Satem: Trusted Service Code Execution across Transactions *

Gang Xu[1,3], Cristian Borcea[2], and Liviu Iftode[1]
[1] *Department of Computer Science, Rutgers University, USA*
[2] *Department of Computer Science, New Jersey Institute of Technology, USA*
[3] *AT&T Labs, New Jersey, USA*
gxu@cs.rutgers.edu, borcea@cs.njit.edu, iftode@cs.rutgers.edu

Abstract

Web services and service oriented architectures are becoming the de facto standard for Internet computing. A main problem faced by users of such services is how to ensure that the service code is trusted. While methods that guarantee trusted service code execution before starting a client-service transaction exist, there is no solution for extending this assurance to the entire lifetime of the transaction. This paper presents Satem, a **S***ervice-***a***ware* **t***rusted* **e***xecution* **m***onitor that guarantees the trustworthiness of the service code across a whole transaction. The Satem architecture consists of an execution monitor residing in the operating system kernel on the service provider platform, a trust evaluator on the client platform, and a service commitment protocol. During this protocol, executed before every transaction, the client requests and verifies against its local policy a commitment from the service platform that promises trusted code execution. Subsequently, the monitor enforces this commitment for the duration of the transaction. To initialize the trust on the monitor, we use the Trusted Platform Module specified by the Trusted Computing Group. We implemented Satem under the Linux 2.6.12 kernel and tested it for a web service and DNS. The experimental results demonstrate that Satem does not incur significant overhead to the protected services and does not impact the unprotected services.*

1 Introduction

Users of Internet services [1, 2] are increasingly worried about the service trustworthiness. Verifying the authenticity of the server (e.g., through server certificate authentication) is no longer sufficient to convince the user that she is interacting with the right service. This is because the service software may be tampered with. As a result, before she starts a transaction with the service, the user wants a guarantee that the service will execute only trusted code during this transaction. Recently, several methods [19, 28, 10, 5, 18, 20] based on software attestation [15, 9] have been proposed to ensure code genuineness and integrity on untrusted hosts. However, they are insufficient to achieve trusted service code execution for the entire lifetime of a client-service transaction for two reasons.

First, in order to start a transaction, the user demands a guarantee that the service will only execute trusted code. Methods such as [20, 14] can be used to attest the service to the user before the transaction starts and prove the trustworthiness of the service. They do not provide, however, any guarantee that the trustworthiness will persist after the attestation (i.e., during the transaction). BIND [26] addresses this problem by protecting the code being attested in a sandbox, which guarantees that this code cannot be tampered with during its execution. Under this mechanism, however, it is impossible to guarantee a priori that the service will run only trusted code. BIND can only prove that an output was produced by trusted code after the execution was complete. This is a significant problem for any transaction that involves confidential user data or critical irrevocable missions because untrusted code can handle them before being deemed untrusted.

Second, the existing methods lack the capability of precisely measuring and protecting the integrity of the service code base, which consists of all the programs loaded by the service at runtime. For instance, Sailer et al. [20] attest all files on the service platform. This approach is subject to *false positives* (the established trust has to be revoked even if a single bit is changed in a file irrelevant to the service). BIND alleviates this problem by letting programmers define the piece of code to be attested. Due to the dynamism and complexity of software execution, this method can only be selectively applied to critical pieces of code rather than the entire service code base. Terra [14] can run the service in a dedicated virtual machine and attest the virtual machine as a whole. On one hand, this avoids the false positives problem.

*This work was supported in part by the NSF grants CCR-0133366, ANI-0121416, CNS-0520123 and CNS-0520033.

On the other hand, the attestation done at the level of memory block of the virtual machine is difficult to be evaluated at the client platform and incurs significant overhead.

In this paper, we propose Satem, a service-aware trusted execution monitor, to achieve trusted service code execution across client-service transactions. The Satem architecture consists of a *service commitment protocol*, a *trusted execution monitor* in the operating system kernel on the service platform, and a *trust evaluator* on the client platform. The service commitment protocol is the key to providing a priori guarantees on trusted service code execution across transactions. During this protocol, before starting a transaction, the client requests the trusted execution monitor on the service side to provide a *commitment*, which describes all the code files the service may execute in all circumstances, such as executables, libraries, etc. We define a procedure for the service provider to generate the commitment through cooperation with the service software vendors and a third-party trusted authority. The client uses the trust evaluator to verify the commitment against its local policy and then starts the transaction. On the service side, the monitor enforces the commitment after the service was started to ensure that the trusted code execution promised by the commitment will not be compromised (i.e., it forbids the service to load any code files that are either undefined in the commitment or tampered with).

To initialize trust on the execution monitor, the operating system kernel (including the trusted execution monitor) of the service provider is attested through a trusted boot process using the Trusted Platform Module (TPM) [28] specified by the Trusted Computing Group (TCG). The attestation results along with the commitment are presented to and verified by the client during the service commitment protocol. Therefore, the successful verification of the OS kernel, the monitor, and the commitment convinces the client that (1) the service has executed only trusted code up to the time of commitment; and (2) the service will continue to do so during the transaction due to the enforcement of the service commitment.

We have implemented a Satem prototype under the Linux 2.6.12 kernel and the TPM (by National Semiconductor) integrated into IBM ThinkCenter S51. The core of the prototype is the trusted execution monitor, which is small but integrated in many places in the operating system kernel, including system and kernel calls. To add these modifications, we patched the original Linux kernel. Furthermore, we have implemented and evaluated two common Internet services on top of our prototype: a web service and DNS.

This paper has three contributions. First, we propose Satem, a novel service-aware trusted execution monitor that provides an a priori guarantee to a client that only trusted service code will be executed across the upcoming client-service transaction. Due to its service-awareness, Satem only protects code executed by the service and reduces both the attestation overhead and false positives. Second, we propose a certificate-based trust evaluation method. In this method, the service requester does not have to collect and manage a huge database of authentic code hashes. Instead, the evaluation of the service trustworthiness is reduced to one certificate check. Finally, the experimental results for two common Internet services implemented over our prototype show that Satem incurs low overhead to both the services and the provider platform. Furthermore, Satem does not impact the performance of unprotected services.

The rest of the paper is organized as follows. Section 2 motivates Satem by presenting several security threats that are either unsolved or only partially solved by existing methods. Satem's system architecture is described in Section 3. Section 4 presents the service commitment protocol. Our prototype implementation is explained in Section 5, and experimental results for two Internet services executed over this prototype are presented in Section 6. Section 7 discusses a number of limitations and future work, and Section 8 goes over the related work. Finally, the paper concludes in Section 9.

