Building Dynamic Integrity Protection for Multiple Independent Authorities in Virtualization-based Infrastructure

Ge Cheng[1], Hai Jin[1], Deqing Zou[1], Xinwen Zhang[2], Min Li[1], Chen Yu[1] and Guofu Xiang[1]
[1]Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, 430074, China
hjin@hust.edu.cn
[2]Samsung Information Systems America, San Jose, CA USA
xinwen.z@samsung.com

Abstract

In grid and cloud computing infrastructures, the integrity of a computing platform is a critical security requirement in order to provide secure and honest computing environments to service providers and resource consumers. However, due to the fact that software components running on a single platform are usually provided and maintained by different authorities which are potentially untrusted to each other, the problem to monitor and protect runtime system integrity become very challenging and has not been well addressed yet. In this paper, we present a virtualization based dynamic integrity protection method which ensures that only appropriate authorities can control over their components without interfering with other component providers or authorities. In our solution, integrity requirements defined by the authorities of upper components (e.g., service middleware and applications) are respected by preventing the underlying components (e.g., operating system) from exposing their sensitive data, which can be caused by update of the underlying components or other malicious actions. We implement our solution on Xen-based platform, and our evaluation results show that the solution is effective for integrity protection with acceptable performance overhead.

1. Introduction

In grid and cloud computing infrastructures, the operating system, service middleware, and applications deployed on a typical computing platform are provided and maintained by different authorities, which may not trust each other. For example, in a grid or cloud platform, there are normally three different software levels controlled by different authorities: (1) resource providers including the owners of computational nodes or other physical resources; (2) solution producers including the owners of software solutions and/or databases which are deployed on physical resources; and (3) users including the owners of applications running on a solution producer's product. The integrity of such platform can not only be threatened by malicious attacks, such as defects, Trojan horses, and viruses, but also by the update of components from multiple independent authorities. While the first type of threats is relatively easy to eliminate with such as widely-deployed anti-virus software, the latter becomes very challenging. Specifically, it mandates that after a component is updated, the authorities of other components still can trust the updated code and data, and consequently the behaviors of the updated component. From another point of view, a computing system in grid and cloud computing environments should be *trusted* by a remote client or user such that, for example, its declared quality of service is preserved, or user's valuable or privacy-sensitive data on clouds are protected from other entities including cloud service providers.

By "trusted" here we mean that a component is authentic: the integrity of its code and data is protected, it can only be updated by its authority, and then its behavior is predictable. In collaborative distributed systems, it is mandatory for a remote platform to provide its integrity status in a trustworthy way to others in order to detect and prevent applications from being deployed on untrusted even hostile platforms. For this purpose it is very important to verify if the platform is a known-good implementation and is running with a known-good configuration. Trusted Computing Group (TCG) has specified a small and low-cost Trusted Platform Module (TPM) [1] hardware component to enhance the security of desktop and portable computers. Various mechanisms have been developed to use such hardware to generate a proof of a system's integrity, such as remote attestation and authenticated boot [2][3]. Other approaches have been proposed to extend integrity measurement and verification up to application level [4][5][8]. These approaches provide a way to start with a small trusted computing base (TCB) including TPM and operating system (OS) kernel, and try to build a proof for the whole system by measuring each piece of software components according to the sequence of platform booting and application loading.

Unfortunately, previously proposed TCG-like integrity measurement and attestation mechanisms have some shortcomings which make them unpractical: first, traditional approaches lack the ability to protect sensitive information when a system's integrity is broken during runtime, which is only featured by some expensive trusted hardware such as IBM 4758 secure coprocessor [9]; secondly, by extending the integrity measurement and verification to application level, these approaches has a large TCB including whole OS [4][5], and they are not transparent to the OS and require modifications of OS kernel; and consequently, as a usual general-purpose OS is very large and complex, these approaches are frequently error-prone and vulnerable.

