Processor-based Trustworthy Tailored Attestation

ABSTRACT

The increasing number of security-critical interactions in cyberspace require commodity computing platforms to provide attestation capabilities. The Trusted Platform Module (TPM) provided in many computers today enables a form of launch-time attestation of the entire software stack. However, such attestation includes the reporting of unrelated software and cannot reveal subsequent runtime corruption. The TPM threat model also does not defend against physical attacks. In this paper, we leverage two important trends—virtualization for security and migration of security mechanisms into the main processor—to provide Tailored Attestation, where the requester asks for reports of only the software modules required for executing a security-critical task. Tailored Attestation provides runtime integrity guarantees and is also resilient: attestation to a piece of critical code can take place despite a breach of integrity in unrelated code. Integration with the microprocessor also makes Tailored Attestation much faster than TPM attestation.

1. INTRODUCTION

Society's dependence on cyberspace increases at the same time as cyber security violations escalate. Many security breaches are due to the exploitation of software vulnerabilities in commodity Operating System (OS) or application code, leading to the installation, and eventually execution, of malicious code. With the large number and long lifetime of security vulnerabilities in commonly-used software [32, 33], we must assume that essentially all software stacks are vulnerable to compromise. In the midst of this sea of untrusted software, attestation aims to provide the important guarantee that security-critical tasks can be safely executed on a remote system.

Attestation is the reporting by a computer of the software it is running to a remote party. The remote party must then make a *trust decision*, to determine whether this computer can be trusted to securely execute a given piece of critical application code. Currently, computers supporting attestation use a Trusted Platform Module (TPM) chip [29] to report the state of every single layer of software from the BIOS code, to the application of interest, including the entire Operating System (OS) code base. To make a trust decision, the remote party must thus inspect a full software stack to verify that the critical application code and the underlying software it depends on are correctly written. This is a very daunting task in practice, as evidenced by the exploitable vulnerabilities found in most software systems, even after thorough verification by their developers.

Rather than actually verifying the trustworthiness of a software stack via code inspection, the remote party could maintain a database of software components a trusted party certified as correctly written [24]. The remote party then simply checks the reported stack is formed only of components in the database. Unfortunately, this approach suffers from many potential false positives, i.e. attestation fails when critical code can in fact execute securely [15]. Indeed, this approach is often an overkill since attestation fails when a single component is either unknown, updated or corrupted, even if it is unrelated to

David Champagne and Ruby B. Lee, "Processor-based Tailored Attestation," Princeton University Department of Electrical Engineering Technical Report CE-L2010-005, Nov 15, 2010.

critical code. Hence the need for a computing platform providing *resilient attestation*, where critical functions can securely execute and be attested to despite compromise, updates or unknown software in unrelated parts of the system.

We leverage two important trends—virtualization technology and the willingness of hardware vendors to add security mechanisms into the microprocessor—to provide *Tailored Attestation*, a resilient form of functional attestation reporting only the relevant software needed for securely executing critical functions in an application. We present the design of a small Trusted Computing Base (TCB) consisting of a thin hypervisor (or Virtual Machine Monitor) and a security-enhanced microprocessor that can implement Tailored Attestation. Exploiting the modular nature of software systems to decompose the software stack into modules, our TCB isolates each module so that it can be attested to individually. This enables the reporting of only a small fraction of the running software stack to a remote party.

Tailored Attestation is a type of *functional attestation*, as it reports only software modules whose functionality directly contributes to the remote party's security objectives. Our TCB identifies a critical software module at launch time and protects its memory space against snooping and corruption during runtime. As a result, a module deemed trustworthy at launch time can still be trusted at attestation time.

To make a trust decision, the remote party must obtain guarantees as to how critical modules interact with other software. To provide these guarantees despite reporting only partial software stack information, our TCB implements new *Shadow Access Control* and *Secure Module Transition* mechanisms. Shadow Access Control ensures the memory state of critical modules is only accessible to the module itself, except for explicit sharing interfaces. Secure Module Transitions provide hardware-software enforcement of authorized module entry and exit points and module state protection upon preemption by untrusted OS software. Tailored Attestation reports the sharing and transitions authorized by the TCB at runtime, so a trust decision can be made despite partial software stack information.

The main contributions of this paper are:

- Enabling resilient attestation and execution of security-critical code, even in the presence of corrupted code in the operating system or applications
- Defining the essential architecture in a microprocessor to support Tailored Attestation
- Defining Secure Trusted Module Transition mechanisms.

The rest of this paper is as follows. Section 2 compares our approach to past research efforts improving on TPM attestation. Section 3 presents our threat model. Section 4 describes the security mechanisms implemented in our TCB to support Tailored Attestation and gives an example. Section 5 gives a security analysis of our mechanisms. Section 6 considers the cost and performance, and Section 7 concludes.

2. PAST WORK

Recent secure computing proposals such as [8, 21, 25, 14] leverage the current trend towards virtualization (e.g. [26, 3, 12, 2]) by using a trusted Virtual Machine Monitor (tVMM). The tVMM can isolate critical application code in a Virtual Machine (VM) running a software stack much simpler than the typical software stack which includes a commodity OS. This VM isolation allows a TPM chip to attest only to the simpler stack that is easier for the remote party to verify. This approach does not achieve functional attestation, however, as there is still a need to report an entire software stack, which includes software whose functionality is unrelated to the critical application code (e.g. every OS device driver is reported). While [6, 30, 31] isolate application code from the OS using a tVMM, they do not provide attestation capabilities.

Another trend in commercial computing platforms like [13, 2, 11] is to integrate security mechanisms in the microprocessor chip itself. This allows speeding up certain operations that otherwise go through the external TPM chip, which has much lower computation speed and is connected to the microprocessor via the slow Low-Pin Count (LPC) bus. Executing in the microprocessor also provides greater visibility into software context, allowing hardware features such as Device Exclusion Vectors [2] to protect critical code against Direct Memory Access (DMA) attacks. These platforms cannot provide resilient attestation, however, as there is still no protection against privileged code attacks on security-critical application modules.

Flicker [19] uses the late launch hardware mechanism [13, 2] to launch, without a hypervisor, critical application code in an execution compartment isolated from the rest of the software stack. Although this solution achieves functional attestation by attesting only to the critical compartment, it cannot support frequent security operations, due to the high overhead of a context switch (called a "world switch" in virtual machine environments) to such critical code—on the order of several milliseconds [19]. Another research proposal aiming for functional attestation is property-based attestation [23], where the platform reports properties rather than software measurements. However, it still requires a trusted third party (or the remote party itself) to inspect a full software stack to derive the properties to be attested.

