Efficient TCB Reduction and Attestation

Jonathan M. McCune, Ning Qu, Yanlin Li
Anupam Datta, Virgil D. Gligor, Adrian Perrig

March 9, 2009

CMU-CyLab-09-003

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Efficient TCB Reduction and Attestation

Jonathan M. McCune Ning Qu Yanlin Li
Anupam Datta Virgil D. Gligor Adrian Perrig
CyLab / Carnegie Mellon University

Abstract

We develop a special-purpose hypervisor called TrustVisor that facilitates the execution of security-sensitive code in isolation from commodity OSes and applications. TrustVisor provides code and execution integrity as well as data secrecy and integrity for protected code, even in the presence of a compromised OS. These strong properties can be attested to a remote verifier. TrustVisor only adds 5306 lines to the TCB (over half of which is for cryptographic operations). TrustVisor imposes less than 7% overhead in the common case. This overhead is largely the result of today's x86 hardware virtualization support.

1 Introduction

Current commodity operating systems and the majority of applications lack assurance that the secrecy and integrity of security-sensitive code and data are protected. The size and complexity of current commodity operating systems and applications suggest that we will not achieve the level of assurance necessary to run security-sensitive code and data on these platforms in the near future. Yet, commodity platforms offer unmatched incentives for both casual users and developers, and hence will remain a dominant presence in the marketplace.

We develop a secure hypervisor, called TrustVisor, to provide a safe execution environment for security-sensitive code modules *without* trusting the OS or the application that invokes the code module. TrustVisor protects security-sensitive code and data on untrusted commodity platforms from malware, e.g., kernel-level rootkits. More specifically, TrustVisor is designed to protect the integrity and execution of security-sensitive code, and confidentiality and integrity of the data used by that code; as well as attest these properties to remote entities.

Our system enables many applications, but is particularly well suited when the security-sensitive code is small and self-contained. For example, a webserver serving SSL/TLS-based connections needs to protect its private signature key. If the OS or application is compromised, the attacker can steal the private key and impersonate the web server. In our system, the code accessing the private key can be confined to a self-contained code module protected by TrustVisor, and the http server code can call that code module for setting up SSL/TLS connection state. In this setting, even if the OS or http server are compromised, the secrecy of the private key is guaranteed, as well as the integrity of the code that is allowed to access it.

The triple design goals of verifiable application-level security properties, performance, and binary-level backward compatibility are achieved through a novel combination of HW- and SW-based mechanisms for *virtualization* and *integrity measurement*, with integrity measurement subsequently enabling *attestation* and *data sealing*.

TrustVisor leverages commodity x86 hardware virtualization with support for a device exclusion vector and nested page tables [2] to enforce MMU-based protection of itself and application-level

security-sensitive code and data from the operating system, other applications, and malicious DMA-capable peripherals (e.g., malware such as rootkits that exploit software vulnerabilities in the OS or applications, or DMA writes via Firewire peripherals). This design choice reduces the size of TrustVisor's code base and interface and brings it within the realm of formal verification techniques. The nested paging implementation is the only performance overhead imposed on non-sensitive code execution in this system.

TrustVisor employs a two-level approach to *integrity measurement*. This mechanism enables *attestation* of the code and execution integrity properties of security-sensitive codeblocks (SSCBs), and also enables these code blocks to *seal* data by encrypting it such that decryption is only possible if the right set of integrity measurements are present. The first-level mechanism that we employ is a system's physical TPM chip, and the second level is implemented in software in the form of a novel "micro-TPM" (μTPM) that is part of TrustVisor.

Our μTPM executes as part of TrustVisor on the system's primary CPU, thereby alleviating the adverse performance impact of frequent use of dynamic root of trust in hardware (cf. Flicker [15]). Other designs (e.g., vTPM [4], Overshadow [7]) include a commercial VMM in their TCB, thereby ruling out the possibility of formal verification with known techniques.

The main contribution of this work is the design and implementation of a comprehensive system that enables application developers to achieve strong security guarantees for their data and code executing on legacy platforms, and to prove those security properties to an external verifier. The small TCB, efficiency, and ease-of-use distinguish our approach from previous efforts.

2 Problem Definition

2.1 Design Goals

We seek to enable the isolated, attestable execution of a security-sensitive codeblock (SSCB) on commodity systems. Our goal is a system with the following properties:

P1 Protection. Provide fine-grained protection of code integrity, data secrecy and integrity, and execution integrity. *Code integrity* is the property that executable code is unmodified. *Execution integrity* is the property that measured code P actually executes with inputs P_{inputs} and produces outputs $P_{outputs}$.

P2 MinTCB. Reduce the Trusted Computing Base (TCB) not just with respect to code size but also with respect to the size of its interface, to facilitate formal verification and to ensure that the TCB provides the required properties (although such verification is beyond the scope of this paper, Section 6.2 presents a brief sketch of an approach).

P3 Attestation. Enable remote verification of property P1 while achieving P2: (1) provide attestations covering only the code relevant to a particular application, (2) avoid leaking information about software outside the TCB, and (3) enable light-weight verification by an external entity.

P4 Performance. Provide high performance to minimize the overhead on the legacy system when security-sensitive operations are *not* active, and dramatically reduce the overhead by the protection and verification (attestation) mechanisms for security-sensitive code compared to TPM-intensive solutions (e.g., Flicker [15]).

P5 Compatibility. Maintain binary compatibility with existing legacy OSes and applications (i.e., no change is required to the legacy code to run as an unprivileged guest), and minimize the porting effort for privilege-separated programs to leverage TrustVisor to protect the execution of their security-sensitive component.

2.2 Adversary Model

We distinguish between a local adversary and a network adversary, though the two may collude.

Local Adversary. We consider a local adversary with access to two significant system interfaces. First, we assume that the adversary can execute arbitrary code as part of the legacy OS. Second, the adversary can access the system's DMA-capable devices, e.g., Firewire interface. Thus, the adversary may be able to read or write secrets in memory without modifying the legacy OS.

We do not allow the adversary to perform physical attacks against the system's CPU, memory controller, main memory, Trusted Platform Module (TPM), or the busses that interconnect them.

Given the hierarchical privilege structure of legacy OSes, this model gives the adversary the ability to tamper with executing code in or on the legacy OS, both while it executes and when the relevant executable and configuration files are at rest in non-volatile storage. Common manifestations of these abilities are rootkits and Trojans.

This leaves us at the mercy of the adversary for the availability of many system services. However, we observe that today's adversaries are financially motivated and often prefer to keep machines online. Furthermore, the adversary does not have the ability to interfere with the operation of hardware virtualization features such as virtual machine control blocks (VMCBs), nested page tables (NPTs), and the device exclusion vector (DEV) that operate with higher privilege than the legacy OS.

Network Adversary. We adopt the standard Dolev-Yao threat model [10] for network communication, thus giving the network adversary the ability to block, inject, or modify network traffic between entities in our system. However, the adversary cannot break cryptographic primitives.

3 TrustVisor Design

We first present an overview of the TrustVisor architecture (Section 3.1). We then focus on mechanisms to protect code and execution integrity (Section 3.2), and data integrity and secrecy (Section 3.3). We describe the μTPM (Section 3.4), the integrity measurement process (Section 3.5), and the attestation protocol to allow a remote entity to verify that TrustVisor is invoked and protections are active (Section 3.6).

3.1 TrustVisor Design Overview

A primary goal of this work is to enable the execution of self-contained (e.g., statically linked) security-sensitive codeblocks (SSCBs) in total isolation from a legacy OS and DMA-capable devices. We further seek to provide facilities for remote attestation and the long-term protection of sensitive state, thereby achieving the Protection and Attestation properties (P1 and P3) from Section 2.1.