In our work we leverage the advantage of virtualization technology to address above problem. The authorities of the upper components are permitted to provide their protection strategies for their sensitive data. For example, in cloud computing, the application environment including operating system and other necessary middleware may be provided by solution producers (authority of underlying components). We should permit the users of this solution (authors of upper component) to protect their sensitive data. We ensure that secrets belonging to an authority are only accessed in appropriate environment that the authority trusts, through monitoring the changes of software components in virtual machines (VMs) or domains, and dynamically checking the integrity of the corresponding components according to the strategies defined by their authorities and controlling the access to the disk and memory. Our solution is transparent to platform OS as it is implemented in virtual machine monitor (VMM) layer.

In this paper, we present a formal security foundation for integrity requirements in multi-authority computing environments based on a trust dependency concept. We then illustrate our implementation which consists of modules for integrity measurement, monitoring, and access control in Xen. We leverage hardware-enhanced virtualization extensions to offer fast system call tracing and strong memory context protection.

The remainder of this paper is organized as follows. Section 2 presents related work on platform integrity protection. In section 3 we formally analyze the integrity protection requirements for the authorities of upper components on a platform, which builds the theoretical foundation of our work. We describe our implementation in Xen hypervisor in Section 4. Section 5 analyzes the effectiveness and runtime performance of our implementation. We conclude this paper in Section 6.

2. Related Work

The IBM 4758 security coprocessor [10] implements both secure boot and authenticated boot, albeit in a restricted environment. It promises secure boot guarantees by verifying all partitions of the whole system before activating them. By enforcing signature verification on executables before loading them into the system, it further protects the security of sensitive data when the predefined integrity of the system is broken. A mechanism called outgoing authentication [2] enables attestation that links each subsequent layer to its predecessor. Smith and Weingart [6] discussed how to ensure security deployment and the correctness of software updating, which relies on the security coprocessor in multi-authority environments. High design and development cost prevents it from being widely applied, and it's difficult to support large-scale commercial applications.

TCG has proposed an open interface for hardware TPM which provides cryptographic functionalities and protected storage [1]. TPM enables the verification of static platform configurations, both in terms of content and loading order, by collecting a sequence of hashes over target codes. Researchers have examined how to use TPM to prove that a platform has booted with a valid OS with trusted BIOS and OS loader [4][11].

IMA [4] is a scheme to extend the TCG specified measurements to programs in application layer. Donald et al. [5] discussed how to convert a business Linux system into trusted virtual computing platform with TPM and ensure its trusted environment through integrity check. However, both approaches need modifying OS, which limits their application, e.g., on Linux platforms. Another major issue of TCG and IMA is that they only have load-time integrity measurement thus no runtime integrity guarantee.

Terra [7] is a trusted computing architecture built on a trusted VMM that authenticates software running in a VM for challenging parties. Terra measures the trusted VMM on the partition block level. Thus, on one hand, Terra produces about 20 MB of measurement values (i.e., hashes) when attesting a 4 GB VM partition. On the other hand, it is difficult to interpret varying measurement values. Our system selectively measures those parts of a system that contribute to dynamic runtime system integrity; it does so on a high level that is rich in semantics and enables remote parties to interpret varying measurements on file level.

3. Formal Foundations

In this section, we analyze how to protect the sensitive data of upper components according to their integrity protection requirements, considering the existing of multiple independent authorities on a single platform. We start with a simple and abstract model for program execution and then present the basic concepts and principles related to trusted state. Our analysis in influenced by the outgoing authentication problem [2].

3.1. Program Dependency

Assumption A computing environment has exactly one memory place to hold software and the memory is untampered by outside. The computing environment cleans all memory states when it is restarted. We denote the environment which satisfies the above assumption as $\&_{CE}$.

Authority An authority can authorize updating or loading a program *p* in $\&_{CE}$.

As aforementioned, the computing environments we face force us to partition the code space in $\&_{CE}$ into three layers: OS layer, service provider layer, and application layer. Software in different layers within $\&_{CE}$ are typically controlled by different, mutually untrusted authorities. So we need to tolerate malicious authorities including those of OS and bootstrap. Under this scenario we consider a system state as the collection of the content of memory and CPU registers. The instructions and data can be affected by former loaded programs.

