An Effective Auditing Scheme for Cloud Computing

Ryan Houlihan, Xiaojiang Du
Department of Computer and Information Sciences
Temple University
Philadelphia, PA 19122, USA
Email: {ryan.houlihan, dux}@temple.edu

Abstract— **In this paper, we present a novel secure auditing scheme for cloud computing systems. Several auditing schemes have been proposed for the cloud, which periodically trigger the auditing function. These schemes are designed to monitor the performance and behavior of the cloud. One major problem with these kind of schemes is that they are vulnerable to the transient attack (also known as the timed scrubbing attack). Our secure auditing scheme is able to prevent the transient attack via modification of the Linux auditing daemon - auditd, which creates attestable logs. Our scheme utilizes the System Management Mode (SMM) for integrity checks and the Trusted Platform Module (TPM) chip for attestable security. Specifically, we modify the auditing daemon protocol such that it records a hash of each audit log entry to the TPM's Platform Configuration Register (PCR), which gives us an attestable history of every command executed on the cloud server. We perform real experiments on two cloud servers and the results show that the overhead of our scheme is very small.**

***Keywords*-Cloud computing; performance; auditing**

I. Introduction

Virtualization of a machine allows us to run several operating systems at the same time on one system. Each operating system is run on its own Virtual Machine (VM) which allows for great flexibility. We can efficiently use the best "tool" for the job as well as allow for things such as testing and debugging to be done for multiple operating systems without constant rebooting. Cloud computers implement this virtualization technology and are becoming increasingly popular. A major problem that cloud providers are facing is how to provide better security to their clients.

These virtual machines are managed by a Virtual Machine Monitor (VMM) which is also known as a hypervisor. A hypervisor is a piece of software that manages the hardware for multiple VM's on a given system. It is the most privileged piece of software on the system and thus a breach in the integrity of the hypervisor results in a breach in the integrity of the entire system. Thus, hypervisors should be well-protected and contain a minimal code base. This, however, is not the case since hypervisors do have security flaws and hypervisor based security approaches are not sufficient to ensure the integrity of a system.

An example of this is Xen Hypervisor [12] which is used in systems such as Amazon's Elastic Compute Cloud (EC2) [16]. In recent attacks (eg. [1]) Xen [12] has been shown to be vulnerable to runtime attacks that allow data and code to be modified by malicious users. For cloud computers to become more viable, an attestiable hypervisor must be implemented to give possible clients peace of mind.

A. Related Work

Protection of the highest privileged software on a system is essential to provide integrity of the system. Many approaches attempt to integrate a higher privileged software level such as higher privilege hypervisors and micro-kernels. This is not an efficient solution to the problem since you once again must protect the new highest privileged software.

1) Hyperguard [1] and Hypercheck [2]: Both rely on the System Management Mode (SMM) which provides hardware protection for the integrity measurement code. Both frameworks alert the hypervisor before an integrity measurement leaving them vulnerable to the scrubbing attack where the hypervisor cleans up traces of an attack before the integrity measurement is started. Neither solve the technical problems associated with using SMM.

2) Copilot [3]: This scheme employs a PCI device which is used to poll the physical memory of the host and periodically send it to an administration station. A semantic gap between the code running on the PCI device and the running system on where PCI device resides. In result, Copilot cannot access the CPU state (CR3 Register). There are also existing attacks that can prevent copilot and any other PCI RAM acquisition tool from correctly accessing the physical memory using the hardware support of protected memory ranges. [5]

3) Flicker [4]: Employs a TPM based method to provide a minimum Trusted Code Base (TCB), which can be used to detect the modification of the kernel. It requires advanced hardware features such as Dynamic Root of Trust Measurement (DRTM) and late launch. The scheme is also directly vulnerable to the scrubbing attack because the measurement target is responsible for invoking the integrity measurement.