2 Motivation

To motivate our research, this section presents three real-life threats that could be faced by services and their users. Trying to address them using existing attestation-based methods is difficult or even impossible. For all these threats, we consider an online banking service that runs on a Linux-based server. The service consists of an `Apache` web server, an Apache Tomcat JavaServlet engine, and java based online banking service applications. We assume the attacker gains root privilege on the server, and her goal is to trick the user into having requests processed by untrusted code. Depending on the attacker's motivation, the untrusted code may help the attacker steal the user's confidential information such as account login credentials or perform some evil tasks (e.g., transferring certain amount of money from the user's account to another account every time the user logins).

Attack 1: Service Spoofing
The attacker does the following:

1. Runs her own evil service, i.e., `/tmp/ehttpd`, on TCP port 80.
2. Runs the legitimate Apache binary, i.e., `/usr/apache/bin/httpd`.

After step 1, if `/tmp/ehttpd` is not in the scope of attestation, no record is taken. After step 2, the `/usr/bin/httpd` is attested if it is in the scope of attestation. At this moment, however, the evil service runs

on TCP port 80 rather than the legitimate `httpd` service because `httpd` failed to bind to the port.

The attestation cannot reveal the problem if it does not cover `/tmp/ehttpd`. To solve the problem by including `/tmp/ehttpd` in the attestation scope implies including all files on the system because there is no way to predict which file will be used to attack the service. Attesting the entire file system is impractical. Not only does it increase the cost, but also makes it difficult for the requester to assess the attestation result. For example, knowing that both `/tmp/ehttpd` and `/usr/apache/bin/httpd` were executed does not help the requester to understand which is the service she intends to connect to.

An alternative to counter this attack is to also attest the runtime process status. This solution, nevertheless, further complicates the attestation because the process status constantly changes. Furthermore, since the process status is not standard, the requesters will be unable to verify its integrity unless it is also published. In general, this is unacceptable to the service provider because it seriously compromises its privacy.

Attack 2: Service Tampering
In this example, a determined hacker attempts to compromise the service integrity by forcing `httpd` to link to an evil shared library (e.g., `/lib/tls/libc.so.6`) without being detected by the attestation. She does the following:

1. Runs a program which also uses `libc.so.6`, i.e. `/bin/ls`.
2. Installs an evil shared library with the same name in `/tmp`, `/tmp/libc.so.6`.
3. Sets `LD_LIBRARY_PATH=/tmp:/lib/tls`.
4. Runs `/usr/apache/bin/httpd`.

The attack is based on the assumption that `/tmp` is not watched by the attester. Also, as explained in Attack 1, it is impractical to attest all files. After step 1, the `libc.so.6` as well as `/bin/ls` are attested correctly. After step 4, `httpd` is attested, but linked to the evil library even though the attestation results contain the correct `httpd`, `ls`, and `libc.so.6`. The attack may be detected by attesting and publishing the environment variables. However, similar to attesting the process status, this is not acceptable in general.

Attack 3: Post-Request Attack
Compared with the difficulty of assessing trustworthiness of the service as shown in attacks 1 and 2, an even more complicated problem comes from the impossibility to guarantee a priori that the service will run only trusted code after the initial trust is established. In other words, the trustworthiness of the service is valid, at best, at the time of attestation, not even the time of verification. From then on until the next attestation, the service is completely vulnerable to tampering.

Let us assume that the requester decides to trust the service to process the sign-on request after verifying the attestation result. The attack takes place as follows:

1. The requester sends her user name and password to the service.
2. Before receiving the request, the attacker replaces `/usr/apache/servlet/login.jsp` with a malicious software, `elogin.jsp`, which in addition to authenticating the user performs a balance transfer of $Sum to another bank account. In this way, the attacker does not have to know the user login credentials, but is able to steal money from her account.

Since the malicious `elogin.jsp` has not been called by the time of the pre-request attestation, it is not revealed by the attestation report. However, it is invoked after the requester decided to trust the service and send the request. Although the attestation can be done immediately upon execution, the client will not know this until it requests the next report.

In summary, although existing attestation based approaches, such as [20], address attacks 1 and 2, they are subject to high false positive rates due to lack of service-awareness. The attack 3 cannot be handled by any existing approaches because they are unable to guarantee trust across service transactions before the transactions start. As we will describe in the following sections, Satem solves all these types of attacks.

3 The Satem Architecture

In this section, we present the architecture of Satem, which comprises of components on both the service provider and the requester sides. As Figure. 1 shows, the service provider components include a TPM (Trusted Platform Module), a trusted execution monitor, and a commitment for each protected service. On the service requester side, Satem includes a trust evaluator and a trust policy. In our model, we assume that the attacker is unable to perform direct hardware attacks. For example, she cannot write to or read from the TPM, network card, CPU registers or physical memory without going through the OS kernel. In particular, we assume that the attacker cannot perform Direct Memory Access (DMA) based attacks. Other than that, we consider that the attackers can get super-user privileges and modify any software, including the OS kernel, at any time.

Figure 1. Architecture of Satem

3.1 TPM: The Root of Trust

At the boot time, the service provider runs through a trusted boot procedure, in which each component in the boot sequence attests the next one before handing over the control. The TPM helps establish trust on the OS kernel and the monitor, which is the root of trust in all service transactions. The attestation result is saved in a PCR register (PCR_0), which is an internal configuration register of TPM. The only way to change the content of a PCR is through the function `TPM_Extend`, which computes a `SHA1` hash over the PCR's current content and the new object to attest. This prevents the PCR content from being reset (i.e., the attacker can change the attestation value, but it cannot set it to an arbitrary predetermined one). In our case, the TPM invokes this function to attest the BIOS image and transfers control to the BIOS after that. Consequently, the BIOS calls `TPM_Extend` over the OS loader (e.g., LILO), and the latter does the same over the OS kernel image, denoted as OSK. As a result, after the OS kernel is loaded,

$$PCR_0 = SHA1(SHA1(SHA1(0|BIOS)|LILO)|OSK)$$

assuming $PCR_0 = 0$ initially.

The content of selected PCRs is reported via the `TPM_Quote` API, which signs the content with a TPM internal key. To counter replay attacks, this API also includes a 20 byte random $nonce$ as a parameter in the signed report. We assume that the TPM is unbreakable and, therefore, we consider that the attestation it conducts and the report it produces cannot be tampered with [1].

3.2 Trusted Execution Monitor

The monitor resides in the OS kernel of the service provider and has two main goals: (1) provide a guarantee to the service requester that only trusted code will be executed by the service during the transaction, and (2) enforce fail-stop protection of the service during transactions. The monitor is loaded and attested along with OSK. It starts to run in *attestation* mode to attest all dynamically loaded OS kernel modules, denoted as $M_1, M_2, ...M_N$. Accordingly, it updates PCR_0 every time a new module is inserted as follows:

$$PCR_0 = SHA1(...SHA1(SHA1(PCR_0|M_1)|M_2)|...|M_N)$$

Therefore, the attestation results captured in PCR_0 are sufficient to prove the trustworthiness of the OS kernel including its loaded modules. In particular, the results show that a genuine Satem monitor exists on the service provider. The monitor ignores any code execution in user mode, including network services, while in *attestation* mode.

