CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:1092–1109
Published online 11 April 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3273

OB-IMA: out-of-the-box integrity measurement approach for guest virtual machines

Bin Xing*,†, Zhen Han, Xiaolin Chang and Jiqiang Liu

School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China

SUMMARY

Infrastructure as a Service cloud provides elasticity and scalable virtual machines (VMs) as computing service to multiple tenants, but the tenants lose the full control of their data. Measuring the integrity of critical files of the VMs and providing the integrity attestation to the tenants on the basis of TCG trusted computing techniques is an effective way to alleviate their anxiety. This paper considers how to measure the integrity of the processes run in guest VMs and files opened in guest VMs. We propose an out-of-the-box integrity measurement approach to measure the integrity of critical files through system call (syscall) interception without any modification of the guest VMs. Out-of-the-box integrity measurement approach can not only measure the integrity of all files that have been considered by existing approaches but also measure the integrity of the system configuration files, program loaders, and script interpreters, which affect the system behaviors and integrity. The ability of supporting both system and manual measurement policies makes our approach flexible. We implement this approach in Xen hypervisor with little modification of the existing syscall interception method, and this approach can be ported to other virtualization platform easily. Copyright © 2014 John Wiley & Sons, Ltd.

Received 02 February 2013; Revised 31 January 2014; Accepted 07 March 2014

KEY WORDS: virtual machine; cloud computing; trusted computing; integrity; Xen

1. INTRODUCTION

Cloud computing provides on-demand and dynamic scalable services in various modes. It has been regarded as a new evolutionary phase of computing. From the Cloud Service Provider's (CSP's) point of view, many types of computing resources can be provided as a service to the users. Different from Software as a Service and Platform as a Service, Infrastructure as a Service (IaaS) provides elasticity and measurable computing, storage, and I/O resources to tenants, and tenants only need to pay for the resource actually used, which makes IaaS to be considered as a brand-new, neoteric, and remarkable service in cloud computing. Although the features of energy saving and money saving bring great opportunities to IaaS, security problems are still the hurdle for its wide applications as before. How to design a mechanism to satisfy the tenants' security requirements is still an open issue. In general, measuring the tenants' files and providing the integrity attestation on the basis of trusted computing is an effective way to make tenants trust the CSP's service.

Trusted computing technology, which was proposed by Trusted Computing Group (TCG) [1], uses the Trusted Platform Module (TPM) [2] as the root of trust for measurement and storage to guarantee the integrity of computing platform. TPM changes the architecture of computer by putting the Core Root of Trusted Measurement (CRTM) in front of BIOS when booting a system. During the boot process, the former object will measure the latter object's integrity and extend the digest

*Correspondence to: Bin Xing, School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China.
†E-mail: xingbin@bjtu.edu.cn

into the one-way registers Platform Configuration Registers (PCRs) in TPM chip. Besides, the digest will also be stored in the Stored Measurement List (SML) following the load order. For proving whether the computer platform can be trusted to a remote challenger, the computer will send the signed PCR values and SML to the challenger according to the remote attestation protocol, and the challenger will check its validity and expectation. In opposite, the computer can also judge whether the platform of the challenger can be trusted.

The integrity can be measured either in the operating system (OS) or outside the OS. As in [3], we use 'in-the-box' and 'out-of-the-box' to denote these two kinds of methods, respectively. The in-the-box method generally relies on the security hooks provided by the monitored OS. For example, Linux kernel patched with Integrity Measurement Architecture (IMA) [4] or policy-reduced IMA [5] is able to measure the executable files and others, through Linux Security Module (LSM) or Linux Integrity Module (LIM). Catuogno *et al.* proposed an architecture to verify the digital signatures of executable files in the kernel model to prevent the malicious code [6]. SmartK is another approach that provides such security services in the kernel but using smart cards [7].

However, there are at least two disadvantages with hook-based methods: (1) modifications must be made to the monitored OS, and (2) such methods are vulnerable to kernel rootkits. The out-of-the-box methods can overcome these shortcomings. With the help of virtualization, if we stand out of the box, we can monitor the behaviors occurring in the box. Program integrity [8–11], file-system integrity [12], process execution [13], rootkit activities [14, 15], malware [16], and so on can be monitored or analyzed out of the box using virtual machine (VM) introspection method. Trusted computing has been applied in the system virtualization environment in many aspects up to now. For example, each guest VM in Xen [17] hypervisor (namely virtual machine monitor (VMM)) can use the trusted computing functions on the basis of its own virtual TPM chip [18]. Furthermore, these technologies can also be used in trusted virtual domains [19–21], trusted virtual datacenter [22], and some other secure distributed systems such as Shamon [23].

Whereas 'semantic gap' is a big problem in the out-of-the-box methods, the semantic view in the VM needs to be translated to fit the view outside the VM. Using the semantic-view-reconstruction methods proposed in [3], we can monitor the malware and system calls (syscalls) in VMware. XenAccess proposed in [24] is an introspection library for Xen, by which we can access the memory and syscall table directly at any moment but not periodically. Wang *et al.* [15] proposed an 'in-and-out-of-the-box' checking method, but this kind of method loses the transparence advantages. Cheng *et al.* [8] proposed a dynamic and transparent integrity measurement method, but it cannot measure all the libraries, script files, and Java Byte Code File (Class File). Quynh *et al.* [12] proposed a monitor tool that focused on file-system integrity, but not all operations to the file-system need to be checked, such as modifying users' normal documents and music files. What is more, the other similar approaches, which have been proposed in [9–11, 13, 25] in Xen virtual machine, Xen cloud platform, or KVM, are mostly based on virtual disk drivers or syscall interception. The former can capture I/O operations of a VM but has disadvantages in performance, while the latter is used in many approaches, but we find that not all programs will be loaded or executed by invoking syscalls. Meanwhile, these approaches also lack the consideration of the integrity of system configuration files. Overall, the existing approaches mainly focus on checking the integrity of executable files and shared libraries. But they can not measure the integrity of the following three kinds of files. The first is the system configuration files. Actually, the modification to system configuration files may affect the behaviors of system. The second is the script files and Java class files, which are executed as the input of the script interpreters or Java VM instead of being executed directly. The third is the files that are loaded or executed without trigging syscalls such as program loaders and script interpreters.

The aforementioned discussions motivate our work. In this paper, we propose a scalable and flexible out-of-the-box approach, which works outside the monitored guest VMs but measures the integrity of the files by intercepting syscalls. For scalable, we mean that the guest VMs do not need any modification, as the approach is transparent to the guest VMs. For flexible, we mean that the measure operation supports both system and user-specified policies. As this approach works similarly to IMA but stands out-of-the-box, we name it as *o*ut-of-the-*b*ox *i*ntegrity *m*easurement *a*pproach (OB-IMA).

We summarize the major contributions of this paper as follows:

(1) The proposed approach can not only measure the integrity of all executable files and kernel modules measured by both IMA and the existing out-of-the-box methods but also measure the integrity of various configuration, script files, and Java class files.
(2) The proposed approach can also measure the integrity of files that do not trigger syscalls, such as program loaders and script interpreters. To the best of our knowledge, it is the first work that considers the files that do not trigger syscalls.

Note that although all the discussions of this paper are in the context of Xen hypervisor and para-virtualization guest, the idea of this design can be extended to the other kinds of virtualization technologies. In this paper, we use critical file to denote the file that affects the behavior of the system and need to be measured. We call the other files as normal files. Furthermore, we assume the that hypervisor is secure, which means that the hypervisor will not modify the memory of guest VMs maliciously and the isolation mechanism can prevent the inter-domain attacks. We also assume that the kernel of guest VMs do not modify the memory of processes maliciously and the integrity of the kernel can also be judged. Our approach aims to obtain the information of critical files, but we do not try to analyze whether they are malicious as the approach proposed by Dinaburg *et al.*[16]. We leave the discretionary judgment ability to the challengers and users, and they can estimate the integrity of the computing platform using remote attestation protocol.