4) Hypersentry [6]: Relies on a TCB composed of hardware, firmware and a software component properly isolated from the highest privilege software. An out-of-band channel is used to invoke a System Management Interrupt (SMI) on the target platform to trigger Hypersentry. An Intelligent Platform Management Interface (IPMI) is used to establish this out-of-band channel. The integrity measurement agent resides in the SMM. The framework presents novel techniques to set the CPU to the required context and provide a verifiable and protected environment to run a measurement agent in the hypervisor context. Hypersentry cannot, however, handle

978-1-4673-0921-9/12/$31.00 ©2012 IEEE

transient attacks where the adversary may cause harm and then hide its traces.

B. Our Contributions

We present a framework using some features of Hypersentry [6] while providing a simpler and easier implementation. We contribute a more extensive framework for the requirements of the in-context integrity measurement code as well as preventative measures for transient attacks. This particular attack is a serious flaw of Hypersentry [6] and all other integrity verification tools that rely on periodic invocations (eg, [1]–[3]). A transient attack is defined by an attack in which a malicious user may cause harm, such as stealing data, and hiding its traces before an integrity measurement is run. Similar to Hyperguard [1], Hypercheck [2] and Hypersentry, [6] we use SMM as secure storage for our integrity checking software. Hypersentry [6] relies on the Trusted Platform Module (TPM) and hashes the SMI handler into one of the Trusted Platform Modules (TPM) Platform Configuration Registers (PCR) at initilization. We instead send all logs, encrypted with a distributed public key, to a local external computing system. Log verification is assured if the hash of the encrypted log and SMI Handler are equal to their stored encrypted hash. The message sent consists of the hash of the message itself, the encrypted log file, the log file hash signed by the TPM, and the hash of this SMI handler. These are all again encrypted with the nonce and public key to assure freshness. The nonce is sent to the cloud system from our external integrity checking system as an SMI is initialized. The encrypted logs are created as shown in *(Fig. 1)*. This helps to greatly reduce the overhead induced on the system and instead puts the work onto the external system. To prevent transient attacks we audit the entry into every call of sys_execve(), encrypt the audit info with the public key and extend a hash in the PCR register. This will prevent any of the log files from being modified without our knowledge due to a scrubbing attack before the integrity measurement can be run again. This keeps the system overhead to a minimum while offering maximum security benefits.

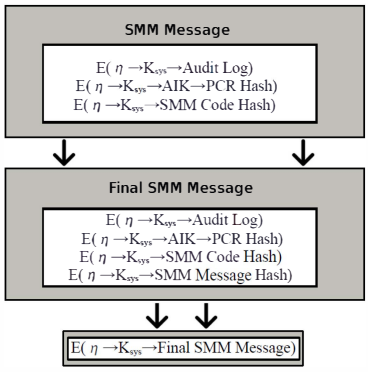

Fig. 1. Integrity measurement message

II. Threat and System Model

A. Threat Model

We focus on an efficient and stealthy measurement framework with out of context integrity checking. We focus on continuous integrity threats where our framework will detect persistent and non-persistent changes in the system. This effectively prevents "transient attacks" and "scrubbing attacks". Scrubbing is the process of a malicious user removing their traces from your system logs, effectively erasing their history.

B. Capabilities of Malicious Users

The malicious user can exploit any vulnerabilities in the system after bootup, including the Hypervisor, Xen [12], and all of its Virtual Machines (VMs). Arbitrary commands can be executed in Domain 0, the highest privileged level, via a grub.conf file [10]. Malicious users have the capability to modify code and data of Xen [12] by unauthorized DMA attacks [11]. Wotjczuk demonstrates such an attack by hijacking a network card to perform an unauthorized DMA to the Xen [12] hypervisors context. Once in this context, a malicious user can read, write, delete data and access the system log files such as Pacct and syslog.

C. System Model and Assumptions

We assume that our scheme is run on a system that supports numerous capabilities. This includes an out-of-band channel to remotely trigger an SMI via the General Input Port 0 (GPI0) which in our case is an embedded micro-controller, namely the Baseboard Management Controller (BMC). We also assume that the system is equipped with TCG's Trusted Platform Module (TPM) [13] in order to provide secure boot via the Core Root of Trust for Measurement (CRTM) as well as secure storage for the log file hashes via the PCR. The CRTM is an extension of the BIOS which will be initalized first, measure parts of the BIOS block, and then pass control back over to the BIOS. Once the BIOS, bootloader, and OS kernel run and pass control to the OS, the expected configuration by examining the TPM's Platform Configuration Register (PCR). Any change to the code between CRTM and the OS running will result in an unseen PCR value. The SMRAM is to be properly setup by the BIOS at boot time and to remain tamper-proof from cache poisoning attacks as in [7], [8]. To prevent these attacks, proper hardware configurations, such as System Management Range Register (SMRR) [9], should be used.