The monitor needs to be switched to *monitoring* mode before the protected service starts. In this mode, it protects both the kernel (including itself) and the protected services from being tampered with. The monitor cannot be switched back to the *attestation* mode or disabled unless the system is rebooted. This guarantees the integrity of the OS kernel across service transactions. More details about the *monitoring* mode will be discussed in Sections 4 and 5.

3.3 Service Commitment

Each protected service has an associated commitment that describes all the code the service may execute in its entire lifetime. It begins with the identifier of the software, such as its name and version, followed by the integrity descriptions of all the software code files in the format of a tuple <file name, SHA1 hash value>. A snippet of a possible Apache web service commitment is presented in Table 1.

It is the service provider's responsibility to prepare the commitments of its protected services. There are two ways to determine it. One is via testing (i.e., tracing the service for every possible request to find out what code it executes). In theory, this method may be incomplete since it is hard

[1] The current TPM specifications use SHA1, which has been found breakable [31]. We expect future releases of TPM to be upgraded with a stronger one-way hash function such as SHA256.

```
software name   =  Apache
version number  =  2.0.50
file name       =  httpd
SHA1 value      =  7e7923bb0b7a0e74d2e...
file name       =  libaprutil-0.so.0
SHA1 value      =  d888e5f9916761cca24...
...
```

Table 1. A Commitment Example

to ensure that the test exhausts all branches of executions. A better way is to trust the service software code producers to provide this information [14]. Satem adopts and extends this approach. In Satem, $C(S)$ is a certificate signed by a certificate authority (CA) trusted by service requesters of service S. This commitment is generated as follows:

1. *Request code certificates.* The service provider requests each vendor to generate a self-signed code certificate in the same format as the commitment for its code.

2. *Sign the commitment.* The requester forwards all the code certificates and the commitment to the CA. The CA needs to verify the signatures of all code certificates and compare the code hashes in the commitment against the certificates. The CA signs the commitment if and only if it verifies all code certificates and code hashes in the commitment.

$C(S)$ only guarantees to the requester that the code described in $C(S)$ is what its vendors released. The requester has to verify against its local trust policy that vendors and their code are trusted. To the service requester, decoupling code trustworthiness from genuineness not only simplifies trust management, but also provides the flexibility to use any trust policy.

The CA plays a central role in trust establishment. Its job is simplified, however, by decoupling code genuineness from code trustworthiness. All a Satem CA has to do is to verify authenticity of software code certificates. Unlike issuing identity certificates, no manual and time consuming background investigation is necessary. Furthermore, the need for certificate revocation is minimal because the certificate only vouches for the fact that the vendor certifies the code's unique digest. The only possible scenario for revocation is when the attacker compromises the signing key of the software vendor and generates bogus integrity description for the vendor's software. Although such certificate authority service does not exist today, any reputed security organization (e.g., CMU CERT [4] or SANS [8]) can take over this role.

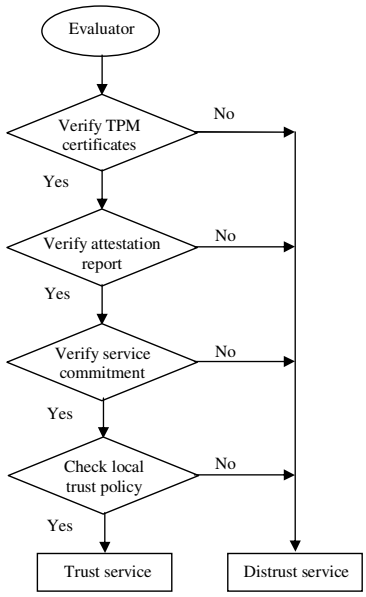


Figure 2. Satem Trust Evaluation

The commitments of the protected services are loaded by the monitor in the monitoring mode when the services are started. The monitor then enforces them by executing the following three actions during transactions. First, the service will not be allowed to load any code that is not defined in its commitment. Second, the monitor forbids the service to load any code that is defined in the commitment but tampered with. Third, the monitor protects the kernel and itself from being tampered with by disabling module insertion during transactions. An extensive description of the methods used to enforce the commitments is presented in Section 5.

3.4 Evaluator and Trust Policy

In order to establish a trusted transaction with a service, the requester asks the service provider to deliver an attestation report of the service platform OS and the service commitment. The evaluator determines if the service execution described by the report and its commitment can be trusted. Figure. 2 illustrates the trust evaluation process. The evaluator first checks the authenticity and the integrity of the report and the commitment. For the report, this requires the authentication of the TPM public key (TPM's private key is used to sign the report). TCG defines a series of certificates for users to authenticate TPM and its public keys. After verifying the genuineness of the TPM and TPM keys, the evaluator checks the attestation report. For the commitment, the evaluator needs to authenticate the public key certificate of the CA which signs the commitment. We assume that the

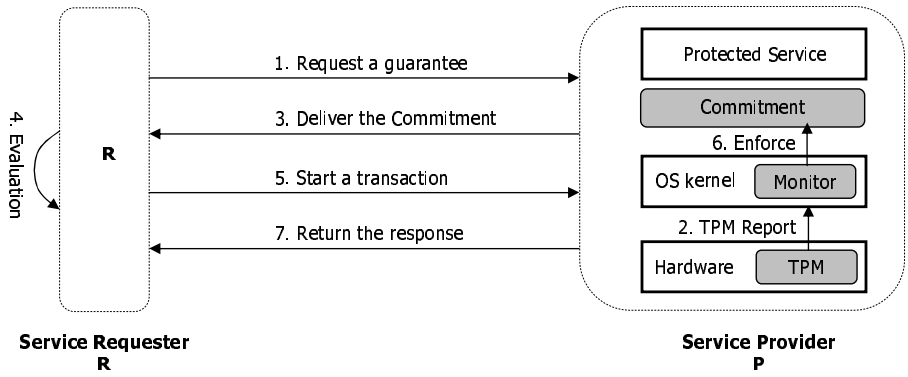


Figure 3. The Steps of the Service Commitment Protocol

requester has means to verify these certificates (e.g., PKI).

Verifying the report and the commitment proves the genuineness of the code that can be executed by the service platform OS kernel and the service. The requester has to verify the commitment against its local trust policy to make the trust decision. Although creating an appropriate trust policy is an interesting research topic, it is beyond the scope of this paper. A promising approach is to have the service software certified using methodologies proposed by Voas [29]. Here, we simply assume that the policy exists on each service requester who wants to use Satem. If the evaluator verifies the report successfully, the user will trust the execution monitor to enforce the commitment. Consequently, if the evaluator verifies the commitment, the user is convinced that the service will only execute trusted code.

