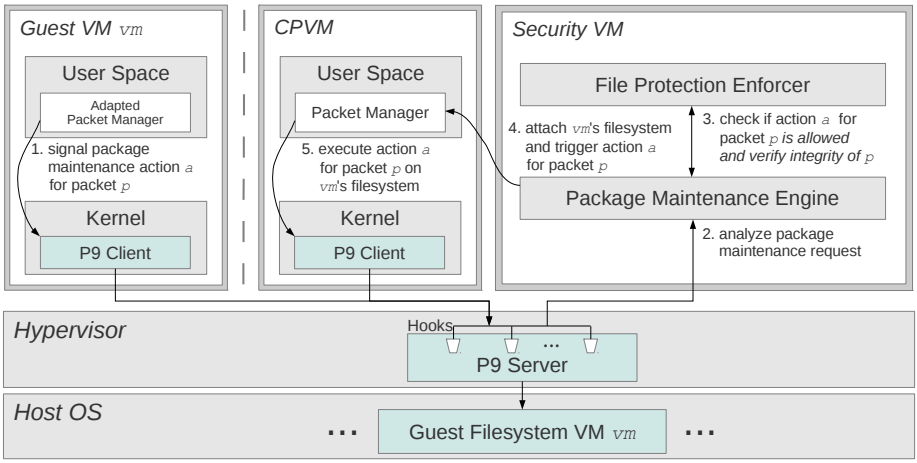


Fig. 2. Installation and upgrading of packages via CPVM

The work flow for installing, removing, upgrading, and downgrading software packages is depicted in Fig. 2 and will be described in the following.

i) Signaling of Package Maintenance Request. First, a legitimate user of the guest VM executes the package manager within the VM with the corresponding maintenance action a (and parameters) for a package p (step 1 of Fig. 2). The request is forwarded by the P9 client to the P9 server. The Package Maintenance Engine (PME) located in the security VM catches and analyzes the package maintenance request (step 2). In this regard, it is important to note that PME considers all information gathered from the guest VM as untrustworthy. This means that even if an attacker compromised the guest VM, it is not possible for him to use the package manager to send fake information in a way that would allow the circumvention of the policy rules or the malfunctioning of any other security-critical component outside of the guest VM.

ii) Checking Package Integrity and Permissions. PME sends a query to FPE in order to determine whether p is a known and valid package on which the requested action a may be applied. Hence, FPE first checks if the action a on package p is allowed for the respective VM by evaluating the P_i and P_r predicates of the corresponding ACE. Following this, FPE verifies the integrity of the package p by evaluating the $P_h(p)$ predicate of the corresponding ACE (cf. Section 3.2). The usage of reference hash values allows to selectively permit only certain packages – and package versions – while prohibiting others (e.g., older versions with known vulnerabilities) that may otherwise be exploited by an attacker to compromise the system. If the hash value is not valid, the maintenance process is aborted and an error is signaled to the package manager of the guest VM.

iii) Executing Package Maintenance Request. The package maintenance process for all guest VMs is executed in a special VM, called the Complementary Privileged Virtual Machine (CPVM). The CPVM runs in parallel to the guest VMs and has exclusive permission to install, remove, upgrade, or downgrade packages of all VMs. A key feature of the CPVM is that it operates (via the P9 protocol) on the same filesystem (located in the host) as the guest VM *vm* that triggered the respective package maintenance request. This is achieved by attaching *vm*'s filesystem (on the fly) to CPVM, for the duration of the package management process. This way, all package management operations done by CPVM are immediately visible to *vm*, and vice versa. This prevents synchronization problems and guarantees that both VMs always operate on the same state of the VM (e.g., information on which packages are installed, package versions, configuration file settings, etc.). Note that the guest VMs only require minimal (non-security critical) user space modifications of the package management tools (cf. Section 4.1) but no kernel modifications.