Entity A program *p*, including code and data, is loaded and executed inside computing environment $\&_{CE}$ at a particular moment.

A system state is not determined by one entity; on the contrary it is determined by the entities come from all the software levels. For this reason we need to explore what happens to a particular platform: not only long-term action sequences, but also specific instants along that sequence.

History and Run A history is a finite sequence of computations for a particular computing environment. A run is an unbounded sequence of computations for a particular computing environment. *H*←*R* means history *H* is a prefix of run *R*.

When a program *p* is loaded in run *R*, the system state is changed, and *R* becomes *R'*. So we can say entity *e* corresponds to a series of procedures which are loaded into the memory in a particular sequence. We refer to *S* as the system state, and at a particular moment the system state S_R can be denoted by the set of all entities which run in $\&_{CE}$, that is, $S_R = \{ e_1, e_2 \cdots e_n \}$. We note that *p* belongs to an authority, and the authority might authorize the computing environment $\&_{CE}$ to load *p* to change the state.

The system state is determined by an entity set in run *R*, and the entities interact with each other. However, the relationship between them is complex and some entities have the ability to read or write other entities.

Dependency Function Let *E* be the set of all entities in $\&_{CE}$, , for $\forall e_1, e_2 \in E$, if e_1 can read/write the data of e_2 , then $e_2 Dep_{data}(e_1)$; if e_1 can write/control the code of e_2 , then $e_2 Dep_{code}(e_1)$, where *Dep* represents the union of Dep_{data} and Dep_{code} on *E*.

Naturally, we have the following deduction.

$$\begin{cases} e_1 \in Dep(e_1) & Idempoten \\ if\ e_2 \in Dep(e_1), then Dep(e_2) \in Dep(e_1) & Transitive \end{cases}$$

Relation *Dep* depends on run R. Let $\xrightarrow{R}$ be the transitive closure of *Dep*. For entity *e* in run *R*, we define $Dep_R(e) = \{ f : e \xrightarrow{R} f \}$.

For entity *e* in run *R*, $Dep_R(e)$ lists all the entities in $\&_{CE}$ that can subvert the correct operations of entity *e* in run *R*. As mentioned above, an entity's action can be possibly damaged by other entities. We need some notion of trust. Usually, an authority *Au* has some ideas of which applications it might trust and of which ones it does not trust.

Trustset For an authority *Au*, let *Trustset (Au)* denote the set of entities that *Au* trusts.

3.2. Integrity Protection Requirement

We use *C* to denote a system configuration which consists of relevant properties, including a vector of conditions for each authority: its trust set, authority status, code contents, and protected data. A system state consists of a program running sequence and programs permitted to run in $\&_{CE}$. We denote this by $\&_{ENC}$.

Suppose *Au* is an application authority in a valid configuration *C*. For $\&_{ENC}0 \in Trustset(Au)$, $\&_{ENC}0$ denotes a state that the protected data of authority *Au* has its initial contents, but no program in $\&_{ENC}0$ writes to the protected data since these contents have been initialized. $\&_{ENC}i$ denotes an updated system state when program p_i is loaded.

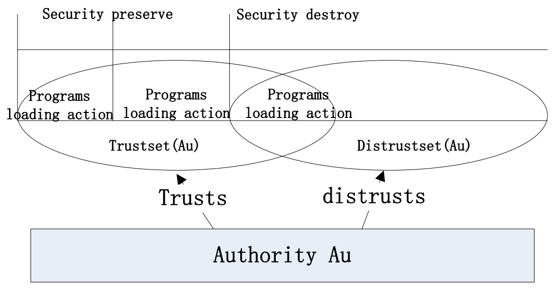

Fig. 1. An authority stops trusting a computing environment when a loaded program doesn't belong to trust set.