The rest of this paper is organized as follows. Section 2 gives the background and related work of 'out-of-the-box' based on VMM and trusted computing. Section 3 describes our approach in detail. Section 4 shows the experiments. Section 5 discusses some correlative topics, and Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

2.1. *Virtualization and out-of-the-box monitoring*

As an important feature of cloud computing, virtualization technology can separate resources into different domains and emulate hardware for each domain. Hypervisor, also named VMM, is the core of the virtualized system. It monitors the execution of all virtual domains. In general, Domain0 (Dom0) is the privilege domain, in which the host OS runs and the administrator can manage guest domains and VMM. Guest OS runs in DomainU (DomU, Guest Domain, or Guest VM), which is rented to users. Xen, KVM, VMWare, and Hyper-V are familiar VM software. Xen is an open-sourced VMM software, which supports two types of virtualization technologies: fully virtualization and para-virtualization. The VMM of the former type simulates independent virtual resources for each VM, in which the guest OS can run directly. In the latter type, the guest OS needs to be modified to adapt the VMM and can get higher performance.

On the basis of virtualization technology, we can enhance the security of guest VMs in some aspects. For instance, out-of-the-box methods can significantly improve the abilities of the malware monitoring and detection facilities. The reason is that this kind of methods are transparent to the programs run in the VMs, which prevents the security mechanism from the tampering of malicious programs.

The predicament in applying out-of-the-box methods is a well-known 'semantic gap' between the inside and outside view of the VM. An out-of-the-box approach loses the internal context of the guest OS, such as processes, modules, and files. Instead of that, from the outside view, we can watch the memory, register, disk image, and so on. So the critical problem is how to catch the context information and rebuild the semantic view.

2.2. *Trusted computing*

Trusted computing aims to upgrade the security of system based on improving the PC hardware architecture. A special chip named TPM contains the functions of random number generation, cryptographic key generation and management, encryption, secure storage, and so on, which can be used as the root of trust of a system. Here, word *trust* means the expectation that a device will behave in a particular manner for a specific purpose [26]. Namely, the computer will consistently runs in expected ways. When the platform powers up, the CRTM will measure the integrity of BIOS at the very beginning. Then the CRTM will pass the control to the BIOS if the BIOS is verified to be trusted. This process continues until the user applications are measured as a chain, named trusted chain. Finally, the trusted chain can be established as follows:

CRTM→BIOS→OS Loader→OS Kernel→Application

The IMA is a secure integrity measurement system for Linux, which establishes the trust relation between OS kernel and user applications. On the basis of the *file_mmap* hooks of LSM or LIM, or the modified syscalls (e.g., *init_module*), IMA can measure the integrity of the runtime executable programs, shared dynamic libraries, and kernel modules. After calculating the hash digest of a file, IMA extends the digest to the predefined PCR and adds it to a list. The list of measurement is kept inside the kernel space and can be used in the judgment of the system integrity during the remote attestation process. However, there is no suitable hooks for measuring the integrity of the non-executable files. So IMA cannot measure the integrity of various scripts, Java class files, system configuration files, and others.

The SML is a list that contains the sequence of measurement records. A typical record includes the PCR number, the filename, and its hash digest. The sequence of program loaded during the boot process may be different every time in multi-thread system, which means the PCR value is not a certain value in this situation. However, according to the records in the SML, the challenger can still re-compute the expect digest and compare it with the PCR value to judge the integrity of the system.

Trusted computing guarantees the integrity of a computing platform but cannot always stop attacks. The running of OS kernel, critical files, and the modification made to them will be recorded. Whether they can satisfy the security requirements is able to be judged by the challengers. The kernel vulnerabilities may give the chance to launch attacks without special program or tools. In this situation, trusted computing may not protect the platform because the kernel was still considered to be 'trusted' and no malware runs.

2.3. Syscalls intercepting

Syscall is the interface for user mode program to request kernel services. (e.g., accessing files and creating processes). Whenever receiving a syscall, the system turns into kernel mode and handles the syscall. As soon as the syscall processing is finished, the system turns back to the user mode and returns the results to the program. Both software interrupt and fast system call can request a syscall. Some older CPUs such as Intel Pentium only support the way of software interrupt. In this case, the user mode program puts the syscall number to the *EAX* register and generates the 0x80 interrupt. Fast system call supported by the later CPUs can fasten the requesting process, which means the user mode program only needs to set the value of *SYSENTER_EIP_MSR* register and use *SYSENTER*/*SYSEXIT* instructions.

In Linux systems, a tool named *strace* [27] can record all syscalls invoked by the specified program. In a virtualized system, we can also monitor and capture the syscalls by using API or modifying the VMM. But because of the semantic gap, the translation of memory address between inside and outside VM is very important.

In para-virtualized guest, the syscalls trap directly through the VMM, so the syscall interception in para-virtualization is easier. However, the fully virtualized guest needs the support of CPU, and it can invoke syscall with CPU directly like the normal OS, which makes the syscall interception to become very difficult. Frédéric *et al.* [28] showed an essential method to intercept syscalls in Xen hypervisor. Baiardi *et al.* also proposed similar methods in [29, 30] and discussed the application of their proposed methods in [31].

3. OB-IMA

In this section, we first describe the challenges of OB-IMA. Then we present OB-IMA.

3.1. Challenges

An out-of-the-box method must be able to catch the processes and the files dynamically loaded in the VMs at runtime with semantic contexts. There are commonly two ways to intercept the processes of VMs: (1) *CR3*-based approaches and (2) syscall-based approaches.

CR3 register contains the page directory address of the current running process, so the *CR3*-based approaches can detect the creation and termination of processes or access the memory of current process. However, using this kind of approaches is difficult to obtain rich semantic contexts, such as the name of process.

Syscall interception is also widely used in monitoring user behaviors. All processes and file operations in the user mode will trigger syscalls, but these operations in the kernel mode will not trigger any syscall. As mentioned earlier, syscall is the interface for user mode program to request kernel services.

We note that the interception of file operation is very important, because non-executable files may also affect the integrity of system. For instance, the modification of configuration files will affect the system behaviors, and the execution of the script files that loaded by interpreter but not run directly will also affect the system behaviors. So we still use the syscall-based approach in OB-IMA, but we must solve the following three challenges:

(1) How to use syscall interception to catch all the files executed or opened in VMs.
(2) How to catch the files that can be measured in IMA but do not trigger any syscalls.
(3) How to distinguish the critical file that should be measured from a normal file that needs not to be measured in VMs.

For the *first challenge,* there are several syscalls involved in process, file, or kernel module operations, such as *sys_execve*, *sys_open*, and *sys_init_module*. We can obtain the file name and path of the programs, file, or kernel module by capturing these syscalls invoked by the VMs and analyzing its parameters. We will give the detailed discussion in Section 3.2.1.

The *second challenge* is crucial and difficult to be handled. First, loading dynamic libraries in a program will not trigger syscall *sys_execve*. In fact, like opening configuration files, script files, and other normal files in user mode, opening the libraries also uses syscall named *sys_open*. If a user wants to install a kernel module, *sys_open* is also invoked before using *sys_insmod*. Second, except the files in the preceding text, there still exist some files that are loaded in the system but do not trigger any syscall. We will give the detailed discussion in Section 3.2.2 (1).

The second challenge brings out the *third challenge*: both critical files and normal files opened in user mode use *sys_open*. We plan to distinguish them by policies. For example, the files in directories such as */etc*, */sbin*, */bin*, */lib*, or */usr/bin*, */usr/lib.* may be the critical files, and the script files opened by interpreter should also be regarded as critical files. However, the files in directories such as */home* are most likely the normal files. The owners of VMs can also submit their manual-defined policies, which reflect the flexibility and discretion of our approach. We will give the detailed discussion in Section 3.2.2 (2).

3.2. The OB-IMA

Figure 1 describes the architecture of OB-IMA. There are three components: (1) syscall capturer in hypervisor, (2) integrity manager in Dom0, and (3) file hasher in storage servers or the nodes that store VM images. We call the node as *computing node* because the hypervisor, Dom0, and DomUs run in it. Once the user of VM starts up a program or opens a file, the behavior will be intercepted and filtered by the syscall capturer. The filtered record will be transmitted from the hypervisor to the integrity manager through *E-XenTrace*, which is originally used for trace or debug in Xen. Integrity manager maintains all PCRs and SMLs of VMs. When the integrity manager receives new interception records, it hashes the related files with the help of file hasher and then extends the hash digest to associated PCR. We will describe each component respectively in the following subsections.