III. The Auditing Scheme

The following is our continuous auditing scheme for cloud computers. There are two requirements we must fulfill for an attestable system. First, our scheme must log every execution of the system before the execution can take place. Second, we must securely store a history of these logs to provide attestability of our system. Please see *(Fig. 2)* to examine the summary of our schemes execution cycle.

A. Pre-Filled Requirements

Secure invocation of our integrity code into SMRAM is provided by HyperSentry [6] framework. The first General Purpose Input port (GPI 0) provides us with an architecture to invoke a System Management Interrupt (SMI) as well as run our integrity check initialized by our external system

through an out-of-band channel as described in HyperSentry [6]. We use an Intelligent Platform Management Interface (IPMI) to communicate with the BMC but any out-of-band channel will work. By clearing the EFLAGS register and modifying the Interrupt Descriptor Table (IDT) we can insure a non interrupted System Management Mode (SMM) from both Maskable and Non-Maskable Interrupts (NMI).

PCR's have the requirement that they can only be extended and not over written. Thus the only way to modify a PCR is by the following TPM operation:

PCRExtend(*index, data*)

When PCRExtend is invoked on the TPM it updates the PCR with the SHA-1 hash of the previous value of the PCR concatenated with the *data* provided, where *data* must be a 20 byte hash. The TPM, thus, performs the following update:

SHA-1: PCR := SHA-1(PCR + data)

This operation provides assurance that no malicious user can modify the contents of the PCR to pass our authentication test, providing tamper proof evidence of a scrubbing attack. Linux provides an auditing system whose user space component, auditd(8) [15] the auditing daemon, can be used to audit all aspects of the kernels life-cycle. The /etc/audit/audit.rules contains a set of rules loaded into the kernels audit system. Through modification of this file we can have extensive auditing control over every system call. Using the *(entry,always)* option we are able to audit the entry into any system call before the actual system call happens. When any program is executed in the system it initially makes a call to:

```
long sys_execve(const char __user *,
        const char __user *const __user *,
        const char __user const __user *,
        struct pt_regs *)
```

This inturn calls:

```
extern int do_execve(const char *,
        const char __user * const __user *,
        const char __user * const __user *,
        struct pt_regs *)
```

which is responsible for the actual execution of exec(). However, before *sys_execve()* actually runs *do_execve()* a call to

```
char *getname(const char __user * filename)
```

is made which inturn calls:

```
void __audit_getname(const char *name)
```

The call to *__audit_getname()* adds a name to the list of audit names for the given context. Thus, by using the auditd(8) [15] daemon, we fulfill our requirement to log every execution of the system before the execution can take place. This in itself does not yet fulfill the requirement for our system to be tamper evident.

B. Attestable Auditing

To provide tamper evident logging a modification to the *__audit_getname()* function is required. This function, as previously mentioned, is executed before the corresponding command is issued auditing several aspects of the given command. In modifying *__audit_getname()* we can fulfill our requirement for a tamper evident system. We found two very similar approaches to this problem, one adds a little more execution time but added security and one with no added execution time and less security. Even so, both are more then secure enough to prevent even the most adamant attacker from performing a transient attack.