Let $p_0 \cdots, p_i \cdots$ be a valid program loading sequence, which have been loaded into a system in configuration *C*. If p_k $(0 \leqslant k \leqslant i)$ is the first program in this sequence such that $\&_{ENC}k \notin Trustset(Au)$ is true, then the contents of the protected data are destroyed or the system returns to $\&_{ENC}k-1$, as illustrated in Fig. 1. If this is satisfied, then only programs loaded before $\&_{ENC}\ k$ can directly access the protected data. In particular, *Au* may stop trusting a system state when transition from $\&_{ENC}\ k-1$ to $\&_{ENC}\ k$ includes a loading of any code in any underlying layer which *Au* does not trust.

3.3. Trust Validation

Since entity *e* interacts with other entities in the same $\&_{CE}$ which depends on $Dep_R(e)$, a desired integrity monitoring mechanism in VMM should determine the trust of $Dep_R(e)$.

The question is how to identify an entity and how to determine the changes of the entity from the virtual layer. It is very difficult to monitor the changes of the whole memory to achieve this. An alternative method is called as "load time integrate measurement" which identify an entity by checking the hash value of the corresponding program when the program is loading into the memory [3][4][9][13]. In this paper, we assume that code measurements are sufficient to describe the changes of an entity. Thus, self-changing code can be evaluated because the self-changing ability of code is reflected in the measurement and can be taken into account in verification.

Trust State For entity *e*, run *R* is trusted by authority *Au* only if $Dep_R(e) \subseteq Trustset(Au)$.

In order to determine whether $Dep_R(e) \subseteq Trustset(Au)$ after *p* is loaded into *R* and *R* transits into *R'*, the primary function of integrity monitoring in VMM is to trace the entity. We note that the entity is determined by the sequence of loaded programs. Let *trace (p, R, C)* denote the collection of loaded entity hash value which is provided by VMM when the authority *Au* loads *p* in run *R*.

Validating Trust State Validating a trust state is a mechanism that determines whether $\&_{CE}$ is trusted to authority *Au* when *p* is loaded in run *R*, according to *Trustset (Au)* and the collection of loaded entity credentials *Trace (p,R,C).*

The algorithm to validate a trust state is determined by the collection of loaded entity credentials *Trace (p,R,C)* and *Trustset (Au)*. Naturally, *Trustset (Au)* is associated with the application requirement of authority *Au*. Therefore for those entities trusted by *Au* they vary with *Au* with different selections.

Validation by VMM is reliable and complete, if and only if for any entity *e*, *Trustset (Au)*, and any history *H* and run *R* where $H \leftarrow R$, the following is true:

$$validate(Au, Trace(p,R,C)) \Leftrightarrow Dep_R(e) \subseteq trustset(Au)$$

4. Implementation

Our solution is built on hardware virtualization extensions such as Intel VT [14] and AMD SVM [15]. In this section, we discuss our implementation on Xen HVM DomU based on Intel VT. We first give an overview of our implementation, followed by the description of measurement and protection of sensitive data in disk by hooking disk I/O. We then show the mechanism to trace program loading and protect memory sensitive data. At the end of this section we describe how to validate the integrity of a system and make access control decisions.

4.1. Implementation Overview

According to the formal model described in previous section, in order to protect sensitive data specified by an authority when the integrity of its trust set is broken, we need to monitor the process of program loading, verify whether loaded programs belong to the corresponding authority's trust set, and examine the integrity of the loaded programs.

We leverage virtualization technology to fulfill the above requirements. As shown in Fig. 2, all measurement operations and access control of disk files are achieved by hooking disk I/O operations. Monitoring loaded programs and protecting sensitive data in memory are implemented by intercepting corresponding system calls.

We leverage Blktap architecture, X86 fast system call entry mechanism, and Xen memory management subsystem to achieve the measurement, monitoring and access control. Our implementation includes a set of functional modules: trace module (TAM), system call tracer (SCT), and decision-making engine (DME). TAM collects the information of disk operations and measures the trust set and controls accesses to the disk. SCT collects and filters system call arguments and provides memory protection. DME makes decision of measurements or access control according to information sent by TAM and SCT.