Other research proposals have considered physical attacks, e.g., [27, 18] show hardware-based memory protection mechanisms that can provide runtime integrity to application software modules despite potential attacks from privileged code or physical attackers, and thus enable some form of resilient attestation. The Bastion architecture [36] is closest to our approach, but fails to provide any attestation capabilities. The Secret Protection (SP) architecture [17, 7] offers a type of functional attestation but supports only one protected software module at a time, or modules belonging to one trust domain [7]. The AEGIS architecture [27] also provides runtime memory integrity, but does not support functional

attestation, as it systematically reports all OS modules supporting at least one critical application rather than only those depended upon by the application of interest. [15] aims for resilient attestation by introducing a formal integrity model that simplifies trust decisions, but it does not provide physical attack protection or functional attestation.

3. THREAT MODEL

Unlike the TPM threat model, our threat model includes physical attacks as well as software attacks. We define the microprocessor chip as our hardware security perimeter, as in [18, 27, 17, 7]. This is much smaller than the TPM security perimeter (which includes the entire computer box). Having the entire computer box as the security perimeter is no longer reasonable, since this includes machine memory, buses, disks, input/output devices and network interfaces—which are very vulnerable to physical attacks. Physical attacks are also especially relevant with mobile computers, which are easily lost or stolen. Circuits and cache and register storage within the microprocessor chip are considered safe from physical attacks because successful chip probing attacks are extremely difficult, due to a highly layered manufacturing technology for microprocessors. However, physical probing attacks on the hardware outside of the microprocessor chip are considered fair game in our threat model. We assume the microprocessor does not contain flaws, hardware Trojans or viruses.

We assume that all software contains vulnerabilities that can be exploited by attackers, except for a small number of trusted software modules and the trusted hypervisor. Hence, software attacks can be carried out by malicious OS or application code snooping on, or corrupting, security-critical software state in disks, memory, caches or registers. A malicious OS could snoop on or corrupt a sensitive software module while its state is paged out on disk or by setting up a Direct Memory Access (DMA) transfer. An adversary with physical access to the platform can also intercept its signals and tamper with its hardware (except for the microprocessor). For example, a physical attack could use hardware probes to snoop on or corrupt values transiting on the processor-to-memory bus.

We consider both passive and active attacks. Adversaries can easily carry out a passive attack on data confidentiality, i.e. observe bus, memory or hard drive data. Similarly, attackers can affect data integrity using any combination of *spoofing* (illegitimate modification of data), *splicing* (illegitimate relocation of data), and *replay* (substituting stale data) attacks.

We consider operational attacks, not developmental attacks. We do not consider covert or side channel attacks in this paper. We do not aim to prevent denial of service. Since we assume an untrusted OS and physical attacks, such availability attacks can easily be achieved by a compromised OS or a physical attacker, in any case.

4. ARCHITECTURE

4.1 Baseline Platform

The baseline CPU we consider in this paper provides virtualization support (as in [12, 2]) with at least three software privilege levels, or protection rings. From most privileged to least privileged, these are also denoted the hypervisor mode, supervisor mode and user mode. The hypervisor gives its Virtual Machines (VMs) the illusion they each have unrestricted access to all hardware resources, while retaining ultimate control on how the resources are used. We assume there are two levels of on-chip cache (L1 and L2), as is the case in most general-purpose processors.

The platform has virtualized software stacks, where guest operating systems run in separate VMs, monitored by a hypervisor. We use the term *software entity* to refer to a component in a VM running in its own address space. In this paper, the guest operating systems and the applications are the software entities. We use the term *machine memory* space to denote the actual physical memory available in the hardware. The hypervisor virtualizes the machine memory space to create *guest physical memory* spaces for the guest operating systems it hosts. When an OS builds a page table for an application or for itself, it maps virtual pages to page frames in its guest physical memory space. To map guest virtual memory to machine memory, the VMM maintains for each software entity a *shadow page table* [1] translating virtual addresses to machine physical addresses. The hypervisor provides the processor with virtual-to-machine address translations for the executing software entity by servicing Translation Lookaside Buffer (TLB) misses generated by the processor. Alternatively, hardware virtualization support in Intel and AMD processors can be used to translate guest physical addresses to machine physical addresses, as detailed in [hpca]. For simplicity, we discuss only the shadow page table method in this paper.

4.2 Overview

To this baseline platform, we add security mechanisms at the hypervisor level and at the hardware level, to enable more resilient attestation and execution. We first give an overview of the operational flow. We then describe the software and hardware architectural mechanisms, and end the section with an example.

The operational flow involves the following phases:

- Secure launch of the hypervisor by the hardware
- Secure launch of critical software modules by the hypervisor
- Secure execution of these trusted modules, including:
    - response to attestation requests
    - authenticated transitions between modules
    - secure asynchronous transitions on interrupts
- Termination of trusted module execution

We make some key observations regarding achieving Tailored Attestation:

First, we must have evidence of a chain of trust, and must anchor this trust chain securely. We plan to do this by having a trust chain from the hyperviser to the trusted software modules (in the applications or OS), with secure hardware anchoring of the trust chain. launching the hyperviser securely with new hardware support, then using the hyperviser to launch trusted software modules. The rationale for secure launching is that one cannot gaurantee that software is executing in a secure state if it did not even start in a secure state.

Second, to prevent Time-of-Check-to-Time-of-Use (TOCTOU) attacks, one must protect the securely launched trusted software (hyperviser and protected software modules) during execution. This requires protecting its virtual memory, its physical memory, its interactions with untrusted modules, and its secure transitions to other trusted modules, and protecting it when interrupted.

We propose co-design of new features in the processor hardware and a trusted hypervisor to support Tailored Attestation efficiently. (We do not need a separate TPM chip.) New hardware and software security mechanisms are needed for the software launch phase and the runtime phase. New hardware mechanisms in the processor enable the secure launch of the trusted hypervisor, and runtime protection of the hypervisor's code and dynamic data. The trusted hypervisor then securely launches trusted software modules whenever requested during runtime, setting up appropriate memory protections. Our TCB provides finer granularity for attestation by having the hypervisor define protection for software modules that is enforced by the processor at runtime. A modified hyperviser provides virtual memory protection against software attacks. Machine memory integrity and confidentiality protection against physical attacks are provided by hardware memory protection mechanisms. Inter-module control flow integrity for critical code is provided by our Secure Module Transition mechanisms. A new attest instruction provides tailored attestation capability.

We utilize the modular nature of software systems to decompose the software stack into OS and application modules. Examples of a module are a subroutine and its private data space, a library of functions, a class in an object-oriented programming language, a plug-in application extension or a loadable OS module. This natural form of decomposition can be leveraged to describe functional attestation via the attestation of a set of modules, without imposing a new programming model.

In this paper, we focus on architecture for resilient attestation and execution. Identifying security-critical modules in an application or OS, and verifying the correctness and lack of vulnerabilities in these security-critical modules are studied elsewhere, e.g. [34, 35], and out-of-scope for this paper.