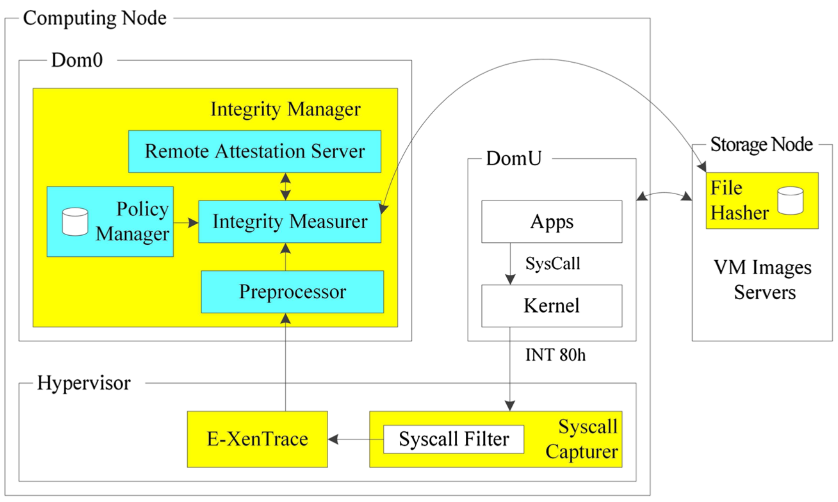


Figure 1. The archiecture of OB-IMA.

3.2.1. Syscall capturer. The syscall capturer has three functions. The first function is to intercept all the syscalls invoked by guest VMs and do some basic filtration to find the syscalls that are related to the integrity measurement. We called them as useful syscalls in the following. The second function is to record the parameters of useful syscalls, such as *domID*, *filename*, and *path*. Here *domID* denotes the ID of the guest VM, and *filename* and *path* denote the name and the path of the file. The third function is to write these records into the buffer shared between the hypervisor and Dom0 using *E-XenTrace* mechanism.

We implement the first function by adopting the method described in [28]. However, not all syscalls are the useful syscalls that we need to deal with. OB-IMA needs to obtain the program, kernel module, and other critical file loaded in the VMs. The related syscalls include as follows:

- Process operation: *sys_execve*, *sys_fork*, *sys_vfork*, *sys_clone*, *sys_exit*;
- Libraries operation: *sys_uselib*;
- File operation: *sys_open*, *sys_close*, *sys_mmap2*;
- Kernel module operation: *sys_init_module*, *sys_delete_module*.

In process operations, *sys_fork* and *sys_vfork* split an existing process into two new processes. Syscall *sys_clone* can start a new process or threads, and *sys_exit* is used for ending up a process. If a user wants to run a new program through the file-system, *sys_execve* will be invoked after the process creation. In other operations, *sys_uselib* allows a process to load a dynamic library; *sys_open* is responsible for opening a file or directory, and *sys_close* will release the opened resources; *sys_mmap2* can map a file or other object into memory; syscall *sys_init_module* and *sys_delete_module* are in charge of inserting and removing kernel module, respectively.

The aforementioned discussion indicates that the program, kernel module, and file loading operations are all involved in *sys_execve*, *sys_uselib*, *sys_open*, *sys_mmap2*, and *sys_init_module*. However, *sys_init_module* and *sys_mmap2* rely on *sys_open*, because the modules or files must be opened before being copied to the memory. So here, we mainly focus on ***sys_execve***, ***sys_open***, and ***sys_uselib***.

For the second function, we can obtain *domID* from the syscall interception functions. At the same time, we can obtain *filename* and *path* by capturing these syscalls invoked by the VMs and analyzing the parameters of them. The first parameter of *sys_execve*, *sys_open*, and *sys_uselib* is the filename string. When these syscalls are invoked, the *filename* can be found at the address contained in register *EBX*. In *sys_execve*, *ECX* and *EDX* contain the arguments and environment variables. Through syscalls interception, we can record the programs running in the VM or the files opened in the VM. In addition, if we know the detail information of the kernel of a VM, we can even find out which user runs that program.

Although syscall capturer can find what program is running or what file is opened in a guest VM, we prefer to measure the integrity of these files in Dom0 instead of directly in hypervisor space for three reasons: (1) the quantity of syscalls invoked by guest VMs is very large, (2) measuring the file integrity in hypervisor space wastes the compute resources and then affects the speed of handling other resource requests of guest VMs, and (3) Xen hypervisor does not have network drivers, which means that transmitting a file from storage server to hypervisor space is a bit complicated.

We next explain the third function. To measure the integrity of the guest VM files in Dom0, the syscall capturer needs to send the *domID*, filename, and path as a record to a daemon program run in Dom0. Here we use the enhanced *XenTrace* [32] mechanism, namely *E-XenTrace*. *XenTrace* is a good mechanism to transmit data to Dom0, but a fly in the ointment is the length limitation of trace record. The max length of each record is 40 bytes, includes 4 bytes for header, 8 bytes for *time stamp counter* (TSC), and 28 bytes for trace data (seven fields of long type data). In the header structure, there are 28 bits for *eventID*, 3 bits for the quantity of trace data (from 0 to 7), and last 1 bit for whether *TSC* exists. In order to reduce the modification of Xen hypervisor and keep the compatibility, we define a special *eventID* and enlarge its max record length. That means, if the *eventID* of a record is equal to our definition in advance, the length of data can be extended up to 180 bytes, including 140 bytes of *filename* and *path*, 10 long type parameters for *domID*, *syscall number*, and so on. The data can be lengthened continuously if necessary.

3.2.2. Integrity manager. Integrity manager has four modules: preprocessor, policy manager, integrity measurer, and remote attestation server. All of them are run as daemons. We present each module in the following.

(1) Preprocessor

Preprocessor is responsible for reading the records of syscall parameters written in the shared buffer using *XenTrace* API and getting the *filename* and *path* of files that be executed or opened directly in guest VMs. But we should note that not all the critical files can be caught through syscall interception, such as the program loader. So we need to discover them on the basis of the records received. Here we first analyze the process of executing an program and then describe how the preprocessor works.

Executable and linkable format (ELF, formerly called extensible linking format) file is the main kind of executable programs in Linux. Once a user mode program has been mapped from an ELF file into memory, the OS passes the control to its program loader to start up the program, as shown in Figure 2. The *INTERP* field of the header structure of an ELF file gives the name of program loader. Meanwhile, most programs that run on Linux are dynamically linked nowadays, which means that they need to load the dynamic libraries once they start to run. In the GNU systems and most systems with Linux kernel, *glibc* (the GNU C Library) sets the program loader to */lib/ld-linux.so.** (e.g., */lib/ld-linux.so.2*) when compiling a program. Generally speaking, file */lib/ld-linux.so.2* is a symbolic link of */lib/ld-2.5.so* (it might be different under some special platform), and it will tell the program where to find the dynamic libraries.

Program loader */lib/ld-linux.so.** should be opened before using it. However, as shown in Figure 3, when we use *strace* to record the syscalls invoked by an ELF file, we cannot find any operation of program loader in the logs. Hence, the general syscall interception cannot catch the program loader outside the VMs either. That is because syscall is only invoked in user mode. When *sys_execve* is invoked, the system enters kernel mode and does some initializations, such as mapping the program into memory. Then it opens the program loader, passes the control to the loader, and turns back to user mode. Therefore, opening the loader is still in kernel mode that will not trigger a syscall, as shown in Figure 2.