One example of an application where such capabilities are desirable is an SSL-enabled webserver, where the private SSL/TLS signing key is rendered inaccessible to all but a dedicated code module. This way, even if the remainder of the web server or the OS on which it runs becomes compromised, the secrecy of the private key is preserved. However, such an environment imposes considerable performance requirements on the isolation mechanism, as it must be able to sustain real-world levels of SSL/TLS session establishment (thereby achieving the Performance property P4).

A powerful mechanism supporting isolated execution of SSCBs is the creation of a dynamic root of trust enabled by modern TPM-equipped platforms [2, 12]. Unfortunately, today's dynamic root of trust mechanisms impose considerable performance overhead [15], and are not suitable for use on a busy server. We enhance this mechanism to further enable remote attestation of SSCBs loaded

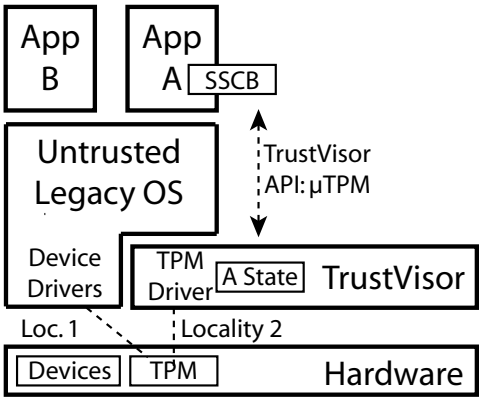

Figure 1: System architecture with TrustVisor. A TrustVisor-aware application can register a SSCB with TrustVisor, so that it will be executed in isolation from the untrusted legacy OS and applications. The untrusted legacy OS remains responsible for controlling the platform's devices. The only interface exposed to a SSCB by TrustVisor is that of a μTPM. The system's physical TPM is shared by TrustVisor and the untrusted OS using the TPM's locality mechanism.

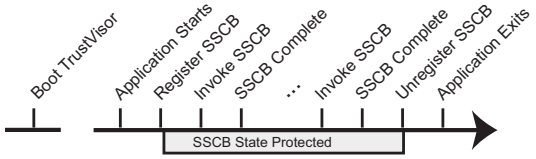

Figure 2: Basic sequence of events for a particular SSCB.

for execution (the Attestation property P3). We compare our system to contemporary alternatives in Section 7.

We wish to provide the security advantages of a system employing a dynamic root of trust mechanism, but without paying a severe performance penalty if the mechanism is heavily used. We further wish to maintain compatibility with available legacy operating system and application software, but exclude this software from the TCB. Our solution is to leverage available hardware virtualization support to provide memory isolation and DMA protection for SSCBs. We implement a small hypervisor called TrustVisor (Figure 1) to virtualize a machine's physical memory, enforce memory isolation between different SSCBs and untrusted code, protect against malicious DMA reads and writes, and enable remote attestation and long-term protected storage. We use dynamic root of trust only once per boot cycle to invoke TrustVisor. To distinguish between legacy code and SSCBs, we devise a registration mechanism by which untrusted applications can register certain code and data as security-sensitive. Figure 2 shows the timeline of an application registering a SSCB, and executing it one or more times.

To provide long-term protection of secrets required by SSCBs, and to enable remote attestation that a particular SSCB has executed (the Protection and Attestation properties P1 and P3), we design TrustVisor to also export a minimal TPM-like interface to SSCBs. We call this interface a micro-TPM (μTPM), and TrustVisor supports a μTPM instance for each registered SSCB.

TrustVisor has three basic operating modes, as illustrated in Figure 3: host mode, legacy guest

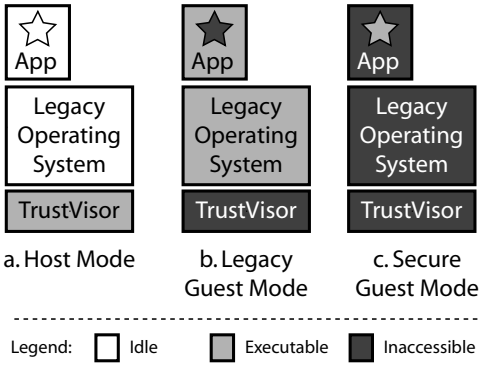

Figure 3: A comparison of the three execution modes of code with TrustVisor. In host mode, TrustVisor itself is executing. In legacy guest mode, either the untrusted OS or the untrusted portion of an application is executing. In secure guest mode, a registered SSCB is executing with TrustVisor's protections.

mode, and secure guest mode. *Host mode* refers to execution of TrustVisor code at the system's highest privilege level. TrustVisor in turn supports two guest modes. In *legacy guest mode*, a commodity x86 operating system and its applications can execute without requiring any awareness of the presence of TrustVisor. The legacy OS manages all peripheral devices on the system (network, disk, display, USB, etc.), with the TPM as the only shared device. TrustVisor accesses the TPM via its Locality 2 interface [22], and prevents the legacy OS from accessing this interface, opting instead to give the legacy OS access to the TPM's Locality 1 interface.[1] In *secure guest mode*, a SSCB executes in total isolation from the legacy OS and its applications. When a SSCB is registered, TrustVisor extends a measurement (cryptographic hash) of the SSCB into the first μPCR in the SSCB's corresponding μTPM instance. All input and output parameters are marshalled by TrustVisor into protected memory before the SSCB begins executing.

3.2 Code and Execution Integrity Mechanisms

TrustVisor enforces safety properties for SSCBs. We first describe how TrustVisor protects itself, then show how TrustVisor provides code and execution integrity for SSCBs (achieving the Protection property P1).

3.2.1 Hardware Memory Protections

The main function of TrustVisor is to isolate SSCBs from untrusted code by supporting both legacy guest mode and secure guest mode (Figure 3). However, in either mode, any code in the *guest* must not be allowed to compromise the code integrity of TrustVisor. Note that the legacy OS may also reprogram DMA-capable devices in a compromise attempt. To defend against these attacks, TrustVisor runs as a hypervisor at the system's highest privilege level. TrustVisor virtualizes the guest's physical memory to provide isolation between TrustVisor and the legacy system, and between

[1] TPM chips are memory mapped to multiple physical addresses, each corresponding to a different privilege level called a *locality*, where 4 is most privileged and 1 is least privileged [22]. TrustVisor can prevent the guest from accessing the Locality 2 – 4 interfaces by unmapping the relevant memory regions.

the legacy system and SSCBs. TrustVisor programs the device exclusion vector [2] (DEV) to protect itself and registered SSCBs against DMA accesses.

TrustVisor uses page tables to control MMU-based memory protections. We choose page tables, rather than other MMU-based protections (e.g., segmentation), because page tables are supported by many current CPU architectures.[2] To virtualize physical memory, TrustVisor maintains page tables that translate guest physical addresses into host physical (aka "machine") addresses. These page tables are located in TrustVisor's own memory region, thereby preventing any code in the guest from tampering with the page translation. Indeed, the code in the guest need not even be aware of this virtualization. To reduce complexity and keep the TCB small, TrustVisor leverages hardware virtualization with support for nested page tables (NPT) provided by recent AMD x86 CPUs[3] and enforced by the system's MMU.

This design enforces the Protection property (P1) even if a compromised legacy OS modifies its page tables (guest virtual address to guest physical address), since the guest physical address space is restricted to specifically *exclude* all of TrustVisor's sensitive memory regions.

3.2.2 Code and Execution Integrity for SSCBs

It is the responsibility of application developers to privilege-separate their programs into sensitive and untrusted portions.[4] The untrusted code can *register* sensitive code with TrustVisor, so that all memory regions (both code and data) corresponding to a particular SSCB will be isolated and measured by TrustVisor. The measurement of the SSCB is extended into the μTPM instance associated with this registered SSCB.

SSCB Registration. TrustVisor implements an application-level hypercall interface for registering SSCBs. The registration interface accepts a specification for all the functions provided by a SSCB, thereby allowing application programmers to specify sets of functions as security-sensitive. The specification includes a list of function entry points, and input and output parameter formats. This information allows TrustVisor to verify that the specified addresses are legal (i.e., they belong to the calling application's address space), and marshal and unmarshal the parameters between legacy mode and secure mode when the functions inside the SSCB are invoked.