Figure 2 summarizes the new or enhanced software and hardware components in our TCB (in the hypervisor or the processor hardware). The vertical sections represent the different functions provided.

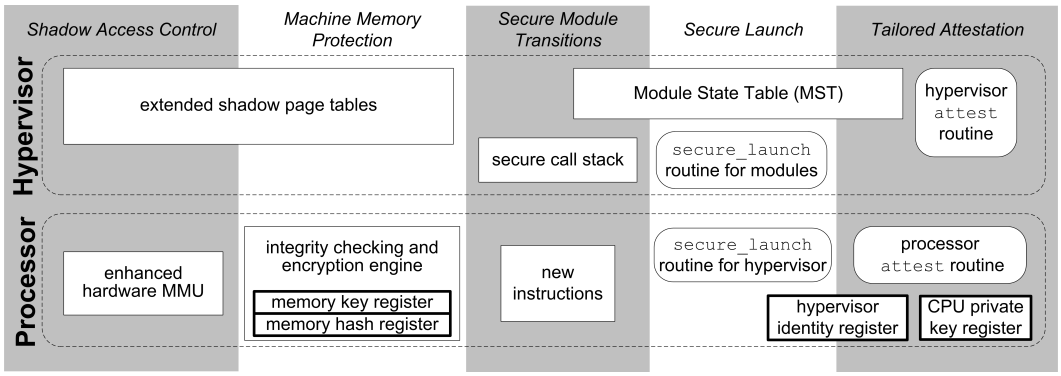

Fig. 2. Software and Hardware Components in our Trusted Computing Base

4.3 Runtime Protection and Monitoring

The key security advantage of our proposed attestation solution compared to TPM's attestation is the runtime protection and monitoring that we provide. Hence, we discuss this first, before discussing Secure Launching of the trusted software modules.

4.3.1. Virtual Memory Protection

A hypervisor (or Virtual Machine Monitor) gives each guest operating system full control over its guest physical memory space. The virtual memory manager of a malicious guest OS can thus snoop on or corrupt a critical application module by reading or writing a guest physical page where the module was loaded. It can also allow malicious applications to attack the critical module by mapping the guest physical page in the virtual memory space of these applications. Our new Shadow Access Control mechanism prevents such attacks by enhancing the shadow page table mechanism so that, in addition to mapping guest physical pages to machine memory pages, it binds each page to a module. This binding, enforced by our enhanced processor hardware, ensures a page can only be accessed by its module, except when the page is explicitly defined as a sharing interface between several modules. As a result, *Shadow Access Control provides fine-grained isolation of modules from one another within a Virtual Machine*, without having to rely on the virtual memory manager of the guest OS.

To bind a page to a module, Shadow Access Control extends shadow page table entries with a field containing the module's module_id. Upon a memory access, the processor looks up an address translation in an entry of the Translation Lookaside Buffer (TLB)—a hardware structure caching the (shadow) page table entries. In parallel, it compares the module_id of the TLB entry with that of the software module requesting the access[1]. If the identifiers match, the access is granted; otherwise, access is either denied or restricted, depending on the requestor. A guest OS is given access to an encrypted and tamper-evident version of the page while an application is simply denied access. Restricted access is given to guest operating systems to ensure they can perform paging or relocate pages in physical memory. Sharing of

[1] This is the `module_id` of the page storing the memory access instruction.

memory pages between modules is achieved by having one shadow page table entry for each module sharing the page.

4.3.2. Machine Physical Memory Protection

Shadow Access Control can protect a critical application or OS module against malicious memory accesses from the rest of the software stack. However, it does not defend against attackers with physical access to the platform or against a malicious OS accessing module state via DMA or the on-disk page file. To protect module memory against these attacks, we propose an *enhanced memory integrity tree* [9, 22] for memory authentication, and cache-line-based memory encryption for confidentiality. Anchored in hardware and managed by the hypervisor, our tree can selectively include or exclude specific memory or disk pages from tree coverage in order to protect critical modules despite multiple untrusted operating systems. Our solution provides protection at a page granularity, coverage of disk pages, and applicability to virtualized software stacks. In addition, our integrity checking engine generates a non-fatal exception when it detects corruption in non-hypervisor memory. The hypervisor handles this exception by stopping affected modules and flagging them as corrupted, but it allows forward progress for unaffected modules. This enables our solution to achieve resilient execution and resilient tailored attestation. Other than these minor novelties, we leverage the vast amount of work done on memory integrity trees, based on Merkle trees, of which [20, 4, 27] are just a small representation. It recursively computes cryptographic hashes on the protected memory blocks until a single hash of the entire memory space, the tree root, is obtained. The tree root is kept on-chip in the microprocessor, while the rest of the tree can be stored off-chip.

To enable protection at a page granularity, the hypervisor assigns a new i_bit to each machine memory page. This bit tells the hardware integrity checking engine whether a page is to be covered by the integrity tree. When the bit is '1', accesses to the page trigger a hash verification on the data read or written off-chip. When the bit is '0', accesses to the page proceed without any integrity verification. Together with Shadow Access Control bindings, the i_bit is stored in the shadow page tables and made available to the processor hardware via the TLB unit.

All pages belonging to critical modules have their i_bit set to '1' to enable runtime integrity checks: any corruption of a critical module's memory space between launch time and attestation time is detected by the on-chip integrity checking engine. When this engine detects memory corruption in a hypervisor page, it resets the platform, since the TCB has been corrupted. If the corruption occurs in an OS or application module however, the engine notifies the hypervisor, which then flags the modules associated with that page as corrupted and prevents forward progress for these modules. Flagging corrupted modules individually enables resilient attestation of uncorrupted modules.

To protect the confidentiality of critical modules, we implement a cache-line-based memory encryption mechanism. Any cache line of a confidential page is encrypted when written off-chip, by the memory encryption engine in the microprocessor chip. It is decrypted when fetched from external memory. Critical modules can thus protect the confidentiality of their runtime data.

4.3.3. Secure Module Transitions

We provide both secure *synchronous* module transitions and secure *aynchronous* transitions. New instructions are added to make explicit the synchronous transition to and from a critical module, allowing a remote party to analyze inter-module control flow integrity. We also protect module register state during the asynchronous transitions that occur when an interrupt preempts module execution to give CPU control to a guest OS, or when the guest OS decides to schedule a previously preempted module. We assume that the hypervisor wipes out any sensitive register state before handing CPU control to its VMs.

Synchronous Transitions

We define a secure synchronous module transition from module A to module B as a pair of code jumps, a call and a return, that must respect the following conditions:

1) the call must be initiated from an explicit calling site in module A,
2) the call must jump to an authorized entry point in module B's code,
3) the call must allow A and B to authenticate one another, and
4) B must only return to A's calling site and must do so using an explicit return instruction.