4 The Service Commitment Protocol

The requester R establishes trust on the service S on the service provider P through the protocol illustrated in Fig. 3. This protocol assumes that the attestation results obtained at the boot and OS kernel loading time have been saved in the TPM (as described in the previous section).

1. R sends a request (TST) demanding P to provide a guarantee of trusted execution of S,

   $$TST =< SID, nonce, PK_R >$$

   where SID is the service identity (in a TCP/IP network, this is the ip address and port number), $nonce$ a random number, and PK_R its public key.

2. Upon receiving TST, P generates a key k, which is saved in the kernel memory controlled by the monitor. Then, it calls the TPM to generate a report Rep of the content of PCR_0,

   $$Rep =< PCR_0, p >$$

   where PCR_0 is the content of PCR_0 of the TPM. p is a parameter fed into the TPM for report generation. It is defined as

   $$p = SHA1(nonce|SHA1(C(S)) \\ |SHA1(PK_R)|SHA1(k)|m)$$

   where $C(S)$ is the commitment and m is the current execution mode of the monitor (1 in monitoring mode, 0 in attestation mode).

3. P sends Rep to R for evaluation. In addition, it encrypts k with R's public key PK_R (denoted as $(k)_R$) and sends it together with the commitment $C(S)$ to R.

4. R must verify Rep and $C(S)$ against the local trust policy before starting the transaction. R first verifies the authenticity and integrity of Rep and $C(S)$. In addition to verifying the TPM signature, it decrypts $(k)_R$ with the private key SK_R, sets `m=1` (i.e., monitoring mode) and computes p' using $C(S)$ received at step 3, its own copy of $nonce$, and PK_R. Rep is verified if and only if $p = p'$.

5. When everything has been verified, R sends S the request $Req(S)$ to start the transaction. R and S use k to encrypt the transaction.

6. The monitor starts enforcing $C(S)$, which guarantees that S will only execute trusted code to process Req.

7. Finally, S may generate and send back a corresponding response Res.

The trust decision on the service S is made at step 4. From Rep, the user learns that the service platform is booted into a trusted Satem kernel. Knowing $m = 1$ from p and $C(S)$ convinces the user that the monitor is trusted to enforce $C(S)$. The enforcement begins from the beginning of the service S since the monitor loads $C(S)$ when S starts. The p of Rep also proves to the user that k, PK_R,

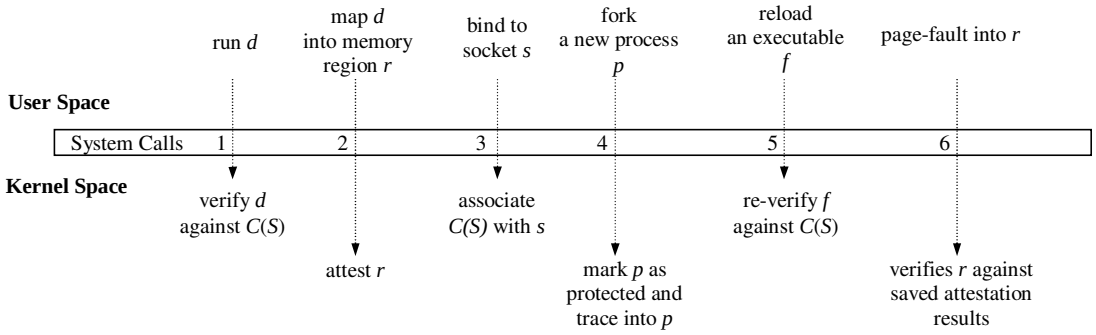


Figure 4. Satem Commitment Enforcement Work Flow

and $C(S)$ received with Rep are the same with those used by the monitor, which defeats spoofing attacks.

From step 5 on, both R and P use k to encrypt the transaction traffic. This prevents the attacker from hijacking the transaction by launching a man-in-the-middle attack. The attacker may try to steal k by intercepting TST at step 1 and replacing PK_R with its own PK_A. Then, it intercepts the returned k at step 4 (encrypted with PK_A), decrypts it and re-encrypts it with PK_R. This attack will be detected by R at step 4 due to the inclusion of $SHA1(PK_R)$ in the parameter p. On the other hand, the attacker can pretend to be P. For this attack to succeed, the attacker's machine must be Satem-enabled. Otherwise, the requester will refuse to trust it at step 5. However, when Satem is enabled, it guarantees that the service will only execute trusted code no matter who owns the service provider platform.

5 Prototype Implementation

In order to verify the design concepts and understand the performance of Satem, we have implemented a prototype under the Linux 2.6.12 kernel and the TPM (by National Semiconductor) integrated into IBM ThinkCenter S51. The most important part of the prototype is the trusted execution monitor. The prototype includes also TPM control functions as well as the evaluator on the client side. We have not implemented the attestation functions in BIOS and LILO, which have already been implemented in the Enforcer project [5].

5.1 Satem Monitor and Commitment Enforcement

The focus of the Satem monitor is to provide a fail-stop protection mechanisms to enforce service commitments. Our implementation has less than 1000 lines of C code. The complexity, however, comes from the fact that the monitor code is integrated into the OS kernel almost everywhere by inserting checkpoints to kernel calls such as `do_execve` and `sys_open` to intercept new code execution invoked by protected service processes. These modifications are added by patching the original Linux kernel of the service provider platform. When a service needs protection, the monitor associates a protection flag with all processes executed by this service, memory regions mapped by these processes, and code files opened by them. By having this flag, we achieve service-awareness by limiting the scope of Satem within the protected service instead of performing attestation and other Satem actions on irrelevant programs. More details about our protection mechanism will be presented throughout this section.

The commitment of a protected service is loaded into the kernel memory. It is implemented as a table, and Satem uses the name of the service code file as the key to look up the corresponding hash value. We use the Linux kernel `crypto` API to implement SHA1 functions. Figure. 4 shows, from left to right, how the monitor in the monitoring mode enforces the commitment when a service is launched and executed. The up-down arrows represent the interception of kernel calls, in which the monitor has the checkpoints. We assume that the services are based on TCP/IP.

5.1.1 Loading

When a service S is started, its daemon program d is executed; this results in an invocation of `do_execve()` that traps into the kernel. The monitor intercepts the call in the kernel (arrow 1 in the figure) and does the following:

1. Compares d with a preset list of protected services, PS, to see if $d \in PS$. Service providers that want to use Satem must provide such a list. In brief, it defines the mapping between d and $C(S)$. For instance, the following PS defines two protected services: a web service associated with a commitment `cw` and a name service associated with a commitment `cn`.
   `</usr/apache/bin/httpd cw>`

`</usr/bind/named cn>`
The monitor does not verify whether PS tells the truth about what commitment is associated with the service. In this way, an attacker may try to associate a bogus commitment with the service in order to run untrusted code. From a client perspective, however, this is not a problem because the evaluator on the client side will refuse to trust the commitment during the service commitment protocol. If the commitment is trusted, the monitor will detect immediately any attempt to execute untrusted code either by d or by the service.