Script file is another kind of executable files. As discussed previously, if we directly run the script file that has the executable attribute, the script interpreter (which can also be considered as program loader) will not trigger syscall either. Figure 4 shows the experiment result of tracing a python script

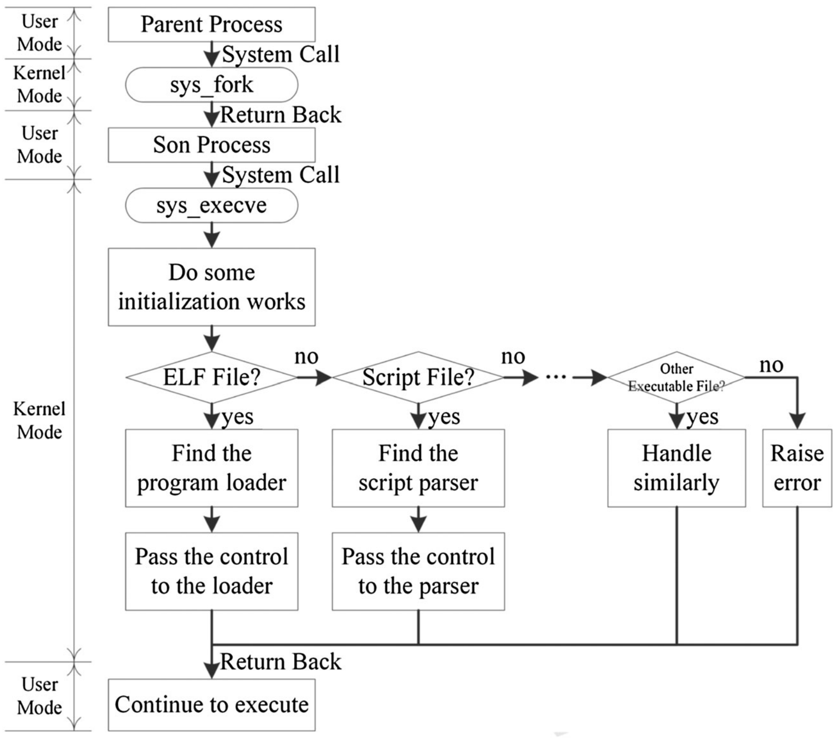


Figure 2. The process of executing a new program.

```
[root@xingbin-pc lab]# cat /root/lab/hello.c
#include <stdio.h>
int main(void)
{
  printf("Demo: Hello World\n");
}
[root@xingbin-pc lab]# gcc /root/lab/hello.c
[root@xingbin-pc lab]# /root/lab/a.out
Demo: Hello World
[root@xingbin-pc lab]# strace /root/lab/a.out
execve("/root/lab/a.out", ["/root/lab/a.out"], [/* 46 vars */]) = 0
brk(0)                                  = 0x8374000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7796000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=84319, ...}) = 0
mmap2(NULL, 84319, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7781000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0@n\1\0004\0\0\0"..., 512) = 512

……
write(1, "Demo: Hello World\n", 18Demo: Hello World
)     = 18
exit_group(18)                          = ?
[root@xingbin-pc lab]#
```

Figure 3. The main syscalls issued by ELF program.

file. But different from ELF files, the script file is an ASCII file and does not contain the header structure. According to the magic code ‘#!’, there are two kinds of script interpreters:

(1) If the first line of script file is **not** started with magic code ‘#!’, then this file will be interpreted by the default shell interpreter (e.g., bash). The default shell interpreter can be found by looking up the user settings profile. After that, the interpreter can be regarded as a new executable program and be processed following the same method. The rest of the script files require no

```
[root@xingbin-pc lab]# cat /root/lab/hello.py
#!/usr/bin/python
print 'python Demo: Hello World'
[root@xingbin-pc lab]# chmod 0755 hello.py
[root@xingbin-pc lab]# /root/lab/hello.py
python Demo: Hello World
[root@xingbin-pc lab]# strace /root/lab/hello.py
execve("/root/lab/hello.py", ["/root/lab/hello.py"], [/* 46 vars */]) = 0
brk(0)                                  = 0x8afa000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb77ef000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=84319, ...}) = 0
mmap2(NULL, 84319, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb77da000
close(3)                                = 0
open("/usr/lib/libpython2.6.so.1.0", O_RDONLY) = 3

……
open("/root/lab/hello.py", O_RDONLY|O_LARGEFILE) = 3

……
read(3, "#!/usr/bin/python\nprint 'python "..., 4096) = 51

……
write(1, "python Demo: Hello World\n", 25) = 25
rt_sigaction(SIGINT, {SIG_DFL, [], 0}, {0xd8fa50, [], 0}, 8) = 0
exit_group(0)                           = ?
[root@xingbin-pc lab]#
```

Figure 4. The main syscalls issued by python scripts.

more special treatments, because the internal commands of the script files are the built-in commands that need not to be checked, and the external commands are actually other programs and will trigger new syscalls that can also be caught and checked.

(2) If the first line of script file is started with magic code '#!', then the path of the interpreter is set just after the magic code. Then we can find the interpreter and check its integrity. The script file that is executed by user manual specified interpreter, or is not written by shell language, must have the interpreter setting, or the system will raise an error.

Note that there is another special case: it is known that Linux OS allows different users who may want to use different versions of interpreter to execute the same script file. In this case, the interpreter is commonly specified as *env* program with the name of interpreter as parameter, such as '*/usr/bin/env python*'. This is the most complex case, because we need not only to measure *env* but also to measure the *python* in right path. So we need to know which user runs this script and then find the interpreter. OB-IMA handles this challenge by exploiting the information of *task_struct* of the related process.

Note that there is another method to execute script files: pass its filename as a parameter to the interpreter in command line, such as '*python /home/user/test.py*'. In this way, the interpreter triggers *sys_execve*, and then script file triggers *sys_open*. All of them will be intercepted by OB-IMA.

It is very interesting when we face the Java class file or Java Archive File (JAR File). These files are executed by running *java* command. However, we cannot find syscall *sys_open* simply by running *strace* tool without some special parameters. That is because *java* command will do some initialization operations first and then start a new sub-process to load the class and JAR files and run them. Because *java* is a user mode command, it cannot load user files without *sys_open*. So all Java class file, JAR files, and imported files can be intercepted by OB-IMA.

On the basis of the aforementioned discussion, the preprocessor module in integrity manager will do the following things: (1) for ELF files, it finds out the program loaders, (2) for script files, it finds out the script interpreters, and (3) for other files, it judges whether the files are critical according to the policies predefined in the system or manually defined by owners. After that, it sends the *domID*, *filename*, and *path* of the origin files and related files to the integrity measurer by pipe file (a first-in-first-out named pipe, which can transfer data between processes).

Besides ELF and script files, Linux also supports some other formats of executable files, such as *a.out*, *som*, and *em86*. These file formats are not very common, but they can also be handled similarly as previously discussed. Therefore, we do not explain the details of these formats in our approach. The experiment results in Section 4 show that discovering the critical file is important.

(2) Policy manager

As mentioned before, we only take *sys_execve*, *sys_open*, and *sys_uselib* into account. There is no doubt that the programs and libraries loaded by *sys_execve* and *sys_uselib* will influence the integrity of the system. However, all of kernel modules, configuration files, script files, and other normal files are opened by *sys_open* in user mode. Suitable policies are required to distinguish the critical files from the normal files such as user data and temporary files. Policy manager is responsible for collecting the user-specified policies submitted by the users and providing the measure decision to the integrity measurer based on the policies and VM name.

The policies include three types of rules:

- *White list*: the files satisfying the rules defined in *white list* will not be measured;
- *Pre-measured list*: the files satisfying the rules defined in *pre-measured list* must be measured before the VM starts, unless they satisfy the rules in *white list*;
- *Runtime rules*: the files satisfying the rules defined in *runtime rules* will be measured once they are opened by invoking syscall *sys_open*, unless they satisfy the rules in *white list*.

The rules in *white list* or *pre-measured list* must give the full path and name of files, while *runtime rules* are very flexible—user can set the full path and file name, only path, or even only the file magic code. Figure 5 shows the demo of policy. The rules commented by ‘<!–’ and ‘–>’ are alternative rules but not used in the experiments in Section 4.

(3) Integrity measurer

Integrity measurer is in charge of maintaining the PCR and SML of each guest VM. Once receiving the record from the preprocessor, integrity measurer checks whether the record is from a new guest VM or an existing guest VM according to *domID*. For a new guest VM, the integrity measurer adds a new group of PCR and SML. After that, it looks up *XenStore* (a small database in Xen) with *domID* and finds the storage settings of the specified guest VM, for instance, the name and storage place of disk image file, or logical volume name on NFS (Network File System) servers. Then, it sends the *filename*, *path*, and the storage place to the file hasher, and waits for the result. After receiving the normal result, it adds the measure record in the SML and extends the digest into the PCR of that guest VM. If it receives an error message, it checks the return code of the corresponding syscall to judge whether the file was actually opened successfully or not. The acknowledgement message generated by the file hasher can be ignored, because unchanged file does not affect the SML, the PCR value, and the integrity of system.

```
<?xml version="1.0" encoding="utf8" ?>
<policy domainname="">
    <whitelist>
        <!-- this is a white list demo: <rule filename="/root/a.txt" /> -->
    </whitelist>
    <premeasure>
        <!-- this is a pre-measure list demo: <rule filename="/boot/vmlinuz" /> -->
    </premeasure>
    <runtimemeasure>
        <rule filepath="/bin" />
        <rule filepath="/sbin" />
        <rule filepath="/lib" />
        <rule filepath="/etc" />
        <rule filepath="/usr/bin" />
        <rule filepath="/usr/sbin" />
        <rule filepath="/usr/lib" />
        <rule filemagiccode="0x2321" /><!-- Script Files -->
        <rule filemagiccode="0x7F454C46" /><!-- Executable and Linkable Format Files -->
        <rule filemagiccode="0xCAFEBABE" /><!-- Java Byte Code Files -->
        <!-- this is a runtime rule demo: <rule filename="/root/demo.out" /> -->
    </runtimemeasure>
</policy>
```

Figure 5. A demo policy.