We define new call_module and return_module instructions which ensure conditions 1 and 4 are respected, while the new enter_module instruction fulfills conditions 2 and 3.

Our call_module instruction is used to jump from one module to another. It takes as arguments the module identity of the destination module[2] and a pointer to its entry point. The return_module instruction is used by the callee to return to the caller. As in past work on control flow integrity [16], the correctness of the return procedure is ensured by a trusted part of the platform: in [16] correctness was enforced by the trusted processor using a hardware secure return address stack (SRAS), while we enforce it with the trusted hypervisor using a call stack maintained in its protected memory space. Each entry of this stack matches the address of a call instruction with the module_id of the caller module and the module_id of the callee module. Upon a return, the hypervisor ensures that the callee module returns to the instruction following the calling site in the caller module.

The enter_module instruction allows the programmer to specify authorized entry points into module code and receive, from the hypervisor, the identity of the module which jumped to that entry point. To achieve the former, the hardware requires that any jump to a module other than module 0 land on an

[2] The argument is a pointer to an identity hash allocated in the caller module's static data.

enter_module instruction. This makes entry points into module code explicit and allows a remote party to ensure, during attestation, that authorized entry points conform to its security requirements. The hardware traps to the hypervisor on every enter_module instruction. This allows the hypervisor to keep track of executing modules so it can pass to the callee module the identity of the caller module. This is done through a pointer specified in an operand to the enter_module instruction. The callee can thus determine whether it should ignore the invocation or return results to the caller.

Asynchronous Transitions

To protect the register state of a module upon preemption by a hardware interrupt, the hypervisor intervenes before CPU control is handed over to a guest OS. The hypervisor saves the contents of the module's registers into hypervisor memory space—where their confidentiality and integrity are protected—and wipes out the contents of the hardware registers. It keeps track of the currently executing module by monitoring the page tables used by guest operating systems and the value of the Program Counter (PC) register. When a module is resumed by the guest OS, the hypervisor intervenes to restore the register state of the module and ensures it resumes where it was initially suspended. Thus, the software that executes in between the suspension and the resumption cannot observe or corrupt the module's register state.

System calls are treated as a synchronous transition followed by an asynchronous transition. When an application invokes call_module with a module identity referring to an OS module, the hypervisor pushes the call on its call stack (as in a synchronous transition) and suspends the module as if a hardware interrupt (i.e. an asynchronous transition) had occurred. The OS then interprets the system call and jumps to an entry point in the critical OS module referenced by call_module. This OS module handles the system call and invokes return_module, which causes the hypervisor to pop its call stack and resume the execution of the application where it was suspended.

4.4. Secure Launch

There are two versions of Secure Launch. The hardware is responsible for securely launching the hypervisor. The hypervisor is responsible for the Secure Launch of critical OS or application modules. The main goals of the Secure Launch procedure are to measure the identity of a piece of critical software and to set up its runtime protection.

A new secure_launch instruction is invoked for the hypervisor launch. It is serviced by the on-chip Secure Launch routine in the microprocessor, which computes the identity of the hypervisor and stores it in a new hardware register called the "Hypervisor identity register" (fig. 2), where its integrity is protected by being within the security perimeter of the microprocessor chip. This hypervisor identity is a cryptographic hash taken over the hypervisor's code and data pages. The Secure Launch routine also sends

each of these pages to the on-chip integrity checking and encryption engine, where the initial integrity tree is computed over the hypervisor pages, which are also encrypted before being written back to memory.

To launch a critical OS or application module, a secure_launch hypercall is invoked, which is serviced in the hypervisor by a software-based Secure Launch routine. This routine computes the module's identity (a hash over module code and data) and sets up the module's runtime protections. In doing so, the routine must interpret the module's new *security segment*, referenced by an argument of the secure_launch hypercall. We define this new security segment as a data structure specifying the code and data pages composing the module, and the memory protections to be applied during runtime. It also describes how the module communicates with other modules using shared pages. The module identity computed during this procedure is stored, along with a hash taken over the security segment, within the hypervisor's protected memory space. The integrity of module identities and security segments is thus protected by the integrity tree covering hypervisor memory.

For each of the module's virtual pages, the security segment defines the module's access rights to the page, a pair of (i_bit, c_bit) bits, and a list of (module identity, RWX rights) bindings. The i_bit and c_bit specify whether a page must be covered by the integrity checking and encryption mechanisms. A *binding* specifies an outside module allowed access to the page with the Read, Write and/or eXecute rights specified. These bindings thus define sharing interfaces between modules, either within or across address spaces.

Pages with a c_bit set to '1' are sent to the on-chip encryption engine for encryption upon eviction from caches in the microprocessor chip, so that only ciphertext is written back to memory. Pages with an i_bit set to '1' are included in the identity measurement and added to the coverage of the integrity tree checking mechanism. The hypervisor's Secure Launch routine also assigns the next available module_id to the module and modifies the shadow page tables so that the module's pages are associated with its module_id, and modules authorized to share pages are given the access rights described in the security segment. From then on, the platform only allows modifications to this measured initial state via memory transactions allowed under Shadow Access Control.

The Secure Launch routine also takes care of the dynamic memory needs of each module. Based on information defined in the security segment, the hypervisor allocates each module its own stack, like the allocation of private stacks to each thread in a multi-threaded application. Similarly, the hypervisor reserves a private pool of virtual pages for the module's heap. The module can dynamically request a chunk of this heap memory from the hypervisor via a hypercall.

Table I shows the Module State Table (MST), a software data structure constructed and maintained by the hypervisor to keep track of critical module state. During the Secure Launch procedure, the

hypervisor writes the computed module identity and security segment hash in the MST, along with the module_id of the module being launched. During runtime, the bit indicating runtime corruption, initially set to '0', is set to '1' if the integrity checking engine detects corruption in one of the module's pages. When a module is preempted, the hypervisor's Secure Module Transition mechanisms described in Section 4.5 store the module's register state in the dedicated MST table entry. During Tailored Attestation, the hypervisor looks up the MST entry of every module it reports, to read its identity and determine whether it has been corrupted since launch.

Table I – Module State Table

module_id	*launchtime identity*	*SSH Security Seg Hash*	*runtime corruption?*	*saved register state*
id_1	$identity(id_1)$	$SSH(id_1)$	*yes/no*	$register\_state(id_1)$
id_2	$identity(id_2)$	$SSH(id_2)$	*yes/no*	$register\_state(id_2)$
id_3	$identity(id_3)$	$SSH(id_3)$	*yes/no*	$register\_state(id_3)$
⋮	⋮	⋮	⋮	⋮
id_x	$identity(id_x)$	$SSH(id_x)$	*yes/no*	$register\_state(id_x)$

4.5. Tailored Attestation

The platform's Tailored Attestation procedure is handled jointly by the trusted hypervisor and the processor. The goal of the procedure is to report a set *S* of critical modules to the remote party. It also gives the remote party a way to securely communicate with these modules, despite possible corruption in some other parts of the software stack.