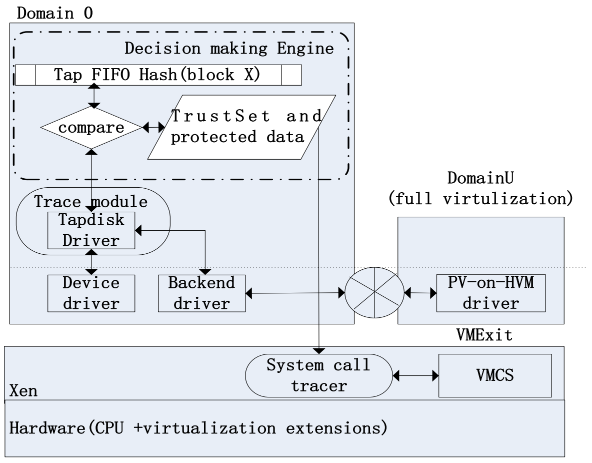

Fig. 2. VMM based integrity protection architecture.

4.2. Measurement and Disk Access Control

Blktap is a user-mode driver which directly manages disk activity with relatively small performance cost [12]. TAM intercepts file operations in the user-mode of Dom0 when a disk data is processed by the tapdisk driver of Blktap.

TAM does not block disk reading operations, but only send operation parameters (include the starting sector location and the number of sectors) to DME which makes access or measurement decisions according to these parameters. According to the decision from DME, TAM makes one the following three types of actions: (1) **measurement operation --** TAM copies the buffer of disk reading operations to the measurement buffer, or invokes Blktap asynchronous I/O operations according to the measurement parameters from DME to read the specified data to the measurement buffer. At the same time, TAM returns disk reading operations and invokes a hash function to take the measurement. When this function returns, TAM submits the hash value to DME; (2) **normal operation --** TAM does not take any action and the reading operation continues; (3) **deny operation --** TAM cleans the file buffer and returns a reading error.

For any disk writing operation, TAM blocks it, sends the arguments of the operation to DME, and then enforces DME decision: permits write operation or not.

4.3. Monitoring and Memory Access Control

Our implementation monitors system state at the following two stages:

Booting Stage In the booting stage a monitored VM, the function of BIOS is offered by the VMM but it does not usually directly load the OS. Instead, it only loads a portion of a boot loader residing in MBR into the memory and transfers the control to the loaded code. So TAM must measure the boot loader and the OS image. We achieve this by intercepting the disk data flow in the booting stage before it is processed by the tapdisk driver, and measuring all the data received by DME.

Runtime Stage During runtime, we dynamically monitor which programs are loaded and where the protected data are loaded into the memory. System calls are intercepted using X86 fast system call entry mechanism. X86 fast system call is generally used on Windows XP and Linux kernel 2.6. The *SYSENTER* instruction triggers the transition from the user mode to the kernel mode. The kernel entry address is specified by two special registers: *SYSENTER_CS_MSR* and *SYSENTER_EIP_MSR*. Whenever user-mode applications require system services, the service number and parameters are transferred into the kernel, and then the instruction *SYSENTER* executes.

In our implementation, the value of register *SYSENTER_EIP_MSR* is set to a magic address which leads to page fault every time by SCT. Whenever a system call in the monitored VM is invoked, a page fault occurs at the special address. When page fault occurs and the page fault linear address is equal to the magic address, it indicates that the system call has happened, and its parameters related to the current process are gathered to record reading/writing operations. Ultimately, the real entry address of a system call is set in the *EIP* register, and the handler executes in the monitored VM.

It is dispensable to inspect all system calls and their arguments. In fact, we focus on system calls for file operations and loading modules and applications, such as *read*, *write*, *int_module*, *execve*, and *fork*. Modules are dynamically loaded into kernel space through *insmod*. Applications replace the current execution code via the system call *execve*. The arguments of these system calls may include the relative pathname of executable files. The absolute pathname can be resolved according to the *task_struct* structure of the current process. After that, the pathname is transferred to DME.

If the protected data is loaded into the memory, accessing to the data is controlled by SCT. The most important thing is the data and the location of the data. Through intercepting *read* system call, we can obtain such information in real time. From the arguments of *read* system call, the file descriptor and buffer address are easily obtained. Similarly, the absolute pathname can be analyzed through the file descriptor and the *task_struct* structure of current process. This information is passed to DME.