During registration, the memory pages that correspond to the SSCB – including pages for the stack, heap, and other execution requirements for the SSCB – are protected by TrustVisor and become inaccessible to the untrusted OS, application, and even DMA requests from devices. It is at this point – "Register SSCB" in Figure 2 – that code integrity protections for the SSCB become active, and the Protection property (P1) is achieved. TrustVisor ensures that no other code on the system can tamper with this SSCB's memory pages. Further, the TCB includes only TrustVisor and the SSCB itself, thereby achieving the MinTCB property (P2).

Once the pages are inaccessible to untrusted code, TrustVisor computes a *measurement* (cryptographic hash) over the contents of these pages and *extends* it into the first μPCR of the μTPM instance associated with this registered SSCB. We defer detailed discussion of the handling of this measurement until we treat remote attestation in Section 3.5, but essentially this will enable an external verifier to check that registration happened correctly.

[2] Segmentation may also be viable for implementing the necessary protections. However, changes to the segment registers require changes to compilation settings, thereby impacting portability. Further, although an individual segment descriptor does not have the alignment restrictions of paging mechanisms, there are too few segment registers to cleanly support discontiguous SSCBs without frequent TrustVisor interposition.

[3] Intel offers similar support called Extend Page Tables.

[4] While automatic privilege separation may be possible in some instances [5], such mechanisms are beyond the scope of this paper.

SSCB Invocation. Following registration, the untrusted legacy application and underlying OS cannot read, write, or directly execute the memory containing the SSCB that it registered. However, the functions inside the SSCB can still be invoked using what appears to be an ordinary function call. Any function call to code inside the SSCB will trap to TrustVisor. TrustVisor then performs the following three steps before transfering control to the called function inside the SSCB:

1. Identify which registered SSCB contains the current called sensitive function.
2. Switch the guest from legacy mode to secure mode, with secure mode configured so that only the pages containing this SSCB are accessible.
3. Prepare the secure-mode execution environment (including marshalling input parameters from the untrusted application and setting up the SSCB's stack pointer) for the sensitive function call.

SSCB Termination. When a SSCB has completed executing and returns back to the calling legacy application, TrustVisor once again gets control. This happens because any attempt to execute code in secure mode outside the SSCB causes a trap into TrustVisor. TrustVisor performs the following two steps before transfering control back to the legacy application:

1. Marshall any returned parameters and make them available to the calling untrusted application.
2. Switch the guest from secure mode to legacy mode, in which the pages containing the SSCB are once again inaccessible from guest mode.

The SSCB's execution state is left intact, so that the corresponding untrusted application can invoke it a second time with different input parameters, or in response to an attestation request (Section 3.5). If a specific application needs to clear sensitive state before another run of the same SSCB, then it should do so before exiting.

SSCB Unregistration. Unregistration can only be initiated by the application that originally registered a particular SSCB. During unregistration, TrustVisor zeros all execution state associated with that SSCB. This includes its stack, heap, and TrustVisor-internal state such as the relevant page table entries and μTPM state. Section 4 presents the full details of unregistration in our implementation. The relevant pages are once again marked accessible to guest mode (the untrusted application and OS).

3.3 Data Secrecy and Integrity Mechanisms

Here we describe how TrustVisor protects the secrecy and integrity of data, where data can be internal TrustVisor state as well as the state of SSCBs. We distinguish two intervals during which data protection is required: residence in volatile storage (RAM) while a particular SSCB is executing, and residence on non-volatile storage while untrusted code is executing.

3.3.1 Data in Volatile Storage

Here, volatile storage refers to data in memory that is protected by TrustVisor and the system's MMU and DEV. The same mechanisms that provide code integrity for TrustVisor and executing SSCBs apply here to protect the secrecy and integrity of data.

TrustVisor-internal state is protected by keeping it in the region of memory that is accessible only to code in host mode (Figure 3). TrustVisor configures the NPTs such that guest physical memory simply excludes the physical pages that contain TrustVisor state. There is no way for code in the guest to address the pages containing the sensitive state. Likewise, TrustVisor programs the system's device exclusion vector (DEV) to prevent access to these pages by DMA-capable devices.

While a SSCB is registered, TrustVisor ensures that the machine physical pages that contain SSCB state are unmapped from the legacy OS's guest physical memory space. Data and code pages are treated differently. Any attempt by the legacy OS to read, write, or execute the data pages will trap to TrustVisor, which will prevent the access. Executing the code pages that belong to any SSCB is legal, and is handled as described in Section 3.2.2. Likewise, a SSCB that wishes to output any of its state to the untrusted world can do so simply by returning it as an output parameter.

3.3.2 Data in Non-Volatile Storage

Part of the TrustVisor model is that the legacy OS manages all devices on the system. Thus, any data that is to be put into non-volatile storage will be under the control of the untrusted legacy OS. This requirement will arise during the lifetime of a system with TrustVisor whenever that system reboots, and whenever a SSCB needs to save state for extended periods of time, across many runs.

To protect the secrecy and integrity of data under these conditions, cryptography is required. In brief, sensitive data that must be protected on non-volatile storage must be encrypted and integrity protected. Numerous algorithms exist for authenticated encryption – the real challenge is to protect the relevant cryptographic keys.

We employ two levels of protection for cryptographic keys with TrustVisor. TrustVisor protects its own secrets using cryptographic keys protected by the system's Trusted Platform Module, and exposes a similar interface to SSCBs that they can use to protect the encryption keys for their secrecy- or integrity-protected data. We explore these mechanisms in greater detail in Section 3.4.

3.4 μTPM Functions

We observe that many applications are still valuable with a greatly reduced set of TPM functions. The software μTPM interface exposed by TrustVisor exports the following TPM-like functions:

1. HV_Extend for measuring data,
2. HV_GetRand for getting random bytes,
3. HV_Seal and HV_Unseal for sealing and unsealing data based on measurements, and
4. HV_Quote to attest recorded measurements using digital signatures.

The secrecy and integrity of sealed data is protected by symmetric cryptographic primitives performed in TrustVisor. We describe the particular algorithms used in our implementation in Section 4. These mechanisms are a significant source of TrustVisor's efficiency for trusted computing operations. Previous systems rely on the TPM's low-cost CPU to perform asymmetric sealing and quote operations (e.g., Flicker [15]), whereas TrustVisor executes the HV-family of trusted computing operations on the platform's primary CPU, and uses efficient symmetric primitives for HV_Seal and HV_Unseal. We now provide an introduction to the functionality of TrustVisor's μTPM interface.

HV_Extend. TrustVisor allocates memory from its own address space for eight software μPCRs (Platform Configuration Registers in the μTPM) for each SSCB. Security-sensitive codeblocks can be written to invoke HV_Extend with arguments of their choosing, thereby enabling measurement of input and output parameters, run-time configuration, dynamically loaded executable code, and any other data that may be relevant to a particular SSCB. The semantics of HV_Extend are identical to those of TPM_Extend: Given a measurement $\texttt{m} \leftarrow \texttt{SHA1(data)}$, a particular PCR is extended as follows: $\texttt{PCR}_{new} \leftarrow \texttt{SHA1(PCR}_{old}||\texttt{m)}$.

HV_GetRand. This simple function returns the requested number of bytes using the hardware TPM as a source of randomness.

HV_Seal and HV_Unseal. These functions are also designed to present the same interface as their v1.2 TPM_Seal and TPM_Unseal counterparts [22]. The primary difference is that instead of

operating with respect to the physical PCRs in the system's TPM chip, they operate based on the values in the μPCRs maintained by TrustVisor. HV_Seal gives SSCBs the ability to specify the required state of the μPCRs for the data to be unsealed. HV_Unseal will only succeed if the values in the μPCRs at the time when HV_Unseal is invoked match those specified as arguments to the original HV_Seal call. Note that HV_Seal outputs a ciphertext, and it is the responsibility of application code to store this ciphertext and provide it as one of the inputs to a later call to HV_Unseal. Security-sensitive codeblocks can be written to output this ciphertext to untrusted application or OS code for long-term storage on the system's non-volatile storage resources (e.g., hard drive).