In the following, we justify the execution of the package maintenance operations within CPVM as opposed to executing them in the guest VM itself. The latter case could be achieved by having FPE properly adjust the policy rules such that the creation, deletion, modification etc. of files belonging to a certain package would be temporarily permitted for a certain VM. However, many modern package managing tools also allow packages (e.g., Debian packages, as used in our prototype implementation in Section 4.1) to contain script files that will be executed before and after a package maintenance operation, respectively. Parsing these script files (which may contain arbitrarily complex commands) and extracting their complete semantics (in order to be able to have FPE temporarily grant the corresponding file operations) is a highly complex task. Possible workarounds include disallowing such scripts or imposing certain constraints on their contents. However, this would prevent taking advantage of real-life packages as shipped with modern Unix-like operating systems. Our CPVM approach solves the aforementioned problems, yet it is fully compatible with full-fledged Unix-like operating systems (e.g., Linux distributions such as Debian).

Note that our approach does not require to suspend or pause a guest VM *vm* while CPVM is executing its software management operations on *vm*'s filesystem but both VMs can run in parallel. This is due to the fact that both VMs communicate with the same P9 server – which deals with the correct synchronization of P9 requests. As such, the functioning of *vm* and CPVM is comparable to two (especially encapsulated) processes operating on the same filesystem within the realm of an ordinary operating system.

4 Implementation

We have implemented a prototype using the Native Linux KVM Tool (KVM) [17], version 3.1.rc7, with enabled KVM full virtualization support. In contrast to QEMU-KVM [18, 19], KVM has the goal to provide a clean, from-scratch,

lightweight KVM host tool with only the minimal amount of legacy device emulation [17]. KVM ships with a P9 file server utilizing the virtio framework [12] for communicating with the P9 clients residing in the guest VMs. The P9 client functionality is provided by the v9fs client of the Linux kernel, which supports both the standard 9P2000 protocol and the extended 9P2000.L protocol, the latter of which we use.

Our host system runs Ubuntu 12.10. Each guest VM runs Debian 6.0 with Linux kernel 3.5.0 and enabled virtio and P9 support. The Linux guest kernel images reside on the host filesystem and will be passed as a parameter to KVM whenever a new VM is started. The security VM and CPVM also run Debian 6.0 with Linux kernel 3.5.0. The attached guest filesystem of CPVM is passed to KVM as a reference to a symbolic link. PME redirects this symbolic link dynamically to other guest filesystems as required by package maintenance requests.

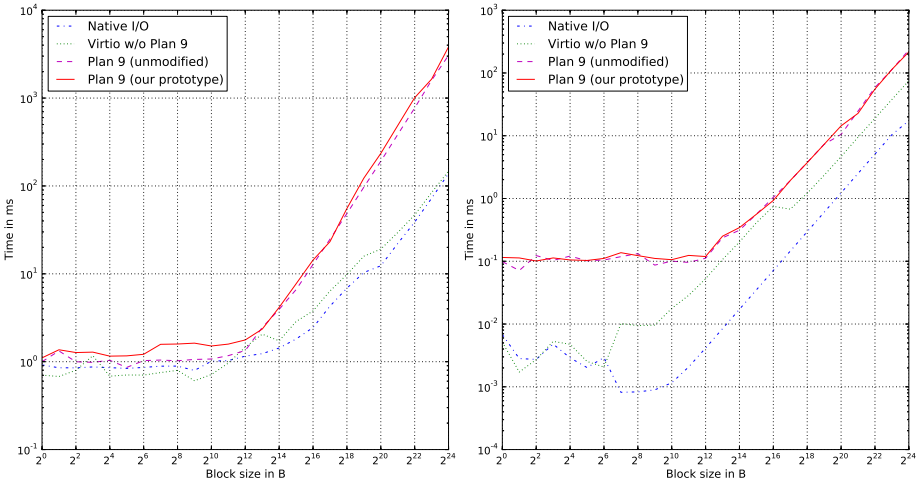
The P9 server hooking functionality is realized by patching all relevant request handlers of the P9 virtio implementation so that FOM gets signaled and forwarded all required information. FOM and EDE are implemented in C. PME and FPE are implemented using a combination of Python scripts and shell scripts. Furthermore, FPE utilizes SQLite3 for efficiently managing our policy rules.