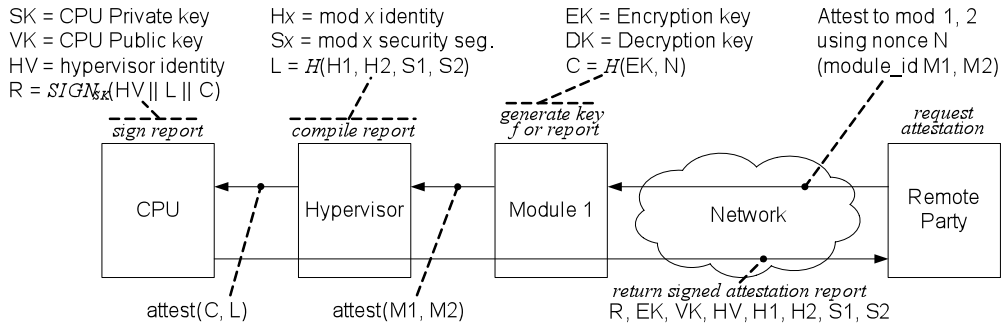

Fig. 5. Tailored Attestation

On receiving an attestation request, a module generates an asymmetric key pair to be used to establish a secure channel with the remote party. The module stores the private part of this key pair in an encrypted and integrity-checked page and creates a certificate for the public part of the key pair. It then invokes the hypervisor's attest routine via a hypercall to request a Tailored Attestation report binding the identities and security segments of the modules in *S* to the public key certificate it just generated and the identity of the hypervisor. The hypervisor compiles the report by looking up the Module State Table entries for the modules in *S* and then asks the processor, via the new attest instruction, to add the *Hypervisor identity* to the report and sign it with the private key in the *CPU private key* register (refer to Fig. 2, which shows the new Hyperviser identity and CPU private key registers).

Figure 5 illustrates the Tailored Attestation procedure. We describe this following the arrows from right to left, then left to right. An attestation request is made for critical modules 1 and 2 with a session nonce, N, for freshness. When critical module 1 receives this request for attestation, it first generates a public-private key pair (EK, DK) and creates a certificate C binding the public encryption key EK to the session's nonce N. The critical module then invokes the hypervisor, which creates a compounded identity L from the identities (H1 and H2) and security segment hashes (S1 and S2) of the modules requested. The processor's attestation function is then invoked. With its private key SK, the processor signs an attestation message binding the hypervisor identity HV to the combined module identities L and the certificate C. The signature R and the session's public encryption key EK are sent to the requester, along with metadata identifying the processor (VK, the processor's signature verification key), the hypervisor (HV, the hypervisor's identity), and the modules (their respective identities and security segments).

The requester uses a Public Key Infrastructure (PKI) to check that the platform's reply comes from a trustworthy processor and, using a local or online trusted database, verifies that HV, H1, and H2 correspond to trustworthy versions of the hypervisor and the required security-critical modules. It also checks that S1 and S2 define runtime protections in line with the requester's security requirements. When these checks are successful, the requester can trust that 1) the processor protects hypervisor execution, and 2) the TCB protects the two critical modules specified, 3) the transitions between the critical modules are checked (inferred from the position of the call_module and enter_module instructions).

With the attestation report, the remote party can be confident that a secure channel established using the session's public encryption key EK is only accessible to the protected modules reported during Tailored Attestation. Additional information required by the remote party to make a trust decision can also be included in the certificate created by the critical module. Tailored Attestation reports can be requested multiple times during a critical module's execution, e.g. before executing a critical module, to establish new key material, to establish configurations, to verify that a particular set of modules is still alive and protected, or to attest to data results produced.

This paper focuses on protecting and attesting to the runtime state of critical modules and does not discuss the protection of their persistent state (e.g. an on-disk file containing long-lived keys). The design of our TCB does include a mechanism binding an integrity-checked and encrypted persistent file to the hypervisor and to each module. Two on-chip registers store a hash and encryption key protecting the hypervisor's file (not depicted in fig. 2). This hypervisor file in turn contains the hash and key pairs protecting the files of critical OS and application modules. While this secure storage mechanism is useful in many application scenarios, it is not discussed further due to space limitations.

4.8 Example Scenario

Consider an internet telephony service, P2Pcall, where client devices participate as nodes in a peer-to-peer network forming a distributed routing infrastructure. Users can place calls and chat with contacts using the service's Voice-over-IP (VoIP) application. The application runs a Secure Routing Module (SRM) which accesses the P2Pcall network to inject the user's outgoing VoIP packets, extract the packets directed to the user, and route packets of other P2Pcall users. P2Pcall allows trustworthy SRMs to access the peer-to-peer network by giving them a session key K with which they can open secure communication channels with nearby nodes. The P2Pcall security policy defines a trustworthy SRM as a module routing calls according to the P2Pcall routing algorithm, without modifying any call packets. Availability of a given SRM is not a concern for the server, as the P2Pcall routing algorithm can circumvent unresponsive nodes without dropping call packets.

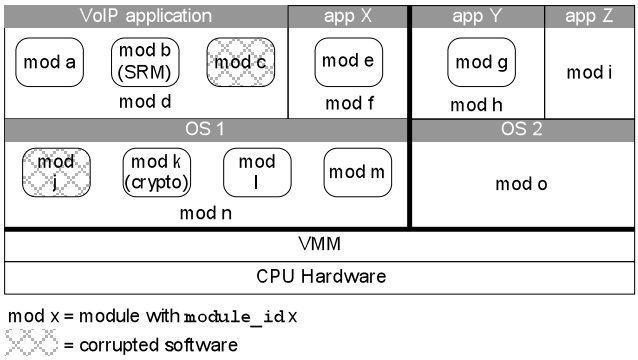

Fig. 6. Software Stack with VoIP Application

Figure 6 shows the software stack containing this VoIP application, with a voice acquisition module (mod a), an SRM (mod b), a chat module (mod c) and a main module (mod d). The SRM uses an OS-based library of cryptographic functions (module k) to establish and maintain secure channels. On receiving a Tailored Attestation request, the platform reports to the P2Pcall server the identity of the hypervisor, the SRM and the OS-based crypto module.