HV_Quote. We have designed HV_Quote to offer fewer options than the corresponding TPM_Quote function. The reason for this is the natural tension between security and privacy in remote attestation, and our desire to achieve the MinTCB property (P2).

HV_Quote uses a single RSA identity keypair μAIK across all SSCBs (μTPM instances), which is generated in a deterministic fashion from the AIK (Attestation Identity Key) currently in use in the system's physical TPM. We generate μAIK by seeding a pseudo-random number generator (PRNG) with a TrustVisor-maintained secret and the active public AIK. μAIK can then be regenerated at a future time without requiring storage of μAIK itself.

In this way, the existing TPM-based mechanisms for protecting the privacy of an attesting system apply equally well to TrustVisor and the SSCBs running thereupon. However, an attacker may still be able to identify which SSCB generated a particular HV_Quote. For mutually distrusting SSCBs, this may be unacceptable. A straightforward extension is to give each μTPM instance its own μAIK keypair, although each attestation will then incur an additional signature operation, since TrustVisor will have to sign each new μAIK keypair as having been generated within TrustVisor.

Once an identity keypair has been generated, it can be cached by TrustVisor and maintained in non-volatile storage using TPM_Seal (sealed to the code image of TrustVisor) on the system's physical TPM. This enables rapid loading during subsequent boot cycles of TrustVisor.

3.5 Roots of Trust for Integrity Measurement

Integrity measurement is the act of keeping track of the cryptographic hash of all software that has been loaded for execution in the TCB. For a particular application's SSCB, this amounts to TrustVisor and the SSCB itself. Measurements of TrustVisor are kept in the system's physical TPM. Measurements of a SSCB are kept in the μTPM instance corresponding to that SSCB.

Integrity measurement is an essential requirement of the Protection and Attestation properties (P1 and P3). Without integrity measurement and a TPM, there is no facility to protect the cryptographic keys that provide data secrecy and integrity when the data is kept on non-volatile storage under the control of untrusted code. Without integrity measurement, there is no trustworthy source of information about what code has been loaded for execution to use in remote attestations – thereby breaking the Attestation property (P3).

Further, our mechanism for maintaining (and attesting and sealing with respect to) integrity measurements is a significant contribution to the MinTCB property (P2). Existing systems to virtualize the TPM suffer from an excessive TCB, as we describe in Section 7. We observe that many applications are still valuable with the reduced set of TPM functions presented in Section 3.4.

Ultimately, trust in a system running TrustVisor stems from TPM-based attestation to the use of dynamic root of trust (DRTM) to invoke TrustVisor via the SKINIT instruction. TrustVisor accesses the TPM chip via its Locality 2 [22] interface and exposes Locality 1 access to the physical TPM chip to the untrusted guest OS (Figure 1) to maintain compatibility with existing suites of TPM applications (e.g., TrouSerS[5]).

[5] `http://trousers.sourceforge.net/`

Each μTPM instance includes the necessary "micro" Platform Configuration Registers (μPCRs) to keep track of a single SSCB's measurements. TrustVisor performs the initial measurement of the SSCB and extends it into the first μPCR in the μTPM instance associated with the registered SSCB. This mechanism is designed to replicate the functionality of the dynamic root of trust provided by the system's physical TPM.

3.6 Attestation and Trust Establishment

Here we describe how attestation enables a remote entity to establish trust in TrustVisor, and subsequently in SSCBs executing on top of TrustVisor. These mechanisms achieve the Attestation property (P3). Building on the two-level integrity measurement mechanism described in Section 3.5, we also design a two-part attestation mechanism. First, we use TPM-based attestation to demonstrate that a dynamic root of trust was employed to launch TrustVisor with hardware-enforced isolation. Second, we use the μTPM functionality built into TrustVisor to demonstrate that a particular SSCB was registered and executed with TrustVisor-enforced isolation.

3.6.1 TPM-Generated Attestation

An external verifier that receives a TPM-generated attestation covering the PCRs into which TrustVisor-relevant binaries and data have been extended conveys the following information to the verifier:

- A dynamic root of trust (e.g., AMD's SKINIT instruction) was used to bootstrap the execution of TrustVisor.
- TrustVisor received control immediately following the establishment of the dynamic root of trust.
- The precise version of TrustVisor that is executing is identifiable by its measurement (cryptographic hash) in one of the PCRs.
- TrustVisor generated an identity key for its μTPM based on the current TPM AIK.

Note that the verifier must learn the identity of the TPM's AIK by some authentic mechanism, such as a PKI, pre-configuration by an administrator, OEM, or system owner. In some cases trust-on-first-use may even be reasonable, but we emphasize that the choice of mechanism is orthogonal to the architecture of TrustVisor.

3.6.2 μTPM- / TrustVisor-Generated Attestation

An attestation from TrustVisor consists of the output of an HV_Quote operation, along with additional untrusted data to facilitate the verifier's making sense out of the values in the μPCRs. The verifier must first decide to trust TrustVisor based on a TPM attestation. If TrustVisor is untrusted, then no trusted environment can be constructed on top of TrustVisor. A verifier learns the following information as it analyzes the contents of the μPCRs:

- μPCR [0] always begins with 20 bytes of zeros extended with the measurement of the registered SSCB. Thus, the verifier can learn precisely which SSCB was registered and invoked during this session on TrustVisor.
- The values in the remaining μPCRs and any other values extended into μPCR [0] are specific to the SSCB that executed, and will not have been influenced by TrustVisor.
- The set of μPCRs selected for inclusion in HV_Quote (and a nonce provided by the remote verifier to ensure freshness) will be signed by TrustVisor's μTPM identity key μAIK, generated by TrustVisor as described in Section 3.4.

Remote Party (RP):	has AIK_{public}, expected hash(TrustVisor) = $\hat{H}$
TV:	TPM_Extend(PCR[18], $h(\mu AIK_{public})$)
RP:	generate $nonce1$, $nonce2$
RP → App:	$nonce1$, $nonce2$
App → SSCB:	$nonce2$
App:	q1 ← TPM_Quote(PCR[17,18], $nonce1$)
SSCB:	q2 ← HV_Quote(μPCR[0], $nonce2$)
App ← SSCB:	q2
App → RP:	q1, q2
RP:	if (¬Verify($AIK_{public}, q1, nonce1$) $\vee\ q.PCR17 \neq h(0\|\|\hat{H})$ $\vee\ q.PCR18 \neq h(0\|\|h(\mu AIK_{public}))$ $\vee$¬Verify($\mu AIK_{public}, q2, nonce2$)) then abort
RP:	μPCR array represents a valid SSCB run.

Figure 4: Attestation protocol by which a remote party verifies that a particular attestation represents a legitimate run of a SSCB.

Note that the verifier can confirm precisely which SSCB executed, and that an SSCB constructed to measure its inputs and outputs enables the verifier to learn that the Execution Integrity of this SSCB is intact.

3.6.3 The Attestation Protocol

Figure 4 contains the attestation protocol used to convince an external verifier that a particular SSCB ran on a particular system with TrustVisor's protections.

4 Implementation

We now describe our implementation of TrustVisor. TrustVisor is a tiny hypervisor that enables a single legacy OS to execute unmodified by leveraging modern x86 hardware virtualization with the latest Nested Page Table (NPT) and Device Exclusion Vector (DEV) support [2, 12]. This hardware support serves to keep the software TCB small (the MinTCB property P2) and to maintain binary compatibility with various legacy x86 OSes (the Compatibility property P5). We have fully implemented TrustVisor on a Dell PowerEdge T105 containing a 2.3 GHz quad-core AMD Opteron CPU with NPT and DEV support and 8 GB of RAM.