4.1 Installation and Upgrading of Packages via CPVM

As already mentioned, the guest VMs run Debian, which ships with the package management tools *dpkg* and *apt-get*. Since we do not allow guests to directly install, remove, upgrade, or downgrade packages on their own (cf. Section 3.4), we replace the user space tools *dpkg* and *apt-get* with our own versions $dpkg^R$ and $apt\text{-}get^R$, respectively, both of which forward all package maintenance requests to PME via the P9 protocol. To avoid having to modify or extend the P9 protocol, we just take advantage of regular P9 requests (that will be treated specially by PME) in order to pass the information of package management action, parameters, and package name. In particular, we utilize the P9 `mkdir` request (cf. Table 1) because it allows us to transfer all required information. PME parses the request and queries FPE on whether the action for package p is allowed and whether p is a valid package. If the request gets granted by FPE, PME places a file (called a *job*) in a special directory which is only accessible by CPVM. The job contains the respective command that will be executed by the privileged CPVM as soon as CPVM gets scheduled by PME. PME attaches the filesystem of the respective guest VM to CPVM and schedules CPVM. Eventually, CPVM detects the new job and executes it. Upon successful completion of the job, PME grants the P9 `mkdir` request to signal to $dpkg^R$ and $apt\text{-}get^R$, respectively, that the package maintenance request has been successfully executed.

5 Performance Evaluation

We assess the performance of our prototype implementation by measuring its write and read performance, and by comparing the results to three other



(a) write performance (b) read performance

Fig. 3. Comparison of write and read performance of different environments

environments. The testing hardware consists of a PC with an Intel Core i7-2640M 2.8GHz CPU, 4 GB RAM, and an Intel SSDSA2BW160G3L solid-state drive containing an ext4 filesystem with a block size of 4kB.

Fig. 3 shows our testing results of the (a) write performance and (b) read performance benchmarks. We conducted the write and read operations with block sizes from 1B of up to 16MB (2^{24}B) and disabled caching.

The write performance is depicted in Fig. 3a. The time (in ms) to write data is given as a function of the block size (in bytes). All four examined environments – native I/O, virtio block without P9, unmodified P9 (plain P9), and our prototype – perform similarly up to block sizes of approx. 8kB (2^{13}B). For larger block sizes, the P9 environments perform worse than native I/O and virtio block. However, in our usage scenario such larger block sizes are negligible since I/O operations are usually done in block sizes of typical filesystems – which normally lie in the range of 512B to 4kB (which is also the maximum block size for ext4 on most architectures). There is no significant performance difference between plain P9 and our prototype.

As for the read performance (Fig. 3b), the results look as expected: native I/O takes the least time to read blocks, followed by virtio block, followed by the P9 environments – which inherently have the biggest performance overhead. However, analogous to the write performance, there is no significant performance difference between plain P9 and our prototype.

6 Security Analysis

In the following, we discuss attacks that target the persistent and non-persistent manipulation of files as well as the circumvention of the execution detection.

An attacker may try to persistently manipulate files within a VM and prevent the propagation of the changes to the P9 server, thus undermining active monitoring. A possible approach would be to compromise the guest kernel and tamper with the P9 client such that certain (or all) P9 messages will be blocked from being propagated to the P9 server. However, as explained in Section 3, all file operation requests must necessarily be routed through the P9 server because otherwise it is impossible for a guest VM to access the VM's filesystem.