The operating system's network stack, a rather large and complex software system, is not included in the attestation report, as it only performs a benign task: relay network packets, which have an encrypted and authenticated payload, to and from the SRM. In the worst case, a compromised network stack merely drops the packets or corrupts them in a tamper-evident way. This results in a denial of service for the affected node. Similarly, the system call handler (not included in attestation reports) could be compromised and carry out any combination of the following malicious actions: 1) ignore the SRM's requests to invoke the crypto module, 2) drop the crypto module's replies to SRM requests, 3) corrupt the crypto module's replies (triggering an integrity tree verification error). In any case, the SRM is unable to generate properly encrypted and authenticated outbound payloads (or decrypt and authenticate inbound

payloads), once again resulting in a denial of service for this particular node. As stated in our threat model, the architecture we propose does not try to address denial of service attacks.

The SRM defines a public-private key pair for the current attestation. The public part EK of this key pair is used by the server to encrypt the session key, which the SRM can decrypt using DK, the private part of the key pair. Since the decryption key DK is stored in the SRM's protected module memory space, the server can also be confident DK will only be used by the SRM. The server can thus use the public encryption key EK bound to the attestation message to encrypt the session key K. K can then be used by the SRM to open secure communication channels with nearby P2Pcall nodes. The certificate C produced by the SRM could also include a configuration data file (e.g., specifying the protocol parameters or packet format to be used on the P2Pcall network) in case the server needed this information to make its trust decision

The TCB presented in this paper does not only apply to VoIP. It can support a variety of security policies in different application scenarios. Our hardware-software mechanisms are flexible enough to allow a remote party to define its own security policy and verify, through Tailored Attestation, that it is properly enforced by the TCB. For example, a media distribution company can check that users run a Digital Rights Management (DRM) module in their media player[3]. An e-commerce website or a bank may require that customers use a secure transaction module isolated from the rest of the browser. A military organization may want to verify that the file system manager implements a Multilevel Security (MLS) policy in their computers deployed in the field.

5. SECURITY ANALYSIS

In analyzing the security of the Tailored Attestation architecture, this section first looks at hypervisor protection and then examines the different phases of critical software execution: launch, runtime, transition and attestation.

5.1 Hypervisor Protection

To make the Tailored Attestation procedure trustworthy, the architectural roots of trust are contained within the CPU chip—our security perimeter and the hardware component of the Trusted Computing Base (TCB). Trust for measurement and reporting of the trusted hypervisor (the software part of the TCB) is established by the combined effect of the Secure Launch routine, the Tailored Attestation routine and two hardware registers storing the hypervisor measurement (Hypervisor Identity register) and the CPU signing key (CPU Private Key register) for signing attestation reports.

The two routines, for measurement and for reporting, are stored within the CPU chip (to prevent physical attacks), in Read-Only Memory (ROM) so they cannot be modified after manufacturing. For this paper, we assume the CPU private key is generated by the (trusted) manufacturer and written to a

[3] This requires a protected display device driver module able to establish a secure communication channel with trusted audio-video hardware.

dedicated write-once CPU register during chip fabrication; this register is only accessible to the on-chip attestation routine. Other private key initialization schemes such as those supported by the TPM [29] could be implemented to support different trust models.

On every chip reset, the symmetric key for memory encryption is generated anew using an on-chip random number generator, and the root hash of the memory integrity tree is recomputed by the secure launch routine. Because they are stored in dedicated on-chip registers, these two hardware roots of trust for memory protection are inaccessible to software and physical attackers. The regeneration of the memory encryption key upon processor reset ensures that knowledge of (plaintext, ciphertext) pairs in one power cycle[4] does not give an attacker information about encrypted memory contents in another power cycle. The regeneration of the root hash of t memory integrity tree for the hypervisor's memory space ensures that memory contents cannot be replayed across power cycles.

Hence, new hardware registers and mechanisms protect the hypervisor and its memory space, and it in turn protects critical OS and application modules.

5.2 Secure Launch

As opposed to platforms that carry out a *secure boot* (where only signed pieces of software can be launched, e.g. [18, 17]), the architecture presented in this paper performs a *measured boot* (as in [29, 27, 24]), where critical software is precisely measured upon launch, so it can be reported during attestation and analyzed by the recipient of the attestation report. The security of the measured boot, carried out by the Secure Launch procedure, thus depends on 1) precisely measuring the identity of software being launched and 2) protecting the integrity of this identity measurement until attestation.

In this paper, the attestation report on a critical software module not only fingerprints the bits forming its initial state (with the identity hash), but also describes the protection provided to the software during runtime (with the security segment hash). This protection information allows recipients of attestation reports to reason about whether software can respect certain security requirements when it runs in an adversarial environment. Indeed, the security segment hash reflects the memory page encryption and hashing bits (c_bit and i_bit) and the precise definition of memory interfaces to be authorized during runtime.

As with any bit of the software component's identity, the security segment can be corrupted by an attacker with access to the binary file storing the component on-disk. This is not a concern since the platform is performing a measured boot, where the recipient of the attestation report can choose not to entrust a software component with critical data in the case of an unexpected security segment hash. To protect the integrity of identity and security segment measurements (hashes), the Secure Launch

[4] A power cycle to be the period of time between two CPU reset events.

procedure stores them in either a dedicated CPU register inaccessible to software (hypervisor identity register) or within the hypervisor's protected memory space, in the Module State Table (for module measurements).

5.3 Secure Execution

Protecting the runtime state of a software module (or the hypervisor) consists in providing strict isolation for certain parts of its code and data space, while enforcing restricted interfaces to surrounding software for other parts of its memory space.

Strict Isolation

Aside from exceptions discussed below, a module's code space is static and must be protected by strict isolation. This prevents corruption (through malice or malfunction) by other modules within the same address space, by the OS or by physical attacks. To prevent software attacks, the hypervisor overrides guest OS memory management: the hypervisor marks a machine page containing critical code as read-only in its shadow page tables. To thwart corruption of critical code by the OS during paging, by a rogue DMA transaction or by a physical attack, the integrity tree detects corrupted code and prevents that software module from executing. The hypervisor code itself is protected by the integrity tree and by the processor's hypervisor mode (negative ring). Static data are protected with the same mechanisms as static code.

Typically, most of a module's data space also requires strict isolation: it must be unavailable to surrounding software, but remain accessible to the module's code for reading and writing. Shadow Access Control achieves this by binding code and data pages to a unique module_id and having the hardware enforce the following rule: data page bound to module_id X can only be accessed by code from a page bound to module_id X. Similarly, dynamically-generated or self-modifying code in module Y can be located in pages assigned module_id Y that are accessible for reading, writing and execution. The recipient of an attestation report for such code decides whether to trust module Y by analyzing the static routine generating the dynamic code or the initial state of the self-modifying code (e.g., to check that they cannot generate runtime code that would overwrite critical data structures).