(4) Remote attestation server

On the basis of the PCR and SML, the integrity manager can provide remote attestation service to the VM owner. The remote attestation server waits for the challengers' requests and responds them. The process of the attestation is similar to the TCG remote attestation process except that the PCR value is not stored in TPM chip but the memory of Dom0. Hence, the integrity of Dom0 should also be checked during the remote attestation process.

3.2.3. File hasher. File hasher is responsible for hashing the files specified by integrity manager and sending back the hash digest. It is also a daemon program but located in the storage servers or the nodes that store VM images, for reducing the internal traffic of file transfer. Meanwhile, the request process can be accelerated by using a measure records database. Whenever file hasher receives a request from integrity measurer, it looks up its database to find whether the measure record of the requested file is exist, and then opens the image file or the volume and checks the status of the specified file. Figure 6 shows the processes of the four possible situations.

(1) If the specified file does not exist, the file hasher will send back an error message to the integrity measurer.
(2) If there is no measurement record of this file in the database, namely, the file has not been measured before, the file hasher will hash it and send back the hash digest to the integrity measurer. Moreover, it records this measure operation in the database.
(3) If the file had been measured but has been changed after last measurement, the file hasher will hash it again and send back the new hash digest to integrity measurer.
(4) If the file had been measured and has not been changed after last measurement, the file hasher will send back an acknowledgement message to integrity measurer.

At last, if the file hasher is notified by the integrity measurer that a guest VM is terminated, it will clean up the measurement record of that guest VM.

4. EXPERIMENTS AND EVALUATION

In this section, we evaluate our approach via experiments in order to demonstrate the following: (1) that OB-IMA is not only able to catch and measure all files that have been measured in IMA but

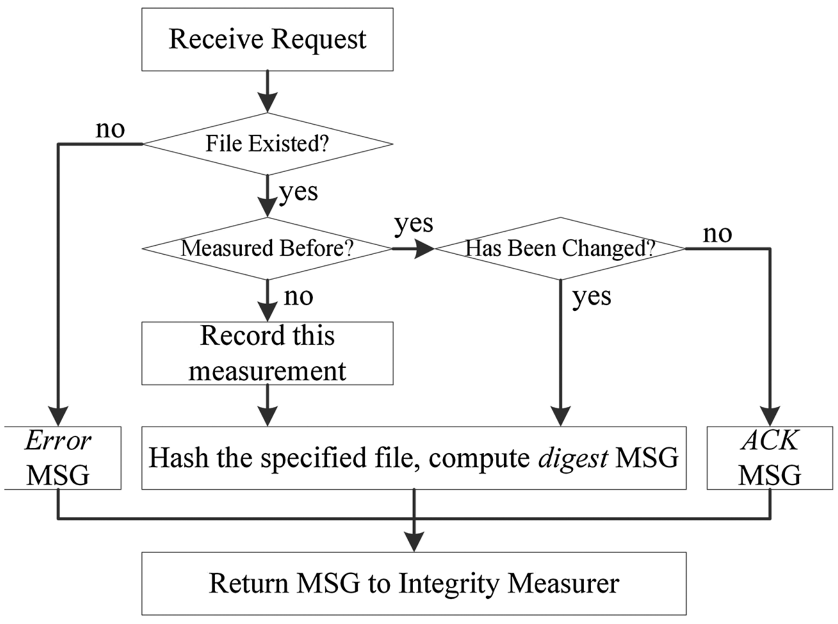


Figure 6. The flow dDiagram of file hasher.

also able to measure the system configuration files, script files, and so on, and (2) that the overhead of this approach is low.

Our experimental testbed consists of a PC with a 2.5 GHz Intel Xeon L5420 quad-core CPU, 8 GB RAM, and 1T SATA2 3.0 Gb/s desktop-level hard disk. We implement our approach in CentOS 5.8 and Xen 3.4.4 using 2.6.18.8 Linux kernel. Each guest VM was allocated a single vCPU, 512 M RAM, 5 G disk image, employed CentOS 5.8 using 2.6.18.8 Linux kernel patched with IMA in test mode. Without loss of generality, we use local hard disk instead of special storage servers, so the file hasher runs on the computing node actually. Meanwhile, we use the measure policy listed in Figure 5 for the following experiments.

4.1. *Experiment of measuring ability*

In this experiment, we check whether the files that have been measured by IMA can also be measured by OB-IMA. We boot the same guest VM in two different environments. For IMA, we use the origin Xen as the VMM, but the kernel of guest VM is patched with IMA. For OB-IMA, we modify the Xen and Dom0 to implement our approach, but there is no modification in the kernel of guest VM. Table I gives the SML statistics of the boot process.

From Table I, we observe the following: the IMA approach measures 256 files, and they can also be measured by our approach. Among them, 184 files have the same filename, and the rest are soft links that point to the origin files. In addition, OB-IMA measures 1563 files more than IMA, which makes the integrity of the system more trustworthy and real. Figure 7 shows some logs recorded by OB-IMA. The bold logs are the files that cannot be measured by IMA, and each of them falls into one of the following four categories:

- System configuration files, such as */etc/inittab*.
- Shared libraries, such as */lib/libc.so.6*.
- Script modules, such as */usr/lib/python2.4/os.pyo*.
- Non-executable script files (which loaded by script interpreter), such as */home/xingbin/hello_nonexecutable.py*.

Table I. Amount of files measured by different approaches.

		Same file		
Approach	Total measured	Same name	Soft link	Different file
IMA	256	184	72	0
OB-IMA	1819			1563

IMA, integrity measurement architecture; OB-IMA, out-of-the-box integrity measurement approach.

```
......
LINE0020     1f4f2e1b1b63a3aa55db59316f3d2a31fa97fda3     /etc/inittab
LINE0021     1ad5cc8d2714b510e4c47141580e6cd5f77d0e1b     /etc/rc.d/rc.sysinit
LINE0022     c4017526c62b97e3a22ecf2257341fa43c162119     /bin/bash
LINE0023     e7452e2080de62aff08d6a36939b61e491ce5dee     /lib/libtermcap.so.2
......
LINE0366     086e1c532e37c87be10549236367cb7f20c3a7e9     /lib/libc.so.6
LINE0367     4368b6cf54d17c7d4eabdd05b1ae558c07df6024     /lib/libcrypt.so.1
......
LINE0514     065da18336507050a845244038fb29590cb05312     /usr/lib/python2.4/os.py
LINE0515     e0941b60d202b288f0eb14676b2b50fce16237bc     /usr/lib/python2.4/os.pyo
......
LINE1159     84728a23eb43021d056fdd9c5aa449df4872fd3e     /home/xingbin/boottime.sh
......
LINE1184     cdc3e189a026f4f82547fc6766b38d6641212ca5     /home/xingbin/hello_nonexecutable.py
```

Figure 7. Part of the logs recorded by OB-IMA.

Table II. Benchmark test.

	Time (usecond)		
Benchmark	IMA	OB-IMA	Overhead
open_usr	1.16264	1.73877	49.5536%
exec	414.0637	450.3123	8.7544%
getpid	0.00358	0.00361	0.8380%
memcpy_10	0.03556	0.03562	0.1687%
fork_10	181.6635	176.9105	−2.6164%

IMA, integrity measurement architecture; OB-IMA, out-of-the-box integrity measurement approach.

Table III. The execution time of the test case.