2. If $d \in PS$, the monitor marks the current process as protected. Then, it reads the commitment, $C(S)$, attests d, and verifies whether d is defined and has the same `SHA1` value as in $C(S)$.

3. When an interpreter, i, is loaded (e.g., `ld.so` for binaries, `perl` for perl scripts), the monitor recognizes that it is loaded with a protected service and thereby attests i in the same way as d.

4. The kernel maps memory for d and i using `do_mmap` (arrow 2 in the figure). Since the mapping is called by a protected service, the monitor marks the memory region r as protected and partitions the entire region into a list of small segments such that $r =< sg_0, sg_1, sg_2, ...sg_{n-1} >$. Each sg_i corresponds to a page. The monitor then reads each page to fill the segments sg_i one after another and computes SHA1(sg_i). The attestation results are saved in kernel internal memory. After attesting each sg_i, the kernel does not attempt to keep the content of the page in memory.

The monitor saves the attestation results of protected memory regions, commitments, and the secret key k (defined in Section 4) in system memory rather than in the tamper-resistant TPM. This is caused by the fact that TPM does not have sufficient space for the variable-sized results. Our prototype addresses this problem by letting the monitor define the area and forbidding access to it from any non-Satem user-space applications. We assume that the OS kernel correctly protects kernel memory such that the only channel to access this area from user space is through direct memory mapping mechanism such as `/dev/kmem` and `/dev/mem`.

5.1.2 Linking

When d is fully loaded, the kernel transfers the control to its interpreter to load the shared libraries. Similar to d, a library l is mapped, not read, into memory and attested.

5.1.3 Binding

Once loaded, the service program needs to bind a network socket. The `sys_bind` system call traps into the kernel (arrow 3 in the figure). The monitor first checks whether the current process is protected. If so, it links the socket to $C(S)$; in this way, the service platform can deliver the right commitment to requesters.

5.1.4 Cloning

The current process may create new child processes. In this case, the monitor intercepts the fork system call (arrow 4 in the figure) and checks if the current process is protected. If so, it marks the new child process as protected and links it to $C(S)$. Then, the monitor will track the child process in the same way with its parent.

5.1.5 Code Changing

In monitoring mode, the monitor disables dynamic loading of kernel modules. Hence, the only way to modify the kernel is to reboot into a different one. In this case, the change is captured by the TPM in the trusted boot and revealed to the requester when the commitment is verified. We assume that the OS kernel isolates processes correctly (i.e., it forbids one user process from accessing memory of another process without authorization). Therefore, once the code is loaded into memory, the attacker cannot modify it without compromising the kernel.

The attacker, however, can modify arbitrary code files. The monitor does not attempt to catch changes in code files. Instead, it detects and blocks any attempt of a protected user space process to invoke the compromised code. For instance, reloading a modified d (arrow 5 in the figure) will be blocked in `do_execve` call.

A less obvious attack is to directly modify mapped binary code chunks on the disk. Consequently, the tampered code chunk is loaded into memory by `filemap_nopage` when a page fault occurs. Attestation at the granularity of files cannot detect this problem. A mapping can exist even without keeping the underlying file open, which makes it impossible to catch changes just by watching system calls such as `sys_open()`, `sys_read()` and `sys_write()` as in [20]. Satem can defeat such attacks by using the previously saved hash values for the protected memory regions. Each time a memory fault causes a real read of a missing page, the monitor recomputes the SHA1 value of the page to be read and compares it with the saved value (arrow 6 in the figure). If the values are different, the monitor concludes that the page was tampered with, on the disk, after being mapped.

5.1.6 Monitoring Interpreted Programs

Interpretation of scripts or virtual machine based executables (e.g., perl scripts or java servlets) are more difficult to monitor because their execution takes place in a black box from the monitor point of view. To solve this problem, we monitor all the files opened and read by protected processes (e.g., in `sys_read` call) and verify each of them against the commitment.

5.2 Lazy Attestation

Attestation of large code files and mapped memory regions is costly. This can cause significant overhead to the service provider system since the service may repeatedly invoke the same code. Since not all code segments in a binary will be loaded for execution, Satem uses lazy attestation to minimize the overhead by avoiding attesting code segments which are not used. When the binary is executed for the first time (via `do_execve()`), Satem attests the file as a whole. In addition, it partitions the binary file into page-sized chunks and also attests the loaded chunks such as those containing the ELF headers. The results are cached, and when the file is executed subsequently, Satem does not attest again the file if the commitment has not been changed. Instead, it verifies each of the loaded chunks against the cached attestation results. Satem will attest the entire file only if the execution loads a chunk which has not been loaded before. Similarly, when a code segment is mapped, Satem uses the previously saved attestation results rather than re-attesting it.

5.3 TPM Functions and Satem Trust Evaluator

Our implementation is based on the IBM TPM driver. Satem uses TPM in a very light way. It only uses TPM for attestation and result reporting (i.e., the `TPM_Extend` and `TPM_Quote` functions). The Satem trust evaluator is a simple user space application. It performs RSA signature verification. The trust policy is set to "allow any" in the prototype. We have not implemented the commitment certificate yet, but this is trivial if the certificate authority is known to the evaluator.

5.4 Case Studies

We conclude this section by demonstrating how Satem solves the three security threats introduced in Section 2.

Attack 1: Service Spoofing
Satem does not interfere through the attack; at the end of the attack, `/tmp/ehttpd` is loaded up and listens on TCP port 80. However, when a user tries to connect to the service, Satem fails to retrieve the commitment since it understands that `/tmp/ehttpd` is bound to the service port, and this program is not in the PS list. Consequently, the user will cancel the connection because she is unable to evaluate the trustworthiness of the service.

Attack 2: Service Tampering
Satem loads the service commitment immediately after the service is loaded (e.g., in this case after `/usr/apache/bin/httpd` is executed). Due to the service-awareness, Satem tracks the execution of the protected service and attests everything it loads, including `/tmp/libc.so.6`, while ignoring `/lib/tls/libc.so.6` because this library is not loaded by the protected service. If `/tmp/libc.so.6` is defined in the commitment, Satem allows it to be loaded, but the user will be aware of it when evaluating the commitment. Otherwise, if `/tmp/libc.so.6` is not defined in the commitment, Satem prevents it from being loaded.

Attack 3: Post-Request Attack
Once the commitment has been evaluated and trust has been granted by the user, Satem will block any code from being loaded by the service, which includes `/usr/apache/servlet/login.jsp`.