Confidentiality of hypervisor and module data is ensured in part by Shadow Access Control since pages with confidential data can be made accessible only to specific software modules. In addition, setting a page's c_bit to '1' ensures it is encrypted when it leaves the microprocessor chip, thus protecting its confidentiality against attacks that target machine memory directly. This also keeps a module's page private while it is being sent to disk by the operating system's paging mechanism. To protect page integrity during paging, the hypervisor detects paging events as part of its maintenance of the shadow page tables and updates the integrity tree so it keeps track of pages that reside on disk. A rogue OS might somehow hide paging events to relocate pages within the guest memory space without the hypervisor's knowledge.

The on-chip integrity checking engine considers this as corruption since the integrity tree no longer reflects the prevailing position of pages in memory. The effect is thus a successful Denial of Service (not addressed by this paper) to the protected module but an unsuccessful attempt at breaching its integrity.

Restricted Sharing Interfaces

A software module is impractical unless is has access to inputs from outside its private memory space and can output results to surrounding software. We support critical module communication with the outside world using restricted memory interfaces shared either with other protected modules or with unprotected code. To share a page between two modules, the hypervisor creates two shadow page table entries bound to two different module_id's but pointing to the same machine page. These sharing interfaces are only created based on directives in the security segments, which are included in attestation reports.

The hypervisor disallows sharing in the case of conflicting directives so the remote party can be confident that a given module only communicates with the outside world through the reported interfaces. When a critical module is interfaced with another protected module, the remote party must decide—based on the attestation report—whether to trust this other module. When a critical module is interfaced with an unprotected module (which cannot be trusted), the remote party must determine whether the critical module validates its inputs and filters its outputs according to its security requirements. This restricted interface mechanism can be used to dynamically link code libraries to existing modules. To do so, the hypervisor assigns a different module_id to the library and sets up a sharing interface between the library and the existing module.

5.4 Secure Module Transitions

Secure execution of a software module also includes secure synchronous and asynchronous transitions into and out of critical module code. These transitions to a critical module of interest can occur from another protected module or from an unprotected module. For transitions from a critical module, the hypervisor's intervention on interrupts and its handling of the call_module instruction ensure the register state of the source module is protected and that the destination module is the one identified by the instruction's operand. For transitions into a critical module, the enter_module instruction allows the module to authenticate its caller and, for example, decline to process inputs coming from a blacklisted module. During attestation, these special instructions are reported as part of the module's code base. A remote party can thus determine whether the inbound and outbound module transitions permitted by the platform conform to specific security requirements.

No security guarantees can be provided for transitions into and from code that is not measured or provided with runtime protection by the TCB. A transition to a specific virtual address within an insecure module (with module_id 0) may effectively land in an arbitrary piece of code since the TCB does not check for

malicious swapping of two code pages in unprotected modules. This must be taken into account by the remote party when defining the set of modules requiring TCB protection.

In the worst case, transitioning into a piece of corrupted code should only result in a denial of service. For example, if the untrusted and unprotected hypervisor loader does not jump into hypervisor initialization code, the hypervisor's Secure Launch procedure is not invoked and the hypervisor is not measured. This condition is easily detected by a remote party since the platform refuses to perform Tailored Attestation without a measured hypervisor. Unprotected code should only be enlisted to carry out tasks that are not security-critical; it is code that might not behave as expected when carrying out the task or jumping back to critical code. Defining a methodology for partitioning applications and operating systems into modules that are either security-critical or non-critical to a given task is ongoing research work.

5.5 Trustworthy Tailored Attestation

The mechanisms just discussed ensure that the measurements reported during Tailored Attestation specify the initial state of critical modules, authorized memory interfaces and memory protection. By this reporting, the platform gives the remote party a guarantee that the data dynamically generated by the module, as well as the interactions it initiates with surrounding software, respect the module programmer's intention. In other words, when the program is properly protected, it behaves as can be expected from inspection of its source code.

This paper focuses on attacks originating from outside a critical module: it does not consider software vulnerabilities located inside a critical module's code and that could thwart the remote party's security objectives. The remote party is responsible for ascertaining that a critical software module is free of vulnerabilities. This should be a feasible task for small modules with a few thousand lines of code. An interesting avenue for future research is to define a form of attestation that allows a remote party to determine whether a large piece of code (that is trusted but potentially contains exploitable software vulnerabilities) has been compromised in a way that affects the remote party's security objectives.

The certificate generated by a critical module during Tailored Attestation binds the module's identity with a public encryption key for further secure communications, as well as additional data that might be required for the remote party to make a trust decision (e.g., the contents of a configuration data file). The CPU signs an attestation report containing this certificate, critical module identities, hashes of their security segments, the hypervisor identity and a nonce provided by the remote party. This report provides the remote party with the (authenticated) information needed to make a trust decision. The nonce ensures that runs of the protocol cannot be replayed. The hypervisor identity and CPU signature identify the TCB (hypervisor and CPU hardware) executing Tailored Attestation to the requester.

6. PERFORMANCE AND COST

The main impact on performance comes from the runtime machine memory protection mechanisms. As shown in previous work, the performance of the integrity tree can be significantly enhanced by caching its nodes [9] to speed up tree operations. Also, the simple hashing primitive can be substituted for one with better performance characteristics [22]. The performance degradation has been shown to be acceptable—under 5% when optimized for reduced latencies [22]. Memory overhead for the storage of integrity tree nodes has been evaluated to be approximately 25% in [9, 22]. With the decreasing cost of memory storage, this overhead should be considered acceptable on general-purpose computing platforms.

The new hardware registers and state added to the microprocessor consist of four new registers, extra TLB entry fields, L2 cache flags, and a routine ROM. Extra logic is also needed for the encryption and integrity checking engines, memory tree management, for enforcing Shadow Access Control, and for implementing new instructions.

Two new processor registers contain the processor's private signature key and the hypervisor identity, both used during attestation. A third register contains the root of the integrity tree, while a fourth stores the symmetric key used to encrypt memory. The Secure Launch and Tailored Attestation routines are stored in the routine ROM.

The cost of these hardware additions is less than that of previous secure computing proposals providing runtime memory protection [18, 27]. The bulk of the hardware overhead consists in the machine memory protection mechanisms, which have been evaluated to require less than 200K gates [28], a small overhead in chip area for modern microprocessors.

Impact on Software

The approach we propose in this paper can support legacy operating systems and applications by having the hypervisor include all of their code and data in a single module with module_id 0. A critical module can also be integrated into a legacy OS or application with a simple wrapper invoking the call_module, enter_module and return_module instructions that allow the hypervisor to monitor the execution flow in and out of the module. Since the hypervisor must be able to track paging operations, operating systems that manage memory in unconventional ways may need to be adapted.

Our approach does not require significant changes to the way critical modules are written and compiled. A new "confidential" data type can tell the compiler when to set a page's c_bit to '1' in the module's security segment. The main change for a programmer consists in using the new instructions to invoke the Secure Launch procedure, ensure secure module transitions and perform Tailored Attestation.