	C			Java			Python		
	Real	User	Sys	Real	User	Sys	Real	User	Sys
IMA (ms)	38.462	8.646	29.584	303.110	255.445	31.288	285.058	247.429	37.181
OB-IMA (ms)	43.081	8.897	34.035	318.191	260.687	42.232	292.691	248.164	43.840
Overhead (%)	12.01	2.91	15.05	4.98	2.05	34.98	2.68	0.30	17.90

IMA, integrity measurement architecture; OB-IMA, out-of-the-box integrity measurement approach.

4.2. *Experiment of overhead*

We evaluate the overhead from three aspects: (1) benchmark test, (2) program overhead test with different programing languages, and (3) VM boot time test.

First, we evaluate the performance of OB-IMA with a number of benchmarks using libMicro 0.3 [33]. LibMicro is a portable set of microbenchmarks for measuring the performance of various system and library calls. The benchmarks of libMicro 0.3 has 259 terms, which include memory, string, process, and I/O operations. The benchmark report shows the following: (1) that the overhead of 84 terms is lower than 10% and the overhead of 54 terms is lower than 2%; (2) that the overhead of 79 terms is more than 50% but almost all the execution time of them are less than 1 ms (usecond); (3) that the overall overhead is 12.12%. Because of space limitation, Table II only gives the results of some benchmarks that are frequently used or involved in *sys_execve* and *sys_open*[‡].

The results in Table II indicate that the overheads of *open* and *exec* operation are higher. That is because the interception affects these two syscalls process. Meanwhile, we find that the overhead of *open_usr* is obviously higher than that of *exec*. The possible reason is that our experiments use desktop-level hard disk instead of network storage to store the VM disk image and this hard disk influences the performance of small-size but heavy-amount I/O operations. Using professional and high performance storage devices can also reduce the overhead of I/O operation in a certain extent.

Second, we evaluate the performance of test case written in C, Java, and Python. The mission of the test case is executing a program, which opens 1000 different files and writes 100 lines to each file in every execution. We run the program 50,000 times respectively in OB-IMA approach and IMA approach and calculate the runtime by using *time* tool in Linux. Here we choose 1000 different files and 50,000 times in order to reduce the errors and the impacts of system cache. The results are listed in Table III. The columns of *real*, *user*, and *sys* denote the elapsed real time, the user CPU time, and the system CPU time, respectively.

From the results, we can find the following: (1) that the OB-IMA approach mainly impacts the system CPU time and the influence on the user CPU time is much lower and (2) that the overall overhead of ELF program (written in C) is about 12.01%, while the overall overhead of the script file and Java byte code file is less than 5%.

[‡]The full results of benchmarks test can be found at http://www.xingbin.net/research/obima/benchmark.htm.

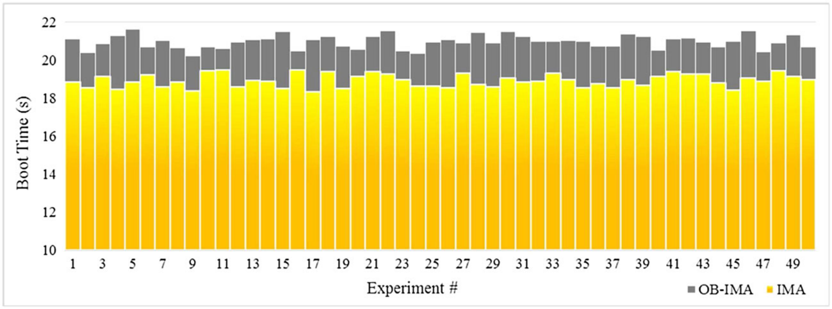
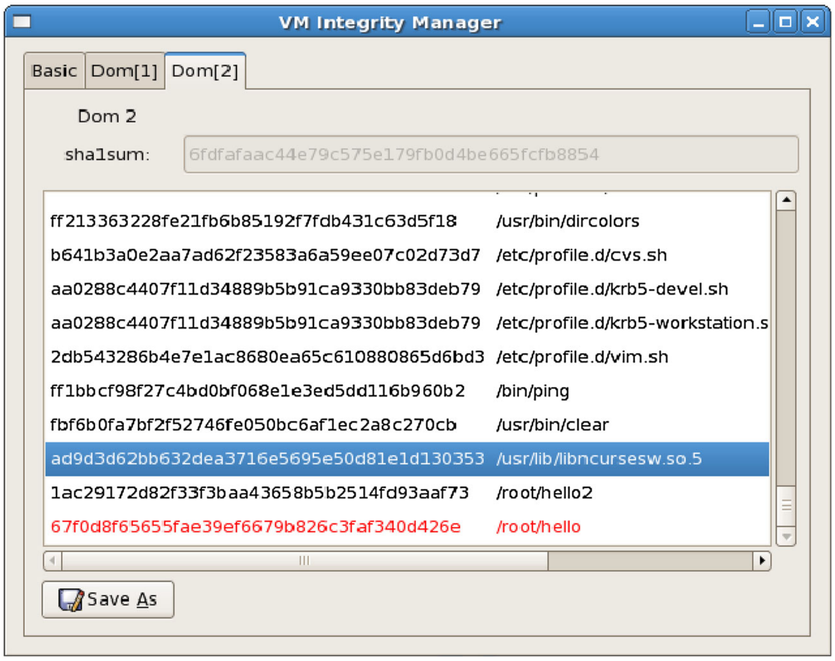


Figure 8. Comparison of boot time.



Dom 2

sha1sum: 6fdfafaac44e79c575e179fb0d4be665fcfb8854

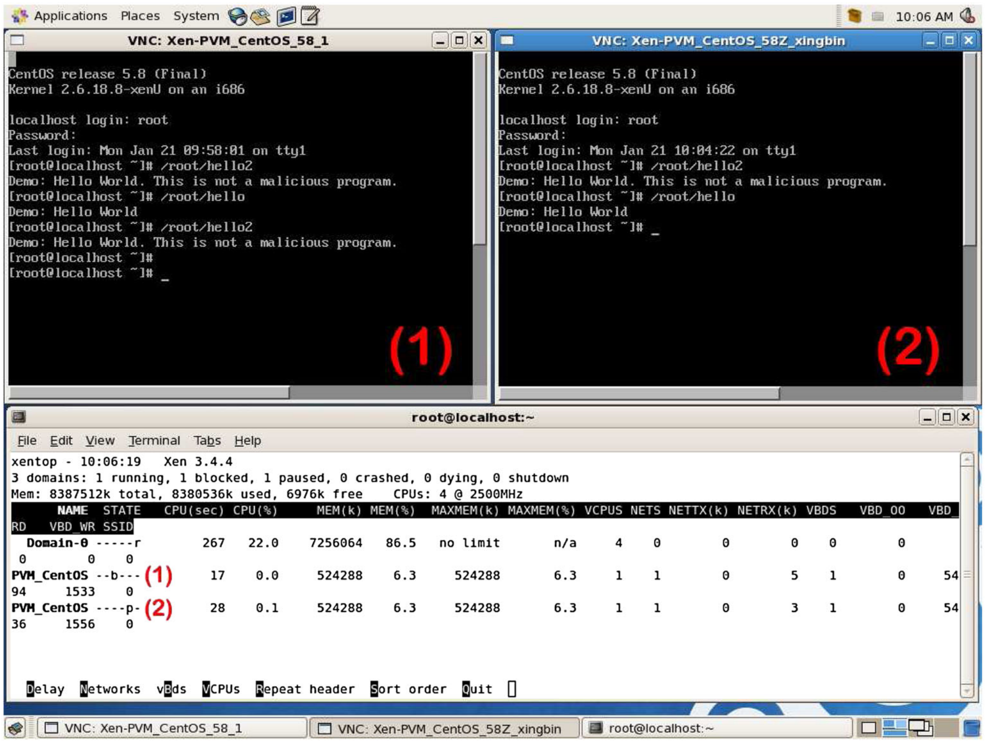
```
ff213363228fe21fb6b85192f7fdb431c63d5f18   /usr/bin/dircolors
b641b3a0e2aa7ad62f23583a6a59ee07c02d73d7   /etc/profile.d/cvs.sh
aa0288c4407f11d34889b5b91ca9330bb83deb79   /etc/profile.d/krb5-devel.sh
aa0288c4407f11d34889b5b91ca9330bb83deb79   /etc/profile.d/krb5-workstation.s
2db543286b4e7e1ac8680ea65c610880865d6bd3   /etc/profile.d/vim.sh
ff1bbcf98f27c4bd0bf068e1e3ed5dd116b960b2   /bin/ping
fbf6b0fa7bf2f52746fe050bc6af1ec2a8c270cb   /usr/bin/clear
ad9d3d62bb632dea3716e5695e50d81e1d130353   /usr/lib/libncursesw.so.5
1ac29172d82f33f3baa43658b5b2514fd93aaf73   /root/hello2
67f0d8f65655fae39ef6679b826c3faf340d426e   /root/hello
```

Figure 9. The screenshot of integrity manager demonstration.