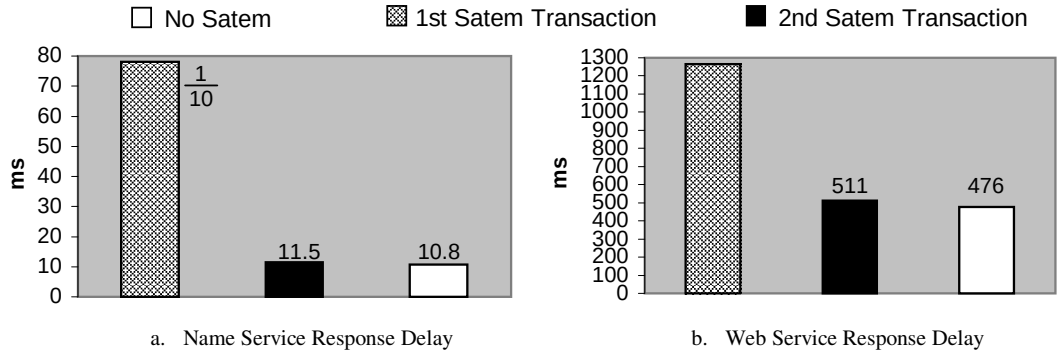
6 Evaluation

To evaluate the performance of Satem, we measured the overhead in terms of monitoring scope, transaction processing delay, commitment delivery, as well as the overhead imposed by Satem on service execution and kernel calls.

6.1 Methodology

Two Internet services were used for experiments: a web service and DNS. The former is a simple conference registration center running Apache 2.0.50 [3] and a `perl` based registration program. The latter is a 8.4.6 BIND server [6]. The services were executed on an IBM ThinkCenter S51 desktop with a Pentium 4 3.2G Hz processor, a National Semiconductor TPM, and 1G RAM. The client was an IBM Thinkpad R40 laptop with Intel M 1.4G Hz processor and 256M RAM. To measure the CPU cost, we turned on `oprofile` [7], which uses the Pentium 4 CPU hardware performance counter to count `GLOBAL_POWER_EVENTS`. To evaluate Satem-enabled kernel calls, we directly read the timestamp counter (TSC).

In our experiments, to measure the overhead to service transaction handling, we ran a simple profiler program from the service requester. In the web service transaction, the



a. Name Service Response Delay

b. Web Service Response Delay

Figure 5. Transaction Processing Time

Service	Satem Monitoring Scope (number of files)
web	42
DNS	13

Table 2. Scope of Monitoring

Service	Commitment Latency (ms)
web	782
DNS	765

Table 3. Commitment Delivery Latency

profiler registers a user with the conference and downloads the conference agenda. In the name service transaction, the profiler conducts a number of DNS queries.

6.2 Scope of Monitoring

Satem does not have to monitor all files on the service provider due to its service-awareness. In this way, it can significantly reduce the monitoring overhead. For our experiments, Table 2 shows the size of the monitoring scope in terms of number of files. In the web service case, less than 50 files are monitored at runtime: 10 kernel files, 22 library files, 1 registration CGI script, the `perl` interpreter, and 9 `perl` modules. Compared with IBM TCG Linux [20], for instance, which attests more than 250 files, the scope of Satem attestation is significantly reduced.

6.3 Transaction Processing Time

To users of protected services, the performance is measured by the time to finish a service transaction. As Figure. 5.a and b [2] show, there is a significant overhead in the first transaction with a Satem-protected services. However, the overhead is dramatically reduced in the subsequent transactions. For instance, the overhead in subsequent transactions of both web and name service is less than 8%. Further experiments as explained in the rest of the section demonstrate that: (1) the high delay in the first transaction is due to the cost of generating the TPM report and performing initial attestation; and (2) the low impact in the subsequent transactions is due to our lazy attestation mechanism.

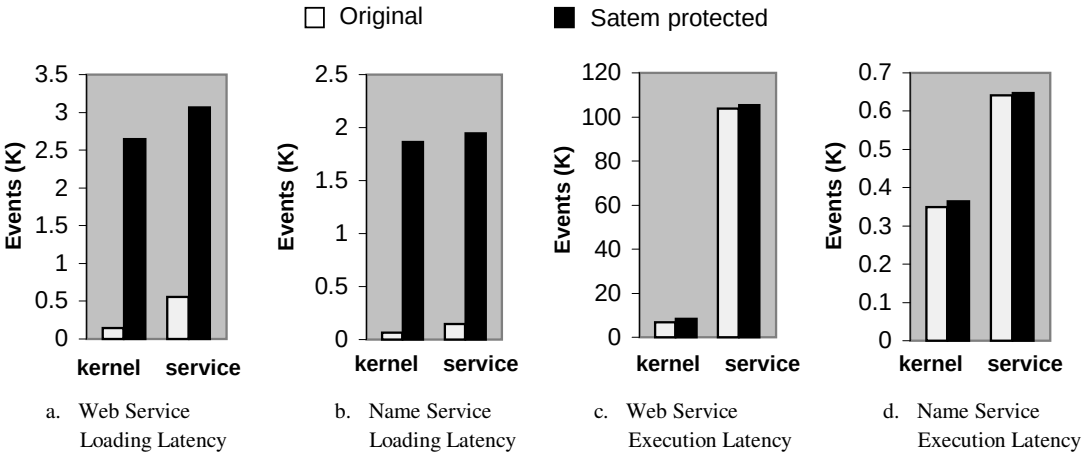
6.4 Commitment Delivery Latency

We measured the service provider's cost for commitment delivery as the latency of getting the commitment ready for delivery. As illustrated in Table 3, this cost does not have a clear correlation with the service type. The reason is that the `TPM_Quote` call dominates the latency. This also demonstrates the efficiency of Satem compared to other attestation-based methods because Satem only calls the TPM once in delivering the commitment if no OS changes occur afterward.