For non-persistent file manipulations, an attacker may cache the filesystem (or parts thereof) locally in RAM and only work on this cached data (e.g., writing files in memory), thus undermining active monitoring. We protect against these kind of attacks by prohibiting the loading of kernel modules. In particular, we prevent the loading of kernel modules supporting other filesystems, including virtual and stacked filesystems, as well as modules enabling filesystems in userspace (e.g., FUSE). Attacking the kernel itself is only possible with runtime attacks (e.g., code injection), which is excluded by our attacker model (cf. Section 2).

For circumventing execution detection by EDE (cf. Section 3.3), an attacker might also employ stacked filesystems. However, we prevent attacks involving stacked filesystems by prohibiting the loading of kernel modules as described above. An attacker may also try to circumvent the execution detection by first mapping an entire file into memory and then executing it from RAM. There exist orthogonal techniques for preventing such attacks (e.g., [20, 21]), which we consider out of the scope of this paper. As mentioned in Section 3.3, we currently do not consider the detection of executed script files (e.g., shell scripts). However, this is not due to a limitation of our architecture but is rather a matter of effort to extend the heuristic in future research.

7 Related Work

Tripwire [1] is a commonly known HIDS, which detects changes to filesystem objects by checking the filesystem in periodic intervals. However, there is no support for real-time checking. Hence, Tripwire cannot *prevent* attacks but just detect them after they have happened. Furthermore, Tripwire is not encapsulated from the monitored system and as such is susceptible to attacks. I^{3}FS [22] tries to improve Tripwire by adding real-time integrity checks. However, since the supervising agent and the relevant databases are located within the realm of the monitored system, I^{3}FS is also vulnerable to attacks.

In [9], Zhao et al. implement active monitoring in a virtualized environment. They try to bridge the semantic gap between disk blocks and logic files with the help of the block tap library *blktap* [6]. However, they still allow the modification of files in security-critical directories (e.g., `/etc`) while only logging these modifications, thus being incapable of *preventing* potential attacks. Our active monitoring approach allows VMs to autonomously upgrade software packages in a controlled manner, thus implicitly enabling the secure and restricted modification of files in security-critical directories.

HIMA [23] provides hypervisor-based active monitoring of critical guest events and guest memory protection. However, their described approach requires

considerable effort for bridging the semantic gap. In contrast, our approach is very efficient in preserving the semantic knowledge of file operation events within VMs on a high-level abstraction by utilizing the Plan 9 protocol.

Lares [7] and Xenprobe [8] place hooks in the guest VMs in order to trace syscalls. However, these hooks can be attacked and disabled from within the VM. Hence, the hypervisor is not able to reliably monitor the VMs. Our approach of relocating the guest VM's filesystem from the realm of the guest VM to the host guarantees that all file operations originating from a VM are necessarily routed through the hypervisor in order to implement reliable monitoring.

8 Conclusion

In this paper, we presented our virtualized architecture that allows for active file integrity monitoring. The key idea of our approach is to relocate a supervised VM's entire filesystem into the isolated realm of the host such that all file operations must necessarily be routed through the hypervisor. This allows for complete active monitoring and the prevention of critical filesystem events. In contrast to existing active monitoring approaches, our technique has the advantage that hooks placed inside the VMs are not prone to manipulation by malware. The reason is that disabling hooks in a VM inevitably renders the VM incapable of accessing or manipulating its own filesystem (provided by the respective hook). Another key feature of our approach is that we enable regular users of VMs to autonomously install and upgrade software packages in a secure and controlled manner, without the need of requiring the intervention of the administrator of the physical system. Finally, we securely measure all executed binaries of all VMs and store these measurements in a single, multiplexed TPM. The experimental results of our prototype implementation show the practicality of our approach.

Acknowledgements. We would like to thank our colleagues Julian Horsch and Steffen Wagner for fruitful discussions and valuable comments. This work was partly supported by the Federal Ministry of Economics and Technology (BMWi) through grant 01MD11012.