Third, we evaluate the overload of booting process. We start up the guest VM with IMA or OB-IMA 50 times respectively and calculate the average boot time. The boot time is automatically calculated by reading */proc/uptime* once the OS is ready. The average boot time of IMA approach is 18.920 s, while the time of OB-IMA is 20.967 s, as shown in Figure 8. Here 'experiment #' denotes the index of the booting. The overhead is 10.82%, which is close to the results of other experiments shown earlier and is still acceptable. We should note that OB-IMA measures 1819 files, and this amount is 5.1 times higher than that of IMA. Meanwhile, the desktop-level hard disk also affects the performance of all the experiments in some extent.

5. DISCUSSION

Figure 9 is the screenshot of the integrity measurer. Figure 10 is the screenshot of the guest VMs and their status at runtime. As shown in these screenshots, the OB-IMA approach supports multiple guest VMs run at the same time, as the *domID* caught in the interception specifies different guest VMs. Meanwhile, we can make different policies for each guest VM. For example, program */root/hello* is judged to be a normal program in VM1 but a malicious program in VM2. After the execution in each VM, the VM1 is still in normal state, but the VM2 is paused by the integrity measurer. In Figure 9,


Applications Places System
10:06 AM
VNC: Xen-PVM_CentOS_58_1
VNC: Xen-PVM_CentOS_58Z_xingbin
(1)
(2)
root@localhost:~
File Edit View Terminal Tabs Help
VNC: Xen-PVM_CentOS_58_1
VNC: Xen-PVM_CentOS_58Z_xingbin
root@localhost:~

```
CentOS release 5.8 (Final)
Kernel 2.6.18.8-xenU on an i686

localhost login: root
Password:
Last login: Mon Jan 21 09:58:01 on tty1
[root@localhost ~]# /root/hello2
Demo: Hello World. This is not a malicious program.
[root@localhost ~]# /root/hello
Demo: Hello World
[root@localhost ~]# /root/hello2
Demo: Hello World. This is not a malicious program.
[root@localhost ~]#
[root@localhost ~]# _
```

```
CentOS release 5.8 (Final)
Kernel 2.6.18.8-xenU on an i686

localhost login: root
Password:
Last login: Mon Jan 21 10:04:22 on tty1
[root@localhost ~]# /root/hello2
Demo: Hello World. This is not a malicious program.
[root@localhost ~]# /root/hello
Demo: Hello World
[root@localhost ~]# _
```

```
xentop - 10:06:19   Xen 3.4.4
3 domains: 1 running, 1 blocked, 1 paused, 0 crashed, 0 dying, 0 shutdown
Mem: 8387512k total, 8380536k used, 6976k free    CPUs: 4 @ 2500MHz
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k) MAXMEM(%) VCPUS NETS NETTX(k) NETRX(k) VBDS   VBD_OO   VBD_
RD   VBD_WR SSID
  Domain-0 -----r        267   22.0    7256064   86.5   no limit       n/a     4    0        0        0    0        0
 0        0    0
PVM_CentOS --b--- (1)     17    0.0     524288    6.3     524288       6.3     1    1        0        5    1        0    54
94     1533    0
PVM_CentOS ----p- (2)     28    0.1     524288    6.3     524288       6.3     1    1        0        3    1        0    54
36     1556    0


  Delay  Networks  vBds  VCPUs  Repeat header  Sort order  Quit  
```

Figure 10. The screenshot of the guest VMs and their status at runtime.

we can find that the record of malicious program is shown by red font. This implementation also works well with other versions of Xen, such as Xen 3.x or 4.x.

Moreover, the OB-IMA approach is transparent to the guest OS; thus, the guest VM that runs FreeBSD, Open Solaris, and other UNIX-like OSes can also be supported. Meanwhile, the OB-IMA approach does not require many modifications to the existing syscall interception method, because the parameters and some related information of the syscalls are needed to be obtained. So it can be ported easily to other hypervisors or virtualization platforms if they has similar syscall interception methods.

There are some attacks that use rootkits, kernel vulnerabilities or malicious kernel modules to modify the kernel routines. They may change the syscall table, make return-oriented programming attacks [34–36], or modify the kernel data structures[37]. These attacks can damage system data or even subvert the VM introspection. The OB-IMA approach may not defend against all attacks, but it can still discover most attacks, because the execution of the first malware will be intercepted, and the challenger is able to find the attacks in remote attestation. With whitelist limitation rules, any unknown kernel modules and programs cannot be loaded and executed, and the OB-IMA can pause the VM or warn either the owner or user. However, not all attacks can be discovered by OB-IMA, because trusted computing can guarantee only the integrity but not fully security of a computing platform. For example, the attacks using vulnerabilities of a 'trusted' kernel directly without any malware or tools cannot be stopped, because the platform configuration is not changed from the view of trusted computing. The administrator can avoid these attacks by checking the security bulletin and updating the kernel periodically.

However, we find that there is a way to escape the OB-IMA approach. For example, if the owner of guest VM modifies the kernel, changes syscall number of *sys_open*/*sys_execve*, or creates new syscalls to replace them before the VM starts up, then our approach will fail to catch the changed syscalls. That is because we only intercept the preset syscalls to enhance the performance. But as a security service provided by CSP, the rules should be established particularly to avoid this situation, and the user who wants to use this service should follow the relevant rules. Meanwhile, the integrity of the kernel can also be judged by the pre-measured process as long as putting the kernel filename into *pre-measured list*.

Last but not the least, we should notice that the privacy protection in cloud computing is a widely concerned topic. The approach in this paper is most suitable for private cloud that needs peculiar secure protection. This approach can also be used in other deployment models of cloud computing, such as public cloud. But becasue measurement operation requires the access permission of guest VM disk image, the CSP still needs the user's authorization or follows the service level agreement to access users' data if they wish to use the integrity service.

6. CONCLUSION AND FUTURE WORK

In this paper, we propose a flexible 'out-of-the-box' approach to measure and check the integrity of critical files of guest VMs without any modification to guest VMs. The 'out-of-the-box' feature makes this approach transparent to guest VMs, which protects the approach without the tamper from the malware working in the guest VMs. Different from IMA and the existing out-of-the-box approaches, our approach considers the integrity of script files, system configuration files, and so on even if these files cannot be caught through kernel interception directly. We implement the OB-IMA approach in Xen and Linux, and the experiment results demonstrate the effectiveness and efficiency of this approach. In our future work, we plan to extend the interception in fully virtualization, consummate the approach with migration and remote attestation protocol, and evaluate this approach again with high performance network storage devices.

ACKNOWLEDGEMENTS

We would like to thank Wei Wang for discussions and feedback. We also would like to thank the reviewers for their comments. The work described in this paper has been supported in part by Research Fund for the Doctoral Program of Higher Education of China (No. 20120009110007), Program for Innovative Research Team in University of Ministry of Education of China (No. IRT201206), Program for New Century Excellent Talents in University (NCET-11-0565), and the Fundamental Research Funds for the Central Universities (No. 2012JBZ010).