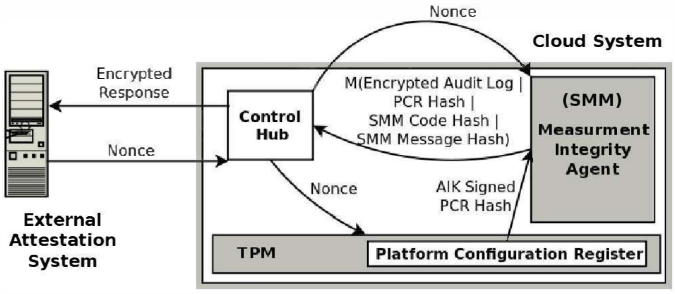

Fig. 2. Attestation process of our scheme which uses a remote integrity verification system to assure the integrity of a target system

Both approaches require that an audit context entry must be created and securely stored to provide assurance that the logs were not modified in anyway. For systems with predictable execution cycles, both approaches require that at secure boot a randomized entry is added to the beginning of the auditing log. This entries hash is then added to the PCR. This additonal requierment prevents a malicious user from erasing the entirety of the log file and replacing it with a predicted system execution and then rehasing the entirity of the PCR to match this new value.

In our first approach we encrypt the aforementioned audit log entry with the kernels public key K_{sys} to create a encrypted audit context entry $K_{sys}(audit)$. The corresponding private key K_{sys}^{-1} is stored on the external attestation agent and thus unaccessible to anyone who breeches the security of the cloud system. The encrypted log entry $K_{sys}(audit)$ will be added to the audit name list as usual. The data must then be extended into the PCR. As previously mentioned, any data to be extended must be 20 bytes, which would exceed the size of the encrypted audit entry $K_{sys}(audit)$.

Therefore, we must take a hash of the encrypted audit context using SHA-1 which would produce $H_{sha-1}[K_{sys}(audit)]$ which is 20 bytes long. Our encrypted audit entry hash is then extended to the PCR register and the encrypted audit entry is scheduled to be appended to the auditing log at the systems convenience. For consistency with our PCR stored hash the audit entries must be scheduled and added in a definite order, namely the order in which there corresponding hashes were extended to the PCR. The encrypted audit entry does not have to be added to the log immediately because even if the malicious user prevents the entry from being appended to the log file we will, in our PCR register, have a history that there is a missing entry. This also saves us from unnecessary system overhead invoked by our scheme.

Our second approach is very similar to the first approach except for one minor detail; we delay the encryption of the audit log entry. We create a SHA-1 hash of the raw audit entry, $H_{sha-1}(audit)$, and then extend this hash to the PCR register. Once the extension is complete we schedule the audit entry

to be encrypted and appended to the log entry. This must, as before, be done in order to provide consistency. As in the first approach if a malicious user prevents the addition of the log entry to the log file we will have, in the PCR register, a history that there is a missing entry.

Using the security and attestability of the PCR register and the existing Linux auditing daemon we have designed a secure auditing protocol that gives us attestation to each auditing log entry. For the remainder of the paper we will assume the first approach has been implemented. Regardless, all aspects of the scheme can be easily modified to work for the second approach.

C. SMM Measurement Message

After an SMI is generated using an out-of-band channel the system enters SMM which runs the measurement agent stored in the SMRAM. The integrity measurement agents responsibility is to deliver a encrypted message, E[M_{smm}], to the external integrity attestation system. In SMM the measurement agent retrieves the following four items: $K_{sys}(Log)$, $H_{sha-1}[K_{sys}(Log)]$, η, $H_{sha-1}(SMM)$. The public key, K_{sys}, encrypted log file $K_{sys}(Log) = \sum_{i=\tau_{prev}}^{\tau} K_{sys}(audit)_i$ where τ is the total number of audit entries and τ_{prev} is the total number of audit entries as of the previous integrity verification. This means we only copy over new additions to the log file since the last auditing session. This saves us valuable system overhead with each integrity verification since we already have the info before τ_{prev} on the external attestation agent and thus there is no need to copy it over again. If it was modified, our integrity check will alert us since we still have the hash of all the each auditing entries, $H_{sha-1}[K_{sys}(audit)]$, for verification. $H_{sha-1}[K_{sys}(Log)]$ is the resulting hash stored in the PCR which is the sum of the extensions of all of the $H_{sha-1}[K_{sys}(audit)]$. Before the hash is transferred to the SMM context the TPM signs the register with its Attestation Integrity Key (AIK) and the nonce, η, sent by the External Integrity Attestation System. The first encryption is used to acknowledge the integrity of the contents of the register, the second to guarantee freshness of the response from the TPM. The nonce, η, is also sent by the External Integrity Attestation System to the SMM context to guarantee freshness of the measurements response. Finally, $H_{sha-1}(SMM)$, is the hash of the measurement code which is executed by the SMI and stored in the SMRAM itself.