To control the access to the protected data in the memory, we leverage Xen's shadow paging mechanism. This technique maintains two kinds of page tables for each VM: guest page tables (GPTs), which are controlled by the guest, and shadow page tables (SPTs), which are controlled by the hypervisor. Xen controls the actual machine frames used by each VM, while also provides each guest OS the illusion that it has full control of the memory. To achieve the memory access control for protected data, we need to control the propagation of entries from GPTs to SPTs and Xen's page fault handler.

We note that we can trace the page information that the protected data is stored with protected page tables (PPTs) created by SCT. SCT populates the PPTs with references to the physical pages corresponding to the linear address space of the protected data. Once the access requirement to the protected data in memory needs to be controlled, SCT removes references to the program's protected pages from the SPTs and flushes the TLBs. Due to this setup, access to code/data from the SPTs to the PPTs or vice versa leads to page-faults that invoke SCT in the hypervisor. This technique only provides page-level protection, which is problematic if a page contains protected and accessible regions at the same time. We need to provide byte-level protection by modifying Xen's page fault handler. Each time a page fault occurs due to a failed access operation, we check the target's virtual address, which is stored in the CR2 CPU register. Next, we check the protection list to see if the target address requires protection. If yes, then a page fault exception is propagated to the guest OS to prevent the access attempt; if not, then the guest attempts to access a non-protected region of a frame that contains a protected region.

4.4. Decision-making Engine

DME processes the life cycle of an authority's protected data according to information sent by TAM, SCT, and the trust set defined by the authority. DME supports the authority to describe its trust set and protected data in higher-level file system--oriented view, by specifying which directories and files belong to the trust set or which files are protected data, and 160 bit hash values are used to identify the integrity of these programs.

At the VMM layer, most of the operations captured by TAM and SCT are low-level operations, which are closely related to specific system architecture, while the trust set and protected data are described with higher level semantics. DME translates the easy-to-manage higher-level representations into a raw physical operation. Specific semantic information translation is closely related to guest OS and selected file systems. DME builds three structures called *trust_inte_file*, *prote_file* and *mem_pro_file* for this purpose. The first two record the translated results according to the authority's trust set and protection data, and the third records the memory address of the protected data. All the files and directories of the first two structures have a block node including all the blocks which the directories and the files have occupied.

When a target VM boots, DME firstly compares the hash value of the boot loader and OS kernel image with the values in *trust_inte_file*. If any of them does not match, it means the boot loader or Os kernel image is not satisfied with the authority's requirement. The access permit bit of the *prote_file* is set. After initialization, for each change DME reads a new record from the tap FIFO sent by TAM. Next, the record's block number is hashed into the *trust_inte_file* and *prote_file*. If the record's block number is found in the *prote_file*, it indicates that this disk I/O operation is accessing the protected data. DME then checks the access permitted bit. If the bit is set, which means the authority expected trusted environment is broken, the access requirement should be denied. DME sends a deny operation instruction to TAM. Otherwise, if the access permitted bit is not set, DME checks whether the measurement buffer is empty. Because loaded kernel modules and programs are measured asynchronously with file reading operations, DME must wait until all the measurements are finished. If the protected data is allowed to access, for a reading operation, DME sets an opening bit to indicate the protected data is to be loaded into memory. For a writing operation, DME records the block with the change of the protected data after the writing operation is completed.

If the block numbers are matched in *trust_inte_file*, which means the disk I/O operation is accessing the trust set. For a reading operation, if the file has never been measured before or has been changed, DME sends the measurement instruction (including the entire block this fill occupied) to TAM. The hash value is compared with that in *trust_inte_file*. If the hash value hits in *trust_inte_file*, it indicates that the loaded data and program are satisfied with the requirements of the authority. On the contrary, if it misses, the access permitted bit is set. DME also needs to verify whether the kernel modules and user-level executables come from trust set. According to the description in Section 4.3, TAM can capture the path information of them so DME can match the path information in *trust_inte_file*. If they do not match, the access permitted bit is set.