We first describe the boot process of TrustVisor using AMD Secure Virtual Machine (SVM) extensions [2] and a hardware TPM. We then explain the protection mechanisms for the TrustVisor runtime and SSCBs. Finally, we describe the details of our μTPM implementation.

4.1 Trusted Boot

We use AMD's SKINIT instruction to bootstrap TrustVisor. The SKINIT instruction creates a "dynamic root of trust" starting from an initially untrusted system state. However, we still need to design a secure startup flow to create an unbroken chain of trust from TrustVisor's launch to the execution of SSCBs.

TrustVisor is invoked in a three-step process by the bootloader (e.g., grub[6]). First, a relocation module we have developed called TLoader relocates TrustVisor, the Linux kernel, and the initial ramdisk (if any). TLoader relocates TrustVisor to the top of physical memory, so that TrustVisor can present the illusion to the untrusted guest OS that the system is equipped with slightly less (see Section 5) physical memory (RAM).

After relocation is complete, TLoader invokes the SKINIT instruction which reinitializes the system's bootstrap processor (BSP) to a trusted state, enables DEV protection for the TrustVisor image, measures it, and transfers control to the TrustVisor entry point. Since the maximum length of the memory region that SKINIT can measure atomically is 64 KB, we split TrustVisor into two parts: initialization and runtime portions. The initialization portion is less than 64 KB and meets the requirements for measurement by SKINIT. After the start address and size of the initialization portion are passed to SKINIT for measurement, the initialization portion takes control of the system and further initializes a protected environment for the runtime portion. Specifically, it sets up DEV protection for the runtime memory region from DMA-based attacks. Then, the initialization portion hashes the memory region of the runtime portion and compares it with a built-in hash value. If the runtime portion passes the verification, then the initialization portion transfers control to the runtime portion. At this point, the initialization portion can be cleared and freed.

The runtime portion of TrustVisor (hereafter: TrustVisor) sets up a VMCB (Virtual Machine Control Block) for the guest OS (currently we support v2.6.21 of the Linux kernel with the Fedora Core 6 patchset) and prepares access to the necessary resources for the corresponding virtual machine (VM). It then boots Linux inside the VM. Thus, the TCB at runtime in the system is only the runtime portion of TrustVisor, which is verified by a chain of trusted software since SKINIT. Even if a vulnerability is found in the TrustVisor initialization code, the window of opportunity for exploiting it is only during the system's boot process.

4.2 Protecting TrustVisor and Sensitive Code

Based on AMD's SVM hardware virtualization, TrustVisor runs as the host while the Linux Kernel and applications run as a guest. Thus, TrustVisor executes at a higher CPU privilege level than the Linux kernel. However, to protect itself and SSCBs, TrustVisor needs to create an isolated environment for them. We first describe the basic memory isolation mechanism employed by TrustVisor. Then, we present how TrustVisor handles the registration process for SSCBs. Finally, we explain how TrustVisor enables a protected environment for SSCB execution.

4.2.1 Memory Isolation for TrustVisor

To achieve memory isolation, TrustVisor virtualizes the system's physical memory using the NPT hardware feature provided by AMD SVM. The NPTs are maintained by TrustVisor in host mode, while the guest OS maintains its own page tables to translate guest virtual addresses to guest physical addresses. Guest physical addresses are further translated to machine physical addresses by the CPU using the corresponding NPT. TrustVisor maintains only one set of NPTs for the guest, which is simply a one-to-one identity mapping from guest physical addresses to machine physical addresses. TrustVisor uses 2 MB page granularity in the NPTs to improve performance by reducing TLB pressure.

To protect itself, TrustVisor sets the NPT permissions such that its physical pages can never be accessed through the NPT from guest mode. To protect its physical pages against DMA access by devices, TrustVisor uses the DEV (Device Exclusion Vector) mechanism, which is a simplified

[6] `http://www.gnu.org/software/grub/`

IOMMU (Input/Output Memory Management Unit) provided by AMD SVM. With DEV support, the system's memory controller is designed to provide DMA read and write protection for physical pages on a per-page basis. TrustVisor sets up DEV protection to cover all of its own physical pages. To prevent an attacker from modifying the DEV settings, TrustVisor also intercepts all PCI configuration space access from the guest. If TrustVisor finds any attempt to access the DEV, it will simply respond as if the device does not exist.

The protection mechanisms described above for TrustVisor are setup statically during initialization. TrustVisor also uses similar mechanisms to protect SSCBs. However, due to the dynamic registration feature TrustVisor exports for SSCBs, those protections have to be setup dynamically at runtime. We describe the details below.

4.2.2 Sensitive Code Registration

As mentioned in Section 3.2.2, application developers need to explicitly register and unregister the SSCBs corresponding to a particular application. Both registration and unregistration consist of a hypercall with parameters to describe the SSCB to be registered. These hypercalls are intercepted directly by TrustVisor, without legacy OS awareness. We have developed tools to automate the process of putting sensitive code and regular code on separate pages. Thus, the registration process is straightforward. During registration, TrustVisor accepts the start address and the size of the SSCB, performs parameter validation to ensure that these values fall within the calling application's address space, and sets up appropriate protections.

TrustVisor performs three steps to setup the protections for a SSCB during registration. First, using the start address and size, TrustVisor collects all the physical pages that correspond to this address range by walking the current guest page tables. Together with these physical pages, TrustVisor also needs to save the base address of the current guest page table structure (maintained in the guest's CR3 register) as part of an indicator that can be used to identify this sensitive code in the future. Second, TrustVisor will create a μTPM instance specific to this sensitive code, measure the sensitive code by hashing the contents of its pages, and extending the measurement into the μTPM instance corresponding to that SSCB. Third, TrustVisor marks all the corresponding machine physical pages in the NPT as not accessible from the guest, and sets up DEV protection for those pages. For performance reasons, whenever changing permissions in the NPT, TrustVisor changes between 2 MB and 4 KB NPT granularities as necessary.

In our current implementation, TrustVisor does not support multiple SSCBs that share share any physical pages. This situation may result from a common optimization performed by Linux for the read-only code pages of multiple instances of the same executable. It is a straightforward endeavor to make copies of such pages, but we have not yet implemented this feature. Currently, if an overlapping registration is attempted, the registration will return a failure code to the calling guest application.

Unregistration is initiated via a hypercall from the untrusted portion of an application. During unregistration, TrustVisor first verifies that the physical page numbers inside the SSCB and the current CR3 in the guest have already been registered. If so, the execution resources for the SSCB are cleared, including the data region of the SSCB and the corresponding μTPM inside TrustVisor. Finally, protections are removed from the NPT and DEV for all the physical pages corresponding to this SSCB. All registration information for this SSCB is removed from TrustVisor's state.

4.2.3 Sensitive Environment Switching

As mentioned in Section 3.2.2, after registration, an application can just use ordinary function calls to invoke a function in the SSCB, and to get the results after the sensitive function returns. There is

no difference between calling a sensitive function in a registered SSCB and calling another function in the untrusted portion of the application.

Therefore, TrustVisor needs to transparently get control of the system (1) when any sensitive function is called by the application, and (2) when the sensitive function returns to the application that called it. TrustVisor will switch between legacy mode and secure mode at those points. TrustVisor's memory virtualization implementation based on NPT makes this interposition straightforward. In legacy mode, the pages that belong to the registered SSCB are marked as inaccessible from the guest. This guarantees that when the application running in legacy mode attempts to execute the sensitive code or touch the data inside the SSCB, the CPU will generate a nested page fault and trap into TrustVisor.