The measurement agent encrypts $H_{sha-1}(SMM)$ with K_{sys} and η to create E[$H_{sha-1}(SMM)$]. $K_{sys}(Log)$ and $H_{sha-1}[K_{sys}(Log)]$ are also encrypted with η to form E[$K_{sys}(Log)$] and E[H[$K_{sys}(Log)$]]. The SMM message is then constructed as $M = \{E[K_{sys}(Log)] \mid E[H[K_{sys}(Log)]] \mid E[H_{sha-1}(SMM)]\}$. A hash of M is then taken, called $H_{sha-1}(M)$, and appended to the end of M creating a new message M_{SMM}. The final message is $M_{SMM} = \{E[K_{sys}(Log)] \mid E[H[K_{sys}(Log)]] \mid E[H_{sha-1}(SMM)] \mid H_{sha-1}(M)\}$. This is encrypted with K_{sys} and η to form E[M_{SMM}] which is our final encrypted message. This message is now ready to be sent to the External Integrity Attestation System for verification.

D. Attesting to Measurement Output

On the External Integrity Attestation System we have stored the encrypted system logs up to $K_{sys}(Log)_{\tau_{prev}}$, the hash of the audit entries up to $H_{sha-1}[K_{sys}(Log)]_{\tau_{prev}}$, and the hash of the measurement agent code $H_{sha-1}(SMM)$. The stored system log is represented as $K_{sys}(Log)_{Stored}$, the stored hash is represented as $H[K_{sys}(Log)]_{Stored}$ and the stored measurement code hash is $H(SMM)_{Expected}$. These are used for attesting to the information received from the cloud system along with the SMM measurement message M_{SMM}. The authentication system receives E[M_{SMM}] from the cloud and proceeds to decrypt the message using the private key, K_{sys}^{-1}, as well as the nonce, η, which results in M_{SMM}. Once M_{SMM} is obtained we break the message back up into two parts, $H_{sha-1}(M)$ and M.

The hash of the message, $H(M)_{Actual}$, is calculated and compared with the received hash $H_{sha-1}(M)$. If $H_{sha-1}(M)$!= $H(M)_{Actual}$, the message was tampered with and we have a security breach. If the message's integrity is assured we proceed to decrypt the remaining three items E[$K_{sys}(Log)$], E[H[$K_{sys}(Log)$]], E[$H_{sha-1}(SMM)$]. First E[$H_{sha-1}(SMM)$] is decrypted using K_{sys}^{-1}. We then check for freshness by decrypting with η and check for integrity assurance of the TPM by decrypting with the AIK which produces $H_{sha-1}(SMM)$. Next, we compare $H_{sha-1}(SMM)$ to the expected value of the measurement code hash, $H(SMM)_{Expected}$. If they are equivalent, this verifies the SMM measurement code has not been altered by an SMM attack.

After the message, M, and the measurement agent on the cloud system are verified, the rest of the attestation process can take place. E[$K_{sys}(Log)$] and E[H[$K_{sys}(Log)$]] are both decrypted with η to form $K_{sys}(Log)$ and H[$K_{sys}(Log)$]. We must update $K_{sys}(Log)_{Stored}$ with the new log entries by appending $K_{sys}(Log)$ to $K_{sys}(Log)_{Stored}$. Now for $\tau - \tau_{prev}$ $K_{sys}(audit)$ added to $K_{sys}(Log)_{Stored}$ from $K_{sys}(Log)$ we hash these $K_{sys}(audit)$ into our stored log hash, $H[K_{sys}(audit)]_{Stored}$. If the newly computed $H[K_{sys}(audit)]_{Stored}$ is equivalent to the H[$K_{sys}(Log)$] transferred in the measurement message then we are assured that our systems integrity is not compromised. If the opposite is true, we are alerted that undefined and potentially malicious behavior has occurred and the appropriate damage control measures can put into place.