REFERENCES

1. Trusted Computing Group. (Available from: https://www.trustedcomputinggroup.org/) [Accessed on 1 July 2013].
2. Trusted Computing Group. *Trusted platform module (TPM) specifications*. (Available from: https://www.trustedcomputinggroup.org/specs/TPM) [Accessed on 1 July 2013].
3. Jiang X, Wang X, Xu D. Stealthy malware detection through VMM-based "Out-of-the-Box" semantic view reconstruction. *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS),* October 2007, ACM: New York, 2007; 128–138. DOI: 10.1145/1315245.1315262.
4. Sailer R, Zhang X, Jaeger T, Doorn LV. Design and implementation of a TCG-based integrity measurement architecture. *Proceedings of the 13th USENIX Security Symposium (SSYM),* August 2004, USENIX Association: Berkeley, 2004; 16–16.
5. Jaeger T, Sailer R, Shankar U. PRIMA: policy-reduced integrity measurement architecture. *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT),* June 2006, ACM: New York, 2006; 19–28. DOI: 10.1145/1133058.1133063.
6. Catuogno L, Visconti I. An architecture for kernel-level verification of executables at run time. *The Computer Journal* 2004; **47**(5):511–526. DOI: 10.1093/comjnl/47.5.511.
7. Catuogno L, Gassirà R, Masullo M, Visconti I. SmartK: smart cards in operating systems at kernel level. *Information Security Technical Report* 2013; **17**(3):93–104. DOI: 10.1016/j.istr.2012.10.003.
8. Cheng G, Jin H, Zou D, Zhang X. Building dynamic and transparent integrity measurement and protection for virtualized platform in cloud computing. *Concurrency and Computation: Practice and Experience* 2005; **22**(13): 1893–1910. DOI: 10.1002/cpe.1614.
9. Azab AM, Ning P, Sezer E C, Zhang X. HIMA: a hypervisor-based integrity measurement agent. *Proceedings of the 25th Computer Security Applications Conference (ACSAC),* December 2009, IEEE Computer Society: Los Alamitos, 2009; 461–470. DOI: 10.1109/ACSAC.2009.50.
10. Neisse R, Holling D, Pretschnet A. Implenting trust in cloud infrastructures. *Proceedings of 11th the IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID),* May 2011, IEEE Computer Society: Washington, 2011; 524–533. DOI: 10.1109/CCGrid.2011.35.

11. Zou B, Zhang H. Integrity protection and attestation of security critical executions on virtualized platform in cloud computing environment. *Proceedings of the 2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing,* August 2013, IEEE Computer Society: Los Alamitos, 2013; 2071–2075. DOI: 10.1109/GreenCom-iThings-CPSCom.2013.388.
12. Quynh NA, Takefuji Y. A novel approach for a file-system integrity monitor tool of Xen virtual machine. *Proceedings of the 2nd ACM Sumposium on Information, Computer and Communications Security (ASIACCS),* March 2007, ACM: New York, 2007; 194–202. DOI: 10.1145/1229285.1229313.
13. Srinivasan D, Wang Z, Jiang X, Xu D. Process out-grafting: an efficient "Out-of-VM" approach for fine-grained process execution monitoring. *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS),* October 2011, ACM: New York, 2011; 363–374.
14. Hwang T, Shin Y, Son K, Park H. Design of a hypervisor-based rootkit detection method for virtualized systems in cloud computing environments. *Proceedings of the 2013 AASRI Winter International Conference on Engineering and Technology,* December 2013, Atlantis Press: Amsterdam, 2013; 27–32. DOI: to be assigned soon.
15. Wang X, Karri R. NumChecker: detecting kernel control-flow modifying rootkits by using hardware performance counters. *Proceedings of the 50th Annual Design Automation Conference (DAC),* May 2013, ACM: New York, 2013; 79. DOI: 10.1145/2463209.2488831.
16. Dinaburg A, Royal P, Sharif M, Lee W. Ether: malware analysis via hardware virtualization extensions. *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS),* October 2008, ACM: New York, 2008; 51–62. DOI: 10.1145/14557701455779.
17. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SSOP),* December 2003, ACM: New York, 2003; 164–177. DOI: 10.1145/945445.945462.
18. Berger S, Cáceres R, Goldman K A, Perez R, Sailer R, Doorn L V. vTPM: virtualizing the trusted platform module. *Proceedings of the 15th USENIX Security Symposium (SSYM),* August 2006, USENIX Association: Berkeley, 2006; 21–21.
19. Griffin JL, Jaeger T, Perez R, Sailer R, Doorn LV, Cáceres R. Trusted virtual domains: toward secure distributed services. *Proceedings of the 1st Conference on Hot Topics in System Dependability (HotDep),* June 2005, USENIX Association: Berkeley, 2005; 4–4.
20. Catuogno L, Löhr H, Manulis M, Sadeghi AR, Stüble C, Winandy M. Trusted virtual domains: color your network. *Datenschutz und Datensicherheit DuD* 2010; **34**(5):289–294. DOI: 10.1007/s11623-010-0089-0.
21. Catuogno L, Dmitrienko A, Eriksson K, Kuhlmann D, Ramunno G, Sadeghi AR, Schulz S, Schunter M, Winandy M, Zhan J. Trusted virtual domains - design, implementation and lessons learned. *Lecture Notes in Computer Science* 2010; **6163**:156–179. DOI: 10.1007/978-3-642-14597-1_10.
22. Berger S, Cáceres R, Pendarakis D, Sailer R, Valdez E. TVDc: managing security in the trusted virtual datacenter. *ACM SIGOPS Operating Systems Review* 2008; **42**(1):40–47. DOI: 10.1145/1341312.1341321.
23. McCune JM, Jaeger T, Berger S, Cáceres R, Sailer R. Shamon: a system for distributed mandatory access control. *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC),* December 2006, IEEE Computer Society: Los Alamitos, 2006; 23–32. DOI: 10.1109/ACSAC.2006.47.
24. Payne BD, de Carbone MDP, Lee W. Secure and flexible monitoring of virtual machines. *Proceeding of the 23rd Computer Security Applications Conference (ACSAC),* December 2007, IEEE Computer Society: Los Alamitos, 2007; 385–397. DOI: 10.1109/ACSAC.2007.10.
25. Wu X, Gao Y, Tian X, Song Y, Guo B, Feng B, Sun Y. SecMon: a secure introspection framework for hardware virtualization. *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing,* Febrary 2013, IEEE Computer Society: Los Alamitos, 2013; 282–286. DOI: 10.1109/PDP.2013.48.
26. Trusted Computing Group. *TCG Specification Architecture Overview, Specification Revision 1.4*. (Available from: https://www.trustedcomputinggroup.org/resources/tcg_architecture_overview_version_14) [Accessed on 1 July 2013].
27. *Strace*. (Available from: http://sourceforge.net/projects/strace/) [Accessed on 1 July 2013].
28. Frédéric B, Olivier F. *Syscall Interception in Xen Hypervisor*. (Available from: http://hal.archives-ouvertes.fr/docs/00/43/10/31/PDF/Technical_Report_Syscall_Interception.pdf) [Accessed on 1 July 2013].
29. Tamberi F, Maggiari D, Sgandurra D, Baiardi F. Semantics-driven introspection in a virtual environment. *IEEE Computer Society: Los Alamitos*, 2008; 299–302. DOI: 10.1109/IAS.2008.17.
30. Baiardi F, Maggiari D, Sgandurra D, Tamberi F. Transparent process monitoring in a virtual environment. *Electronic Notes in Theoretical Computer Science* 2009; **236**:85–100. DOI: 10.1016/j.entcs.2009.03.016.
31. Baiardia F, Sgandurra D. Attestation of integrity of overlay networks. *Journal of Systems Architecture* 2011; **57**(4):463–473. DOI: 10.1016/j.sysarc.2010.06.001.
32. *XenTrace*. (Available from: http://www.xen.org/) [Accessed on 1 July 2013].
33. *libMicro*. (Available from: https://java.net/projects/libmicro/pages/Home /) [Accessed on 1 July 2013].
34. Hund R, Holz T, Freiling F C. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. *Proceedings of the 18th USENIX Security Symposium (SSYM),* August 2009, USENIX Association: Berkeley, 2009; 383–398.

35. Shacham H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS),* October 2007, ACM: New York, 2007; 552–561. DOI: 10.1145/1315245.1315313.
36. Davi L, Sadeghi A R, Winandy M. ROPdefender: a detection tool to defend against return-oriented programming attacks. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS),* March 2011, ACM: New York, 2011; 40–51. DOI: 10.1145/1966913.1966920.
37. Bahram S, Jiang X, Wang Z, Rhee J, Xu D. DKSM: subverting virtual machine introspection for fun and profit. *Proceedings of the 29th IEEE Symposium on Reliable Destributed Systems,* October 2008, IEEE Computer Society: Los Alamitos, 2008; 82–91. DOI: 10.1109/SRDS.2010.39.