Then DME checks whether the opening bit is set. If it is set and the permit access bit is set too, it indicates that some protected data is load into the memory and at the same time the integrity of system is broken, DME then sends memory protected instruction to SCT.

In order to minimize performance impact, we take a new measurement only if a target file has not been measured or it might have been changed since last measurement in *trust_inte_file*. So we use caching to reduce performance overhead.

5. System Evaluation

In this section, we first demonstrate the integrity protection capability of our implementation, and then analyze its performance overhead.

5.1. Security

Due to space limit we present a simple experiment to demonstrate our system's capability of dynamically detecting the integrity change of programs defined in an authority's trustset and protecting the authority's sensitive data.

```
[root@localhost disk_monitor_edit5]# ./monitor
**********env_check(dom_id)**********

**********parse_config(dom_conf)**********
Image :/root/guest/vm-ubuntu.img
ea24521cbaf6febb9f0f06349a986508  /etc/init.d/rc
e3756487011471f7753d5d94ce4b6af4  /etc/init.d/rc.local
6687b5585f865da7e7875b1d9cfff4a0  /etc/passwd
b59ea6ac3a1ad8c0527ec94f73bafca0  /etc/profile
```

Fig. 3. Detecting the integrity change of programs.

As shown in Fig.3, when the integrity of the file */etc/profile* which is included in an authority's trust set does not match the hash value in the trust set which is defined by the ordinary user the authority. Our implementation detects this change and protects the sensitive data of the authority. The output is shown in Fig. 4.

```
[root@localhost-120 root]# ls -l
ls: cannot access protect.txt: Input/output error
total 0
-????????? ? ? ? ?            ? protect.txt
[root@localhost-120 root]# rm -rf protect.txt
rm: cannot remove `protect.txt': Input/output
error
[root@localhost-120 root]#
```

Fig. 4. Protecting sensitive data of an authority.

5.2. Performance

Our prototype system runs on a 2.33 GHz Intel Core Duo processor with 2 MB L2 cache, 2 GB RAM, and 80 GB 7200 RPM disk. The metrics include the latency of disk I/O and system call tracing. We use notation CHECK to represent the disk I/O with our design which needs time to make decision, while use UNCHECK to represent the case in a common Xen system without our design. M_HASH denotes file reading operation with the file's hash value measured. Measurements are made using the Linux time command. The script is executed in different file size. The size of sampled files varies from 1KB to 10MB for each mode. Table 1 presents the experiment results.

We test the performance of system call tracing by selected benchmarks which perform a standard series of tests provided by Linux web servers, database servers, and CPU-intensive applications. We also measure the efficiency of file compression and decompression using Linux kernel source. The size of linux-2.6.18.8.tar.gz is 58.6MB.

Table 1. The overhead of disk I/O (ms)

	1K	16K	128K	1M	10M
R_UNCHECK	7.7	17.8	107.8	862.4	10171.4
R_CHECK	8.7	21.8	139.6	864.2	12731.4
M_HASH	8.7	22.1	144.7	923.2	16322.2
W_UNCHECK	545.1	1173.2	5033.9	10580.2	16000.9
W_CHECK	547.2	1181.4	5097.9	10068.2	17623.9

Table 1 show that the Disk I/O performance overhead can be negligible. Most of Hash measurement is executed asynchronously with regard to actual disk I/O. The asynchrony created by the use of a FIFO allows DME, the most performance-intensive component of the architecture, to execute in parallel with actual disk operations.

Fig.5 presents the performance of system calls tracing. The results show that our implementation adds extra latency to system calls. Latency-sensitive benchmarks, such as web server benchmark, incur a relatively high performance cost. The latency is mainly raised by the notification of TAM. A full in-hypervisor implementation would have much lower latency. In addition, system calls which require I/O access are not affected by the extra latency in our current implementation.