IV. Implementation and Performance Evaluation

Our experiment consisted of three distinct parts to approximate the system overhead invoked with this framework. We tested the time it takes to sign the auditing output with the public key stored by the kernel, the time it takes to create a 20 byte hash of the public key signed auditing output, and the time it takes to extend the 20 byte hash to a PCR register. This will give us a good idea of the system time overhead invoked by our scheme since these are the only modifications done to the kernel itself. They will also be executed at the

Command	Execution Time in Ticks
cat /var/log/messages	1.46E+007
netstat -tupl	2.57E+008
uname -a	1.33E+007
ls -la	2.21E+007
ps	6.82E+007
PCR Extend Execution	3.15E+007
SHA-1	2.92E+004
512 Bit RSA	1.12E+007

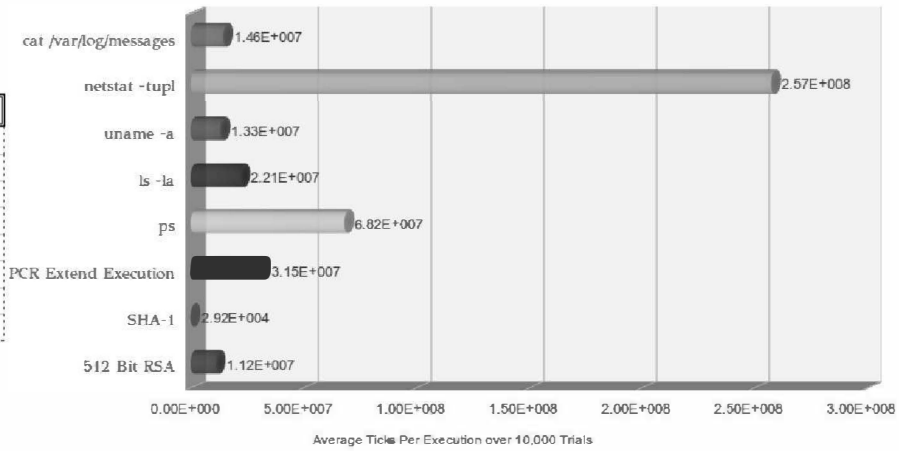

Fig. 3. Secure logging of all calls to execve. The results are measured in ticks over 10,000 tests whereas ticks are machine-dependent cycle counters [20]. All stages of the PCR extension are combined to give the execution time of the overall extension process.

beginning of any execution by all users at any privilege level, thus invoking the most overhead.

A. Testing Setup

All our experiments were run on a Dell PowerEdge T410. This system is equipped with 32 GB of DDR3 RAM running at 1066 MHz, 2 Quad core Intel Xeon E5620 processors clocked to 2.40 GHz and three 150 GB Western Digital Raptor hard drives running at 10,000 RPM. The system runs CentOS 5.7 x86_64 with the Xen Linux kernel 2.6.18-274.17.1. We used TrouSerS TSS 1.1 [19] open-source Trusted Computing Group (TCG) Software Stack to implement the PCR extension process.

First our program took in audit log entries replicating files that would saved by the auditing system before the invocation of a call to execve(). This line is then encrypted with our generated public key using a 512 bit RSA encryption algorithm [18]. Once the 512 bit encrypted auditing entry is produced we create a 20 byte hash using the SHA-1 hashing algorithm [14]. This 20 byte hash is then extended to the PCR register, by *PCR := SHA-1(PCR + data)*, where the *data* is our the auditing log hash. The PCR extension consists of four parts. Initially we must create the Trusted Service Provider Interface (TSPI) context. After this context is created we must open a connection with the context. Once a connection is initialized with the TSPI context we can call on it to initialize a TPM object. The TPM object has functionality which allows us to extend to a PCR of our choosing.

This experiment sufficiently replicates the added execution overhead during the execution of any call to *sys_execve()*. To complete our analysis we benchmarked the time to execute various shell commands such as *ls* which calls *execve("/bin/ls", ["ls"], [/* 39 vars */])* and *cat* which calls *execve("/bin/cat", ["cat"], [/* 39 vars */])*. The overhead of our scheme was then compared to the total execution time of these shell commands. This resulting value is a good approximation of the total overhead invoked on the system. Benchmarking of many other aspects of our framework was roughly performed by Hypersentry [6]. They found the system overhead for an auditing framework based off of SMM integrity measurement to be in an acceptable range of 2.4% system overhead if an integrity check is done every 8 seconds, and 1.3% if invoked every 16 seconds.