6.5 Cost of Service Execution

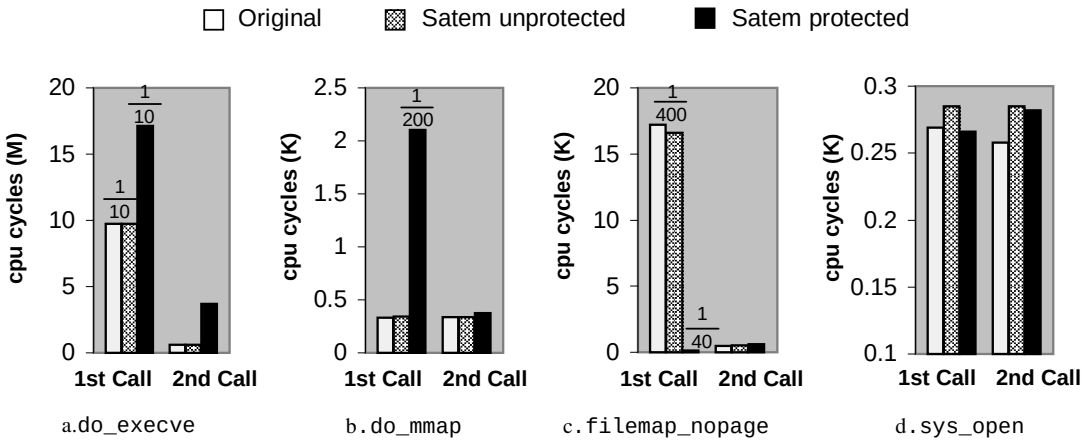
The CPU cost for service loading and execution is measured in number of the `GLOBAL_POWER_EVENTS` as shown in Figure. 6. In these graphs, the left columns represent the service cost in kernel, and the right columns represent its total cost (kernel and user-level). To measure them, we let the profiler continuously initiate transactions and read data from `oprofile` every 50 transactions.

[2] The cost of subsequent transactions may be much lower than the first one. In order to compare them in one figure, we use a reduced scale to plot the following figures: first transaction processing time at 1/10, first call to `do_execve` at 1/10, first call to `do_mmap` in *Satem protected* at 1/100, first call to `filemap_nopage` in original and *Satem unprotected* at 1/400, and first call to `filemap_nopage` in *Satem protected* at 1/40.



a. Web Service Loading Latency
b. Name Service Loading Latency
c. Web Service Execution Latency
d. Name Service Execution Latency

Figure 6. Cost of Service Execution



a.do_execve b.do_mmap c.filemap_nopage d.sys_open

Figure 7. Overhead in Kernel Calls

Clearly, Satem incurs significant overhead in initial loading of both the web and DNS services to kernel and to the services as a whole (Figure. 6.a and 6.b). This is because Satem attests each code file being loaded and each mapped memory region of the service. However, since this is a one-time initialization cost, it does not affect the performance of service transactions. As Figure. 6.c and 6.d show, Satem does not incur significant overhead to the protected services due to its lazy attestation mechanism.

6.6 Overhead in Kernel Calls

Figure. 7 shows the overhead in four kernel calls: `do_execve`, `do_mmap`, `sys_open`, and `filemap_nopage` in terms of extra CPU cycles needed to complete these calls. We measured the cost of these functions in web service transactions. All functions are measured in three cases: in the original Linux 2.6.12 kernel, in the Satem-augmented kernel in which the web service is not protected (called *Satem unprotected*), and in the Satem-augmented kernel in which the web service is protected (called *Satem protected*). For each function, we measured its overhead for the first call (the left three columns in the figures) and the overhead for subsequent calls (the right three columns in the figures).

The graphs show that the impact of Satem on the first time function call is significant. For `do_execve` and `do_mmap`, the cost is dramatically increased because of the per-page attestation for mapped code files. For `filemap_nopage`, by contrast, the cost is significantly reduced. This is because Satem needs to load every mapped page into the page cache when it attests a protected memory region. As a result, when a page fault occurs, it is very likely that the page is still in the cache. Furthermore, the

cost of the subsequent calls in *Satem protected* is almost the same as in the original kernel. This is also because of the lazy attestation mechanism. Finally, if a function in the Satem-augmented kernel is called from an unprotected service, its performance is comparable to that in the original kernel. This means that Satem does not impact other unprotected programs.

7 Limitations and Future Work

A challenging task faced by Satem is to ensure trusted handling of user' data. In addition to guarantee trusted service code execution, other mechanisms are needed to achieve this goal. First, the protected service may cooperate with other processes via inter-process communication (IPC). Satem can be extended to cover the code base of the IPC processes by including it in the service commitment. Moreover, the service provider may further call another service on a remote platform. To address this problem, we are investigating the augmenting of the monitor with new checkpoints in the network connection functions and extending Satem to a multi-tiered architecture. In this way, Satem will be enabled on each platform on the connection chain starting from the service provider. Thus, regardless of the depth on the connection chain traversed by a transaction, only trusted code can be executed to process it. Lastly, the protected service may cache user's data in memory or even save it to disk. The problem is further complicated by the fact that the data can be transformed in service transactions. The transformed data may carry the same information as the original one and thereby must also be protected. We plan to investigate data lifetime [11] to solve this problem.

Satem is designed to ensure that a protected service can not load untrusted code from the disk. An attacker can exploit, however, buffer overflow attacks to cause the protected service to run arbitrary code without changing its disk image. Satem is unable to tackle this type of attack. It only mitigates the problem in two aspects. First, Satem may reveal the code which has known buffer overflow vulnerabilities by attesting it to the user. Hence, the user can avoid trusting the vulnerable code. Second, in the case of a successful buffer overflow attack, the attacker runs her own code on the service stack without being caught by Satem. But due to the limited size of the stack, the attacker's code typically has to call other local programs on the service provider to make the attack meaningful. Satem restricts the attacker's capability of launching arbitrary local code (i.e., any code launched by the protected service must be defined in the commitment).

Satem kernel code is not modularized. The main hurdle in face of modularization is the mapping protection mechanism, which involves deep kernel internal calls. This problem can be avoided by building Satem on top of a virtual machine monitor (VMM) [16, 30, 14, 12]. In such a case, the VMM (instead of OS kernel) can control the memory access and perform integrity checks on memory regions. Another advantage of using VMM is to better protect Satem by governing the access to the memory area holding Satem data without limiting use of `/dev/mem` and `/dev/kmem`.

Transactions are not encrypted in the current implementation of the service commitment protocol. We plan to implement the encryption module using `netfilter` and the kernel `crypto` API.

8 Related Work

Satem leverages previous work done on trusted computing and execution monitoring. Early systems, such as IBM 4758 [27], Citadel [32], and Dyad [33], focused mainly on implementing powerful secure co-processors and protecting trusted software inside these co-processors. These projects made trusted secure hardware such as TCG TPM realistic.

The trustworthiness provided by Satem is based on software attestation. The idea of authenticating software by verifying the integrity of its execution stack was developed by Gasser et al. in [15]. Its application has been seen in many trusted computing models. The AEGIS system [9] makes it possible to secure PC booting by having each component in the boot process, such as BIOS and OS loader, attest the next component before transferring the control to the latter. This process continues until the OS kernel is loaded. Satem extends this idea to service execution, which is significantly more complex than booting.

Cerium [10] and XOM [18] use a tamper-resistant CPU to attest software execution stacks and execute trusted software. Satem differs from them in that Satem uses a co-processor, the TCG TPM [28], to build a trusted computing base (i.e., the Satem monitor) and lets it perform other complex trusted protocols. We also differ in terms of the attacker's models. Both Cerium and XOM do not trust the OS or main memory, while Satem trusts them after successful attestation and verification.