The opposite is also true. In secure mode, all the pages that are not part of the current SSCB are inaccessible from the guest, and will incur a nested page fault whenever they are executed or touched. Therefore, TrustVisor can always get control when an application calls between legacy mode and secure mode. The input data available to a SSCB is that which is marshaled in by TrustVisor, and has passed TrustVisor's parameter checking. Allowing a SSCB to arbitrarily read other memory may compromise the secrecy of data belonging to other applications, e.g., security-sensitive legacy applications. It may also serve as a vector for a compromised application to more easily send corrupt input data to a SSCB. Thus, this design helps to preserve execution integrity.

We now describe the operations that TrustVisor performs to switch from legacy mode to secure mode to guarantee the execution integrity of the SSCB (in response to a trap as described above). First, TrustVisor ensures that the code in secure mode is running in ring 3 with interrupts disabled. Second, only the physical pages belonging to this SSCB are marked accessible from the guest. However, since we try to run SSCBs with the current application's execution environment, we also need to let the guest have access to some critical system resources, such as some page tables, GDT, and LDT. Thus, in the third step, TrustVisor marks pages containing the GDT and LDT as read-only from the guest. Furthermore, TrustVisor marks a small set of page tables as accessible, since those page tables are used to translate addresses for the SSCB, GDT, and LDT. However, TrustVisor also verifies that the system bit is set in the page table entries corresponding to those critical system resources, so the SSCB running in ring 3 cannot write any information in the pages containing the critical system resources.

The steps described above show that TrustVisor sets up a highly restricted, secure mode for executing SSCBs. TrustVisor also marshals input and output parameters, saves the current stack pointer in the guest, and initializes the stack pointer in the guest within the SSCB. The pages allocated for use as the secure mode stack are passed into TrustVisor during registration. Finally, TrustVisor transfers control to whichever sensitive function is called by the application. The return point in the application is saved in TrustVisor so that it cannot be modified by a malicious SSCB.

After the sensitive function returns to the untrusted application, TrustVisor needs to perform the opposite steps to switch from secure mode back to legacy mode. First, TrustVisor marshals the output parameters back into the untrusted portion of the application and recovers the stack pointer in the guest. Then, TrustVisor updates the NPTs to mark all the pages that are not part of the SSCB as accessible from the guest, and sets the SSCB pages as inaccessible from the guest. Finally, TrustVisor transfers control back to the legacy application, so that the application can process the results returned by the SSCB and continue to run. Note that a SSCB that makes an explicit call (as opposed to a return) to untrusted code will be terminated. This is interpreted as an error in failing to register the other code as security-sensitive.

Dbg	Init.	Runtime					.h
	C + ASM	Core	μTPM	RSA	Lib	Total	
451	865 + 128	1631	467	2666	542	5306	2552

Table 1: Lines of code. For code parts that mix C and assembly, we report the counts separately.

4.3 μTPM Implementation

Our μTPM implementation is part of TrustVisor. TrustVisor maintains three long-term secrets in sealed storage protected by the platform's physical TPM. These are the encryption and MAC keys used to protect the secrecy and integrity of data sealed (using HV_Seal) by a μTPM instance, and the PRNG seed used to derive the μTPM's μAIK keypair. For the μTPM seal and unseal operations, we use AES in CBC mode with 128-bit keys and HMAC-SHA1 with 160-bit keys for secrecy and integrity protection, respectively. We use a 160-bit TPM-generated random PRNG seed. The μAIK keypair is a 1024-bit RSA signing keypair, and is used when a SSCB invokes HV_Quote. A unique array of 8 μPCRs is allocated for each SSCB, and used in the HV_* family of operations from Section 3.4. The data structures used to enforce the required μPCR values during HV_Unseal are identical to those employed by the physical TPM [22].

5 Evaluation

In this section, we first present the TCB size of our TrustVisor implementation (Section 5.1). We then present the performance impact on a legacy system running on TrustVisor (Section 5.2), since these results explain the basic hardware virtualization overhead intrinsic to the design of TrustVisor. Finally, we evaluate the performance for SSCBs on TrustVisor and compare it with systems achieving similar security properties (Section 5.3).

Our experimental platform is a Dell PowerEdge T105 with a Quad-Core AMD Opteron running at 2.3 GHz and 4 GB of RAM. However, our current implementation of TrustVisor allocates 2 GB of RAM to the Linux kernel (v2.6.21), and supports only a uniprocessor guest. The additional cores and RAM are unused. Our server runs the 32-bit version of the Fedora Core 6 Linux distribution. Since TrustVisor is binary-compatible with the legacy OS, experiments with and without TrustVisor are run on the identical uniprocessor kernel image. For network benchmarks, we connect another machine via a 1 Gbps Ethernet crossover link and run the T105 as a server.

5.1 Design Requirements Compliance

We evaluate how our implementation achieves properties MinTCB (P2) and Compatibility (P5), including the porting effort for privilege-separated programs.

Reduced Trusted Computing Base. We use the sloccount[7] program to count the number of lines of source code in TrustVisor. The results are presented in Table 1. We divide TrustVisor's code into four parts. The debug code provides printf and serial console functions which are not required on a production system. The initialization code is the initialization portion in Section 4.1, which is measured by SKINIT and initializes the protected environment for the runtime. The header files contain only declarations and preprocessor macros. Finally, the runtime code is responsible for providing the guarantees for SSCBs as described in Sections 2 and 3. We further divide the runtime code into core functionality (including TrustVisor's basic NPT-based protection framework, SSCB

[7] `http://www.dwheeler.com/sloccount/`

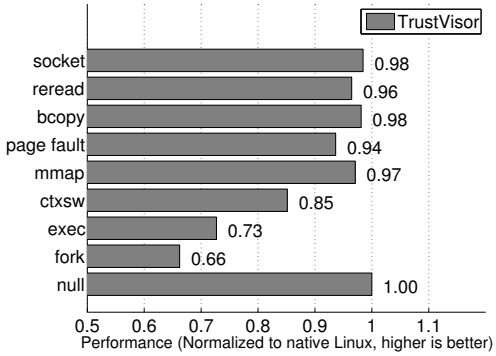

Figure 5: Lmbench micro-benchmark performance comparison between TrustVisor and native Linux.

management, and parameter marshaling), μTPM, RSA libraries, and other libraries (such as SHA-1 and string functions).

As shown in Table 1, the total size of TrustVisor implementation is 6750 lines of C and assembler code (the sum of the debug, initialization, and runtime code). The runtime TCB is about 5306 lines, which includes 3208 lines of RSA and other libraries. This is the extent of the software TCB for TrustVisor, which places it within the reach of formal verification and manual audit techniques.

Compatibility and Porting Effort. As a security hypervisor, TrustVisor virtualizes the physical memory using NPTs, configures the DEV to provide DMA protection for security-sensitive pages, and intercepts a small set of infrequently used hardware I/O operations to prevent malicious code from modifying the NPT and DEV protection mechanisms. TrustVisor can support any legacy x86 OS image without any modifications. The legacy OS and its applications need not be aware of TrustVisor unless they would like to take advantage of registering and executing SSCBs with TrustVisor's protections.

Porting security-sensitive application modules to TrustVisor is straightforward. These modules should be statically linked against a small library and linker script to provide an aligned memory layout. Ordinary code will execute, provided that it does not make system calls to the legacy OS. For workloads such as scientific computation or cryptography, this requirement is readily met.

5.2 Impact on Legacy Software

As explained in Section 4, TrustVisor only receives control as a result of a hypercall or trap. Thus, when well-behaved legacy code runs, the performance overhead is exclusively the result of the hardware virtualization mechanisms, particularly the nested paging. To evaluate this overhead, we run all the experiments in Linux on top of TrustVisor without registering any SSCBs.