7. CONCLUSION

This paper introduced the concept of Tailored Attestation, a resilient and functional form of attestation. We presented the design of a Trusted Computing Base formed of processor hardware and a trusted hypervisor, both enhanced to enable resilient attestation and execution of security-critical modules, even in the presence of corrupted code in unrelated parts of the software stack, in a sea of untrusted legacy software. Our TCB includes hardware-software mechanisms that securely launch critical software modules, protect their runtime state against software and physical attacks and can compile a customized and authenticated report of software identities for a remote party requesting a Tailored Attestation. Central to the TCB is the Shadow Access Control mechanism which leverages hypervisor mechanisms to provide fine-grained isolation of critical modules within operating systems and applications running in Virtual Machines. By reporting only a small fraction of the software stack, Tailored Attestation makes it easier for remote parties to make more accurate trust decisions. Integration of our security mechanisms in the processor hardware also ensures Tailored Attestation is much faster than attestation using a TPM chip.

REFERENCES

[1] K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization", in Proc. of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct. 2006.

[2] AMD, "Industry Leading Virtualization Platform Efficiency", www.amd.com/virtualization, 2008.

[3] P. Barham *et al.*, "Xen and the Art of Virtualization", In Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP), Oct 2003.

[4] M. Blum *et al.*, "Checking the correctness of memories", In Proc. of the 32nd annual symposium on Foundations of computer science, p.90-99, Sept. 1991.

[5] D. Champagne *et al.*, "The Reduced Address Space (RAS) for Application Memory Authentication", In Proc. of the 11th international conference on Information Security, LNCS vol. 5222, pp 47-63, Sept. 2008.

[6] X. Chen *et al.*, "Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems", In Proc. of the Conference on Architectural Support for Programming Languages and Operating Systems, March 2008.

[7] J. Dwoskin and R. Lee, "Hardware-rooted Trust for Secure Key Management and Transient Trust", Proc. of 14th ACM Conf. on Computer and Communications Security, pp. 389-400, Oct. 2007.

[8] T. Garfinkel *et al.*, "Terra: A virtual machine-based platform for trusted computing", in Proc. of SOSP 2003, pages 193–206, Oct. 2003.

[9] B. Gassend *et al.*, "Caches and Merkle Trees for Efficient Memory Authentication", Proc. of the 9th Int'l Symposium on High-Performance Computer Architecture, February 2003.

[10] T. Gilmont *et al.*, "Hardware security for software privacy support", Electronics Letters, 35(24):2096-2097, Nov. 1999.

[11] IBM, "Introduction to the Cell multiprocessor", IBM Journal of Research and Development, Volume 49, No. 4/5, pages 589-604, July 2005.

[12] Intel, "Intel® Virtualization Technology: Hardware Support for Efficient Virtualization", Intel Technology Journal, Vol. 10 (3), Aug. 2006.

[13] Intel, "Intel® Trusted Execution Technology: Preliminary Architecture Specification", http://www.intel.com, August 2007.

[14] Intel, "Intel Centrino Pro and Intel vPro Processor Technology", Intel Whitepaper, 2007.

[15] T. Jaeger *et al.*, "PRIMA: policy-reduced integrity measurement architecture", In Proc. of the 11th ACM Symposium on Access Control Models And Technologies (SACMAT 2006), pp 19-28, 2006.
[16] R. B. Lee *et al.*, "Enlisting hardware architecture to thwart malicious code injection", Proc. of the 2003 International Conference on Security in Pervasive Computing, 2003.
[17] R. B. Lee *et al.* , "Architecture for Protecting Critical Secrets in Microprocessors," Proc. of the 32nd International Symposium on Computer Architecture, pp. 2-13, June 2005.
[18] D. Lie *et al.*, "Architectural Support for Copy and Tamper Resistant Software," Proc. of the 9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pp. 168-177, 2000.
[19] J. M. McCune *et al.*, "Flicker: An Execution Infrastructure for TCB Minimization", In Proc. of ACM European Conference on Computer Systems (EuroSys2008), March 2008.
[20] R.C. Merkle, "Protocols for Public Key Cryptosystems," IEEE Symposium on Security and Privacy, pp. 122-134, 1980.
[21] M. Peinado *et al.*, "NGSCB: A Trusted Open System", In Proc. of the 9th Australasian Conference on Information Security and Privacy, July 2004.
[22] B. Rogers *et al.*, "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly", In Proc. of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO2007), pp. 183-196, Dec. 2007.
[23] A.-R. Sadeghi, C. Stüble, "Property-based attestation for computing platforms: caring about properties, not mechanisms", In Proc. of the 2004 workshop on New security paradigms, Sept. 2004.
[24] R. Sailer *et al.*, "Design and implementation of a TCG-based integrity measurement architecture", In Proc. of the USENIX Security Symposium, 2004.
[25] R. Sailer *et al.*, "sHype: Secure hypervisor approach to trusted virtualized systems", IBM Research Report RC23511, Feb. 2005.
[26] J. Sugerman *et al.*, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", In Proc. of the General Track: 2002 USENIX Annual Technical Conference, pp.1-14, June 2001.
[27] G. E. Suh *et al.*, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," Proc. of the 17th Int'l Conf. on Supercomputing (ICS), 2003.
[28] G. E. Suh, "AEGIS: A Single-Chip Secure Processor," PhD thesis, Massachusetts Institute of Technology, 2005.
[29] Trusted Computing Group, "Trusted Platform Module (TPM) Main – Part 1 Design Principles," Specification version 1.2, Revision 94, March 2006.
[30] J. Yang and K. G. Shin, "Using Hypervisor to Provide Data Secrecy for User Applications on a Per-Page Basis," In Proc. of the ACM Virtual Execution Environment (VEE'08), March 2008.
[31] R. Ta-Min, L. Litty and D. Lie, "Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable," In Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006), November, 2006.
[32] A. Mockus, R. Fielding and J. Herbsleb, "Two case studies of open source software development: Apache and mozilla", ACM Transactions on Software Engineering and Methodology Vol. 11 (3), pp. 1–38, 2002.
[33] A. Ozment and S. E. Schechter, "Milk or Wine: Does Software Security Improve with Age?" in Proc. of 15th USENIX Security Symposium, pp. 93-104, Aug. 2006.
[34] J. A. McDermid and David J Pumfrey, "Assessing the Safety of Integrity Level Partitioning in Software", Proc. Of the 8th Safety-Critical Systems Symposium, Southampton, UK, pp. 134-152, 2000.
[35] H. Chen , D. Wagner, MOPS: an infrastructure for examining security properties of software, Proceedings of the 9th ACM conference on Computer and communications security, November 2002.
[36] Champagne, D., Lee, R.B., "Scalable Architectural Support for Trusted Software", The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India, Jan 9-14 2010.