NGSCB [19], Terra [14], TCGLinux [20], and BIND [26] are based on the same hardware like Satem (i.e., TPM). Both NGSCB and Terra explore a virtual machine monitor (VMM) to partition a tamper-resistant hardware platform into multiple isolated virtual machines. In NGSCB, a system is partitioned into two parts: trusted and untrusted, and only the trusted part is attested. Therefore, to ensure service trustworthiness, the service provider platform has to treat the service and all its code as trusted, which may not be true all the time. Terra partitions the system into virtual machines, each of which may be dedicated to a single application (e.g., a service). As such, the trustworthiness of a service can be evaluated by attesting its virtual machine. This attestation, however, is done at memory block

level, which incurs high CPU and memory overhead. Terra achieves higher assurance of attestation because of strong process isolation provided by VMM, but lacks the capability of ensuring simple and efficient trusted execution across transactions.

TCGLinux is the first secure integrity measurement system which explored the TCG TPM. Satem borrows several implementation ideas from TCGLinux, such as intercepting system calls to attest file integrity upon reference, but differs from it in two perspectives. First, TCGLinux attests a broad range of files. This is not suitable to ensure trusted code execution of services due to the false positive problem described in the introduction. Satem, on the other hand, monitors only the protected service code without paying attention to other irrelevant file changes. Second, TCGLinux provides no assurance about trust after it is granted, while Satem can ensure it across the service transaction.

BIND achieves fine-grained attestation by tying the attestation of the code with the data it produces. Hence, it can be used only after the output was produced to verify whether this output was produced by trusted code or not. Using this solution, compromised code can gain access to confidential data or handle critical irrevocable missions (e.g., stock trading) before being detected. Unlike BIND, Satem can guarantee trusted execution of the service code before the user starts a transaction. In addition, BIND requires programmers to identify the critical pieces of code to protect, which is not a trivial task in many situations. To balance the trade-off between the protection granularity and the burden placed on programmers, Satem ensures integrity of all parts of a protected service.

Opposite to the hardware-based approaches are software approaches such as [17], SWATT [25], and Pioneer [24]. In these projects, the target system is challenged to compute a checksum of its system image using a user-defined procedure. The correctness of these approaches depends on two assumptions. First, the user has sufficient knowledge about the target system's hardware (e.g., clock speed). Second, forging the same checksum of the trusted system by the compromised system causes noticeable delay to the user. Neither holds for typical Internet services.

Satem monitors service execution to ensure that it is executed as expected. To this end, Satem is similar in spirit to a reference monitor which has been widely used to detect execution time anomaly [21, 13, 23, 22]. For instance, [13, 23] transform security policy into target object code and merge the reference monitor in the code execution. In [22], the system monitors the programs' behavior by supervising its system calls. From this angle, Satem can be considered as a monitor in the target system kernel, the policy of which is the commitment. The difference is that the object to monitor is a service, which is a composite of inter-dependent executables, rather than an individual program. Furthermore, a subtle but fundamental difference is the role played by Satem. A reference monitor is trusted by the target system and protects this system from untrusted code (i.e., service code). Satem, on the other hand, must be trusted by the requester (a different system than the one where it executes) and protects the requester from the untrusted service code.

9 Conclusions

This paper has presented Satem, a novel service-aware trusted execution monitor that guarantees trusted service code execution across client-service transactions. Users establish trust with the services through a service commitment protocol executed before starting any new transaction. The monitor exploits the TCG-specified TPM as the root of trust to build trusted components in the OS kernel, which consequently enforce this commitment. Satem achieves service-awareness by limiting the scope of monitoring to protected services instead of performing attestation and protection on all programs. We have implemented a prototype under Linux and evaluated it using two common Internet services. The experimental results demonstrate that Satem incurs low overhead to both the services and the provider platform. Furthermore, Satem does not impact the performance of unprotected services.

References

[1] http://www.w3.org/2002/ws.

[2] http://msdn.microsoft.com/architecture/soa.

[3] Apache project. http://www.apache.org.

[4] Carnegie Mellon CERT Coordination Center. http://www.cert.org.

[5] Enforcer project. http://enforcer.sourceforge.net.

[6] Internet System Consortium Inc. http://www.isc.org/index.pl?/sw/bind/.

[7] Oprofile project. http://oprofile.sourceforge.net.

[8] The SANS Institute. http://www.sans.org.

[9] W. Arbaugh, D. Farber, and J. Smith. A secure and reliable bootstrap architecture. In *Proceedings of IEEE Symposium on Security and Privacy*, 1997.

[10] B. Chen and R. Morris. Certifying program execution with secure processors. In *the Proceedings of 9th Workshop on Hot Topics in Operating Systems*, 2003.

[11] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th Usenix Security Symposium*, 2004.

[12] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[13] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, 1999.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[15] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of 12th NIST-NCSC National Computer Security Conference*, 1989.

[16] R. Goldberg. Survey of virtual machine research. In *IEEE Computer Magazine*, June 1974.

[17] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of 12th USENIX Security Symposium*, 2003.

[18] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems*, 2000.

[19] Microsoft Corp. Next generation secure computing base. http://www.microsoft.com/resources/ngscb.

[20] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of 13th USENIX Security Symposium*, 2004.

[21] F. B. Schneider. Enforceable security policies. In *ACM Transactions on Information and System Security*, 2000.

[22] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the 8th USENIX Security Syposium*, 1999.

[23] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[24] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. V. Doorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[25] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT:Software-based attestation for embedded devices. In *Proceedings of 2004 IEEE Symposium on Security and Privacy*, 2004.

[26] E. Shi, A. Perrig, and L. van Doorn. Bind: A time-of-use attestation service for secure distributed system. In *Proceedings of IEEE Symposium on Security and Privacy*, 2005.

[27] S. W. Smith and S. H. Weingart. Building a high performance, programmable secure co-processor. In *Computer Networks (Special Issue on Computer Network Security)*, April 1999.

[28] Trusted Computing Group. TCG 1.2 Specifications. https://www.trustedcomputinggroup.org/.

[29] J. Voas. A Recipe for Certifying High Assurance Software. In *IEEE Software*, 1999.

[30] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[31] X. Wang, Y. Yin, and H. Yu. Finding collisions in the full SHA1. In *Proceedings of Crypto*, 2005.

[32] S. White, S. Weingart, W. Arnold, and E. Palmer. Introduction to the Citadel architecture: Security in physically exposed environments. Technical Report TR RC16672, IBM Thomas J. Watson Research Center, 1991.

[33] B. Yee. Using secure co-processors. Technical report, Carnegie Mellon University, 1994. PH.D Thesis.