OS Microbenchmarks. We use the lmbench suite to measure the overhead of different OS operations when running on top of TrustVisor. Figure 5 shows the results of 9 important operations in our experiments: null (null system call), fork, exec, ctxsw (context switch among 16 processes, each 64 KB in size), mmap, page fault, bcopy (block memory copy), mmap read (read from a file mapped into a process), and socket (local communication by socket). Most of these benchmarks show less than 6% overhead as a result of TrustVisor. However, fork, exec and ctxsw do incur higher performance penalties of 34%, 27% and 15%. This is not surprising as those operations stress the system's

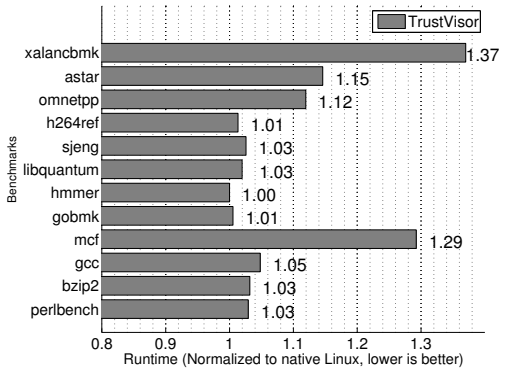

Figure 6: SPECint 2006 runtime comparison between TrustVisor and native Linux.

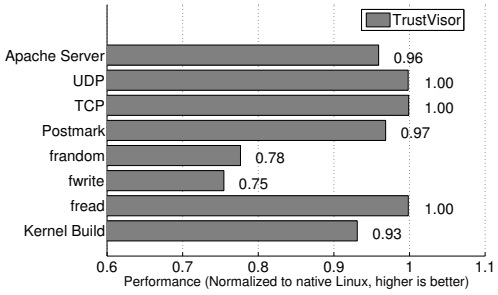

Figure 7: I/O Benchmark performance comparison between TrustVisor and native Linux.

MMU and TLB functionality – components which are highly sensitive to the hardware performance of NPT. In practice, most workloads execute these operations infrequently.

Application Benchmarks. We execute both compute-bound and I/O-bound applications with TrustVisor. For compute-bound applications, we choose the SPECint 2006 suite. For I/O-bound applications, we select a wide range of benchmarks, including the Linux kernel build, Bonnie, Postmark, netperf and Apache webserver.

For the kernel build, we compile the Linux kernel 2.6.21 by executing "make". For Bonnie, we choose a 1 GB file and perform sequential read (fread), sequential write (fwrite), and random access (frandom). For Postmark, we choose 20,000 files, 100,000 transactions, 100 subdirectories, with all other parameters set to their default values. For netperf, we use the TrustVisor system as the netperf server, and run both TCP_STREAM and UDP_STREAM benchmarks to evaluate basic network performance. We run the Apache webserver on the TrustVisor system, and use the Apache Benchmark (ab) included in the Apache distribution to perform 50,000 transactions with 5 concurrent connections.

Our results are presented in Figure 6 and Figure 7. Most of the SPEC benchmarks show less than 3% performance. However, there are two benchmarks with over 10%, and two more with over 30% overhead. We attribute the overhead to the memory footprint characterization of these benchmarks. For I/O application benchmarks, sequential access to very large files incurs the highest overhead –

	Registration			Unregistration			Execution				
	4K	16K	64K	4K	16K	64K	0K	4K	8K	16K	32K
Native	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	0.04	0.04	0.04	0.04	0.04
TrustVisor	56	137	463	1.15	1.23	1.67	27	96	159	288	544

Table 2: Execution times of SSCB microbenchmarks. All times are in microseconds.

over 20%. All of the other benchmarks show less than 7% overhead or even no overhead at all.

5.3 Performance of SSCBs

We evaluate the overhead when TrustVisor receives control in 5 cases: (a) when an application registers a SSCB, (b) when any function inside the SSCB is called, (c) when a function inside the SSCB finishes execution and returns to the application, (d) when an application unregisters a SSCB, and (e) when a SSCB calls any μTPM function.

We use microbenchmarks benchmarks to measure the overhead of the TrustVisor framework in cases (a) – (d), and the overhead of μTPM operations provided by TrustVisor in case (e). We also evaluate the performance of real applications to illustrate the overall performance in a practical environment.

TrustVisor Framework Overhead. The overhead of TrustVisor framework has 4 causes. (1) Each time TrustVisor is invoked, the CPU must switch from guest mode to host mode, which includes saving the current guest environment into the VMCB, and loading the host environment from the VMCB. After TrustVisor finishes its task, the CPU will switch back to the guest by performing the reverse environment saving and loading. Thus, there will be a noticeable performance impact from both cache and TLB activity. (2) When TrustVisor sets NPT protections for SSCB or switches between guest legacy mode and guest secure mode, it will walk the page tables in the guest and change permissions in the NPT followed by some TLB operations. The bigger the SSCB, the more overhead is incurred. (3) Integrity measurement during registration uses SHA-1 to hash the complete SSCB. (4) Parameter marshaling (copying parameters between the untrusted application and SSCB) will incur overhead depending on the size of the marshaled parameters.

Table 2 summarizes the results (from 100 runs with negligible variance) for TrustVisor in these 4 cases. We compare all results to the same operations performed on native Linux. We choose three SSCB sizes for registration and unregistration. For SSCB execution, we choose five different parameter sizes. Our results for the one-time cost of registration show that the performance penalty for a 4 KB SSCB is about 56 μs. With a larger SSCB, the overhead increases by 27 μs per page. This is expected because of integrity measurement overhead: cause (3). Causes (1) and (2) impact unregistration, but the overhead is reasonable – less than 1.5 μs independent of SSCB size. For SSCB execution, the performance is about 27 μs without parameters, compared to 0.04 μ on native. This is a relatively large overhead caused by (1) and (2). Furthermore, because of (4) the overhead of SSCB execution increases linearly with the size of parameters that need to be copied. However, there is no additional performance penalty when sensitive functions run in guest secure mode unless they invoke μTPM operations.

μTPM Overhead. μTPM functions can only be used by hypercalls when the SSCB is running in the secure guest mode. The overhead of μTPM functions comes from two places: (1) the hypercall to switch between the guest and the host, and (2) the performance of the μTPM function itself. For fair comparison with other systems, our results distinguish between these two overheads.

Table 3 summarizes the results (taken over 100 runs with negligible variance) for μTPM operations provided by TrustVisor. We compare all the results to the corresponding operations on native

	Extend	Read	Seal	UnSeal	Quote
Native	24066	18085	358102	1008654	815654
TrustVisor	2.8	0.8	7.9	8.3	4118

Table 3: Execution times of μTPM microbenchmarks. All times are in microseconds.

Linux and the Flicker system [15], which both depend on the hardware TPM.

Application Performance. We have implemented two SSCBs to demonstrate the performance difference between native Linux with hardware TPM operations, and TrustVisor with μTPM operations. Our first SSCB, called hashself, simply hashes its own code pages. The second SSCB is a module to manage a private signing key, as in an SSL-enabled webserver or certificate authority (CA). Its significant operations include unsealing the private key and performing a signature with that key. With and without TrustVisor, the actual signature is performed on the system's primary CPU. The primary TPM and μTPM overhead is the unseal operation.

On TrustVisor, the runtime of hashself is 17 μs, and the CA is 5154 μs. The native runtimes are 4.1 μs and 1013015 μs (which is over 1 second) for hashself and the CA, respectively.

6 Discussion

We now discuss additional issues, including the limitations of our current system, opportunities for formal verification of our system, and additional applications that may benefit from its security properties.

6.1 Limitations

System Management Interrupts Non-maskable interrupts including NMIs and SMIs should be handled when SSCBs or TrustVisor execute. AMD SVM provides facilities for their interception, including the ability to temporarily instantiate another hardware virtual machine that will sandbox the interrupt handler (typically provided by BIOS code).

Availability. A fundamental design decision to minimize the size of TrustVisor is to leave the system's devices under the control of the untrusted legacy OS. This leaves the legacy OS with control over the system's availability. TrustVisor is able to execute applications' SSCBs without OS support, but the OS may prevent these applications from ever running, or it may prevent these applications from communicating over the network. Indeed, the OS may even shut itself down. We consider this to be a worthwhile tradeoff, as alternative designs cause the TrustVisor TCB to grow significantly.