B. Overhead of Auditing

As shown in *(Fig. 3)* the entire PCR extension process adds about a 237% overhead to a call to *uname -a*, or 140% overhead on a call to *ls -la*. For calls like *ps*, it only invokes 46% overhead and for something like *netstat -tupl*, it only invokes a 12.3 % overhead. Typically on cloud systems large computations are done and the overhead invoked on something like *netstat -tupl* is more reasonable to the length of jobs run daily on a cloud system. The 512 Bit RSA encryption takes about 84% as long as a call to *uname -a*, 51% as long as a call to *ls -la*, 16% overhead on *ps*, and only about 4.4% overhead on a call to *netstat -tupl*. The SHA-1 hash algorithms' contribution to the overhead is negligible with a run length of 0.014% of that of *netstat -tupl*.

The total execution time of the PCR extension process is on average 3.15×10^7 ticks. As shown in *(Fig. 3)* creating the initial TSPI context takes 0.187% of the total execution time and is thus insignificant to the overall overhead. Connecting to the TSPI context took quite a bit longer and averaged at 9.62% of the total extension process. Creating the TPM object took the least amount of time at only 0.021% of the total overhead of the extension process. The PCR extension itself took by far the most amount of time in regards to the overall extension process. It took on average, 90.16% of the overall PCR extension process just to write to the PCR register. Thus we saw no ways of improving the implementation speed unless the PCR extension itself was further optimized.

We can combine the PCR extension process, SHA-1, and the 512 bit RSA encryption to get a good idea of the total overhead invoked by the system. The PCR extension process takes 73.7% of the total execution time, the SHA-1 takes only 0.068%, and the 512 bit RSA encryption takes 26.2%. Encryption with the 512 Bit RSA can also be eliminated entirely

Command	Execution Time in Ticks
cat /var/log/messages	1.46E+007
netstat -tupl	2.57E+008
uname -a	1.33E+007
ls -la	2.21E+007
ps	6.82E+007
TSPI Context Get TPM Object	6.59E+003
TSPI Context Connect	3.03E+006
TSPI Create Context	5.88E+004
PCR Extend Total	3.15E+007
SHA-1	2.92E+004
512 Bit RSA	1.12E+007

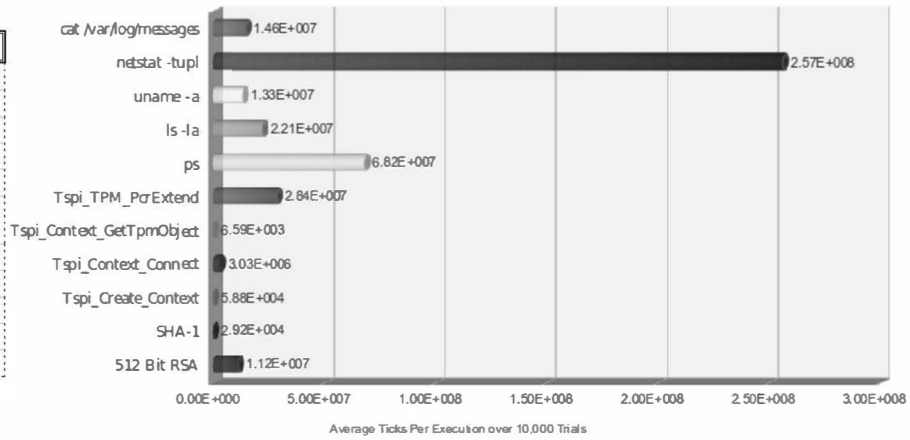

Fig. 4. Secure logging of all calls to execve. The results are measured in ticks over 10,000 tests whereas ticks are machine-dependent cycle counters [20]. Each part of the PCR extension is individually broken up. The PCR extend function itself produces the most overhead for the PCR extension process.

without any negative contribution to the overall security of the system and also save us 26.2% overhead.