References

1. Kim, G.H., Spafford, E.H.: The design and implementation of Tripwire: A file system integrity checker. In: Proceedings of the 2nd ACM Conference on Computer and Communications Security, pp. 18–29. ACM (1994)
2. Smalley, S., Vance, C., Salamon, W.: Implementing SELinux as a Linux security module. NAI Labs Report 1, 43 (2001)
3. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In: Proc. Network and Distributed Systems Security Symposium, pp. 191–206 (2003)

4. Nance, K., Bishop, M., Hay, B.: Virtual machine introspection: Observation or interference? IEEE Security & Privacy 6(5), 32–37 (2008)
5. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: Tracking processes in a virtual machine environment. In: Proceedings of the USENIX Annual Technical Conference, pp. 1–14 (2006)
6. Payne, B.D., de Carbone, M.D.P., Lee, W.: Secure and flexible monitoring of virtual machines. In: Twenty-Third Annual Computer Security Applications Conference, ACSAC 2007, pp. 385–397 (2007)
7. Payne, B.D., Carbone, M., Sharif, M., Lares, W.L.: An architecture for secure active monitoring using virtualization. In: IEEE Symposium on Security and Privacy, SP 2008, pp. 233–247. IEEE (2008)
8. Quynh, N.A., Suzaki, K.: Xenprobes, a lightweight user-space probing framework for xen virtual machine. In: USENIX Annual Technical Conference Proceedings (2007)
9. Zhao, F., Jiang, Y., Xiang, G., Jin, H., Jiang, W.: VRFPS: A Novel Virtual Machine-Based Real-time File Protection System. In: Proceedings of the 2009 Seventh ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2009, Washington, DC, USA, pp. 217–224 (2009)
10. Van Hensbergen, E., Minnich, R.: Grave Robbers from outer space using 9P2000 under Linux. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC 2005, p. 45. USENIX Association, Berkeley (2005)
11. Velten, M., Stumpf, F.: Secure and Privacy-Aware Multiplexing of Hardware-Protected TPM Integrity Measurements among Virtual Machines. In: Kwon, T., Lee, M.-K., Kwon, D. (eds.) ICISC 2012. LNCS, vol. 7839, pp. 324–336. Springer, Heidelberg (2013)
12. Russell, R.: Virtio: towards a de-facto standard for virtual I/O devices. ACM SIGOPS Operating Systems Review 42(5), 95–103 (2008)
13. Trusted Platform Module, Main Specification, Level 2, Version 1.2, Revision 116 (2011), `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`
14. Plan 9 – 9P2000.L Protocol, `https://code.google.com/p/diod/w/list`
15. Tool Interface Standard (TIS) – Executable and Linking Format (ELF) Specification (May 1995), `http://refspecs.linuxbase.org/elf/elf.pdf`
16. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th Conference on USENIX Security Symposium, SSYM 2004, vol. 13. USENIX Association, Berkeley (2004)
17. Native Linux KVM Tool, `https://github.com/penberg/linux-kvm`
18. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the Linux virtual machine monitor. In: OLS 2007: Proceedings of the Linux Symposium, vol. 1, pp. 225–230 (June 2007)
19. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC 2005. USENIX Association, Berkeley (2005)
20. Wessel, S., Stumpf, F.: Page-based Runtime Integrity Protection of User and Kernel Code. In: 5th European Workshop on System Security (2012)

21. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: Proceedings of the 17th Conference on Security Symposium, SS 2008, pp. 243–258. USENIX Association, Berkeley (2008)
22. Patil, S., Kashyap, A., Sivathanu, G., Zadok, E.: I3FS: An in-kernel integrity checker and intrusion detection file system. In: Proceedings of the 18th Annual Large Installation System Administration Conference, LISA 2004 (2004)
23. Azab, A.M., Ning, P., Sezer, E.C., Zhang, X.: HIMA: A Hypervisor-Based Integrity Measurement Agent. In: ACSAC, pp. 461–470. IEEE Computer Society (2009)