Proof of Correctness. Integrity measurement and attestation are tools for learning what code was loaded for execution, or if that code in turn measures its inputs and outputs, then we can learn what code actually executed and with what parameters. However, the decision as to whether this is the correct execution is application-specific, and outside the scope of attestation mechanisms. TrustVisor is unable to make a security decision about the code that it executes.

Recovery. If a SSCB protected by TrustVisor contains a vulnerability and receives a maliciously crafted input, it may be compromised. TrustVisor isolates this compromised code from the legacy OS and other SSCBs, but it does not offer a recovery mechanism beyond terminating and unregistering the compromised code. Recovery efforts amount to restarting the SSCB with different inputs.

6.2 Formal Verification

Datta et al. show that support for a dynamic root of trust for measurement (DRTM) is a viable means for building a system with code and execution integrity, and data secrecy and integrity protection [9]. A hardware DRTM mechanism is the ultimate root of trust for TrustVisor. We then apply these same principles to another layer, and build a DRTM interface (including the μTPM) on TrustVisor for SSCBs. We plan to build on Datta et al. to prove the security properties of the TrustVisor design [9]. We also plan to verify the TrustVisor implementation using software model checking methods [8].

6.3 Possible Applications of our Externally Verifiable Code Execution

We have used a busy SSL-enabled webserver as a motivating example throughout. Here we consider additional applications (beyond those presented in Section 5) where TrustVisor may provide a security benefit. Many applications requiring protection of a secret or private key will benefit from the reduced TCB of operating on that key exclusively within a SSCB protected by TrustVisor. Examples of such applications include hard drive encryption, certificate authorities, SSH host or authentication keys, and private PGP / email signing and decryption keys. With TrustVisor protecting the sensitive code region(s), even if the untrusted portion of the application is under the control of an attacker, the worst-case malicious act may be invoking the SSCB to sign or decrypt selected messages. The actual value of the private key will remain intact.

Many enterprises today save money by building their systems from off-the-shelf software components over which they have little control. Economic pressures make it infeasible for enterprises to devote significant resources to re-engineering these components, as they will be at a competitive disadvantage. With TrustVisor, enterprises can develop small software modules that run as SSCBs and serve as inline reference monitors or wrappers around the third-party software.

7 Related Work

We discuss related work that attempts to perform secure computation on a host despite the presence of malware.

Singaravelu et al. extract the security-sensitive portions of three applications into AppCores and execute them on the Nizza microkernel architecture [20]. While compelling, the trusted kernel contained on the order of 100,000 lines of code, which is an order of magnitude larger than TrustVisor.

Software-based fault isolation [14, 21, 23] (SFI) and control flow integrity [1] (CFI) are mechanisms that insert inline reference monitors. Unfortunately, all of these systems ultimately depend on the security of the underlying operating system remaining intact, and cannot tolerate a compromise of the system at this low level. In our system, only TrustVisor is trusted to this extent.

We design and implement a μTPM with a dramatically reduced interface that provides only basic TPM functions, but with a primitive to establish a dynamic root of trust. Our μTPM executes as part of TrustVisor on the system's primary CPU, thereby eliminating the performance impact of frequent use of dynamic root of trust in hardware (e.g., Flicker [15]). Other designs, such as the vTPM of Berger et al. include all of Xen, a domain 0 OS, and a software TPM emulator that implements the full suite of TPM functions in their TCB [4]. Though their system exposes more features, its security properties are difficult to verify today. In comparison, the TCB for TrustVisor is orders of magnitude smaller, since we use a minimal hypervisor and do not include any other code in the TCB.

TrustVisor facilitates attestation of externally verifiable application properties in the presence of malware. Other researchers have considered systems for remote attestation [3, 11, 17], but these systems all depend on an unbroken chain of measurement and trust, starting from boot. In practice,

these measurement chains become so long and contain a sheer quantity of code about which we have no techniques to make statements regarding security properties. Researchers have also shown that the Trusted Computing Group's Static Root of Trust for Measurement [13, 16] (SRTM) can be readily compromised.

Researchers have developed systems to reduce the requisite level of trust in operating systems (e.g., CHAOS [6], Overshadow [7], and Yang and Shin [24]). However, the protection granularity in these systems is too coarse to provide strong security properties, because the entire application is in the TCB.

Yao developed cryptographic techniques to perform secure multi-party computations [25]. Sander and Tschudin considered the development of cryptographic techniques that enable mobile code to keep secrets despite tampering attempts by the host [18]. However, these researchers considered only certain kinds of computations (algebraic circuits, i.e., polynomials), whereas TrustVisor supports execution of arbitrary SSCBs.

Seshadri et al. develop SecVisor, a small hypervisor that protects kernel code integrity [19]. TrustVisor is also a small hypervisor, but it merely sandboxes the legacy operating system and provides a trusted environment in which to execute SSCBs in isolation from the legacy OS and its applications.

McCune et al. develop the Flicker architecture to measure and execute small codeblocks with support for attestation. However, the performance penalty imposed by Flicker renders it impractical for busy servers and highly interactive (e.g., media-rich) desktop computing [15].

8 Conclusions

TrustVisor is a small hypervisor that enables isolated execution of security-sensitive codeblocks with a TCB containing only the TrustVisor runtime and the SSCB itself. This system enforces code and execution integrity, and data secrecy and integrity for the SSCBs. We are able to generate highly specific attestations to the SSCB's execution. TrustVisor supports unmodified legacy operating systems and their applications, so that only new applications developed with enhanced security properties require any awareness of TrustVisor. The significant security benefits of TrustVisor outweigh the performance costs, which will mostly vanish with improved hardware virtualization support. Given TrustVisor's features, we anticipate that it can significantly enhance the security of current computing systems and applications.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. CFI: Principles, implementations, and applications. In *CCS*, 2005.

[2] Advanced Micro Devices. AMD64 architecture programmer's manual: Volume 2: System programming. AMD Publication no. 24593 rev. 3.14, Sept. 2007.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A reliable bootstrap architecture. In *IEEE S&P*, 1997.

[4] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vTPM: Virtualizing the trusted platform module. In *Proc. USENIX Security Symposium*, 2006.

[5] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proc. USENIX Security Symposium*, 2004.

[6] H. Chen, F. Zhang, C. Chen, Z. Yang, R. Chen, B. Zang, P. Yew, and W. Mao. Tamper-resistant execution in an untrusted operating system using a virtual machine monitor. Technical Report FDUPPITR-2007-0801, Fudan University, 2007.

[7] Chen et al. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, Mar. 2008.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[9] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A logic of secure systems and its application to trusted computing. In *IEEE S&P*, 2009.

[10] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *SOSP*, 2003.

[12] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual: Volumes 3A and 3B. Nos. 253668-253669, 2008.

[13] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proc. USENIX Security Symposium*, Aug. 2007.

[14] S. McCamant and G. Morrisett. Evaluating sfi for a cisc architecture. In *Proc. USENIX Security Symposium*, 2006.

[15] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[16] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy. TCG inside? - A note on TPM specification compliance. In *Proc. Scalable Trusted Computing (STC)*, 2006.

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

[18] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Proc. Mobile Agents and Security*, 1998.

[19] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*, 2007.

[20] L. Singaravelu, C. Pu, H. Haertig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications. In *EuroSys*, 2006.

[21] C. Small and M. I. Seltzer. Misfit: Constructing safe extensible systems. *IEEE Concurrency*, 6(3):34–41, 1998.

[22] TCG. TPM main specification. v1.2, rev. 103, 2007.

[23] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[24] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE*, 2008.

[25] A. C. Yao. Protocols for secure computations. In *Proceedings of the Symposium on Foundations of Computer Science*, 1982.