This occurs beacause an attacker who enters our system and wants to run a transient attack would first have to erase their history from the auditing logs. They would then build a hash that when extended to the PCR register, produces a hash that would correspond to our auditing logs after the removal of their entries. A typical interval between attestation checks would be approximately 8 seconds to 16 seconds, thus giving any attacker only that amount of time to rehash the PCR before their malicious actions would be detected and the history of their attacks moved onto the external attestation agent. It is currently unfeasible for an attacker to calculate the appropriate hash the PCR should contain after the removal of their attack history and then rehash the PCR to reflect this value in the allotted time. Further, the audit entries might not be stored before the users code will be run, thus making their produced hash incorrect. The fact that a malicious user does not have access to the private key, K_{sys}, makes it impossible for this user to create a viable hash and thus unable to use the real log entries to create a new hash.

V. Conclusion

Due to an increased interest in the use of cloud computing, providing accountability to the corporate clients has become a critical component of the value proposition offered by cloud providers. In this paper, we presented an effective scheme that provides fully attestable auditing for cloud computing system. Different from the existing auditing schemes, our scheme is capable of preventing the transient attack. We achieved this by modifying the existing Linux auditing daemon as well as making use of existing software and hardware. Our scheme can provide clients with greater assurance and trust in cloud computing services. We performed real experiments on two servers, and the results showed that the overhead of our scheme is small.

Acknowledgment

This research was supported in part by the US National Science Foundation (NSF) under grants CNS-0963578, CNS-1002974, CNS-1022552, and CNS-1065444, as well as the US Army Research Office under grant W911NF-08-1-0334.

References

[1] R. Wojtczuk and J. Rutkowska. "Xen 0wning trilogy". *Proc. Black Hat conference*, 2008.
[2] J. Wang, A. Stavrou, and A. K. Ghosh. "HyperCheck: A hardware-assisted integrity monitor." *Proc. of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID10)*, September 2010.
[3] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. "Copilot - a coprocessor-based kernel runtime integrity monitor." *Proc. of the 13th USENIX Security Symposium*, p 13, 2004.
[4] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. "Flicker: an execution infrastructure for TCB minimization." *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, March/April 2008.
[5] J. Rutkowska. "Beyond the CPU: Defeating Hardware Based RAM Acquisition Tools." *Blackhat*, February 2007.
[6] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, N. C. Skalsky. "HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity." *Proc. of the 17th ACM Conference on Computer and Communications Security*, pp. 38-49, 2010.
[7] Duflot "Getting into the SMRAM: SMM reloaded" *Proc. of the 10th CanSecWest conference*, 2009.
[8] R. Wojtczuk and J. Rutkowska. "Attacking SMM memory via Intel CPU cache poisoning." Invisible Things Lab, 2009.
[9] I. Corporation. Software developer's manual vol. 3: System programming guide, June 2009.
[10] MITRE. Cve-2007-4993.
[11] R. Wojtczuk. "Subverting the Xen hypervisor." Invisible Thing Labs, 2008.
[12] P. Barham, B. Dargovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. *Proc. 19th ACM Symposium on Operating Systems Principles*, SOSP 2003, Bolton Landing, USA, October 2003.
[13] Trusted Computing Group. TPM specifications version 1.2. https://www.trustedcomputinggroup.org/downloads/specifications/tpm, July 2005.
[14] Department of Commerce National Institute of Standards and Technology. *Secure Hash Signature Standard (SHS) (FIPS PUB 180-2)*. February 2004
[15] SUSE. *The Linux Audit Framework*. Novell, 2008. Avaliable: http://www.suse.com
[16] Amazon Elastic Compute Cloud (EC2). http://aws.amazon. com/ec2
[17] PolarSSL. Offspark, 2011. Avaliable: http://polarssl.org/source_code
[18] RSA Lab. *PKCS #1 v2.1: RSA Cryptography Standard.* June 2002.
[19] TrouSerS Open-Source TCG Software Stack. http://trousers.sourceforge.net/
[20] M. Frigo, S. G. Johnson. FFTW Version 3.3. http://fftw.org