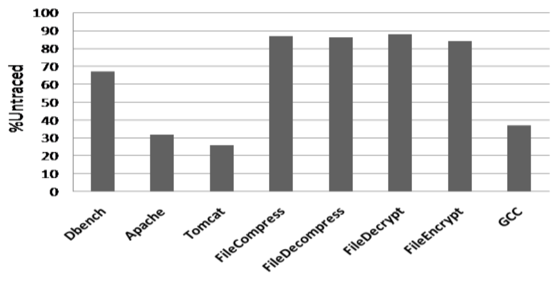

Fig. 5. Performance of system call tracing.

6. Conclusions and Future Work

In this paper, we have presented the design and implementation of a virtualization-based integrity protection approach which permits an authority to bind his sensitive data with integrity requirements. Our approach can guarantee that the sensitive data specified by an authority can only be accessed by programs in an environment that the authority trusts. This approach is applicable to multi-layer software environment where an authority of the upper software can maintain the security of the software when the integrity of underlying software components is broken. Experimental results show that the design is effective and the overhead is acceptable.

The main feature of our solution is that it can enhance the security of an ordinary commercial platform with the same capabilities provided by IBM4785 security coprocessor. Our solution not only measures and reports the integrity of a system, but also protects the sensitive data when the system's integrity is compromised. The approach can be applied in cloud or grid computing environments with multiple independent authorities to protect their sensitive data and to maintain the integrity of an entire system.

7. References

[1] Trusted Computing Group (TCG). https://www.trustedcomputinggroup.org/, 2003.

[2] M. Ceccato, M.D. Preda, J. Nagra, C. Collberg, P. Tonella, and S.W. Smith, Outbound authentication for programmable secure coprocessors, In: Proceedings of the 7th European Symposium on Research in Computer Security, London, UK, 2002, pages 72-89.

[3] H. Maruyama, F. Seliger, and N. Nagaratnam et al. Trusted platform on demand, Technical Report RT0564, IBM TRL, 2004.

[4] R. Sailer, X.L. Zhang, T. Jaeger, and L.V.Doorn, Design and implementation of TCG-based integrity measurement architecture, In: Proceedings of the 13th USENIX Security Symposium, August, 2004, San Diego, CA, USA, 2004, pages 223-238.

[5] R.M. Donald, S.W. Smith, J. Marchesini, and O. Wild, Bear: An Open-Source Virtual Secure Coprocessor based on TCPA, Technical Report TR 2003-471, Department of Computer Science, Dartmouth College, August 2003.

[6] S.W. Smith, and S. Weingart, Building a High Performance, Programmable, Secure Coprocessor, Computer Network, 1999, pages 831-860.

[7] T. Garfinkel, B. Pfaff, P.J. Chow, P.M. Rosenblum, and P.D. Boneh, Terra: a virtual machine-based platform for trusted computing, In: Proceedings of the nineteenth ACM symposium Operating systems principles, NY, USA 2003, pages 193-206.

[8] T. Jaeger, R. Sailer, and U. Shankar, PRIMA: policy reduced integrity measurement architecture, In: Proceedings of the eleventh ACM symposium on Access control models and technologies, NY, USA, 2006, pages 19-28.

[9] T.F. Lomac, Low water-mark integrity protection for cots environments, In: Proceedings of the 2000 IEEE Symposium on Security and Privacy, Washington, DC, USA, 2000, pages 230-245.

[10] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L.V. Doorn, S.W. Smith, and S. Weingart, Building the IBM 4758 Secure Coprocessor, IEEE Computer, 34(10), 2001, pages 57-66.

[11] J. Zhan, H.G. Zhang, Trusted Computing Enabled Access Control for Virtual Organizations, Computational Intelligence and Security Workshops, 2007, pages 490-493.

[12] Warfield, Virtually persistent data, In: Xen Developer's Summit (Fall 2006), 2006.

[13] P. England, B.W. Lampson, J. Manferdelli, M. Peinado, and B. Willman, A trusted open platform, IEEE Computer, 36(7), 2003.pages 55－62.

[14] Intel Virtualization Technology. http://www.intel.com /technology/virtualization.

[15] Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01 May 2005.