TrustVisor: Efficient TCB Reduction and Attestation

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

March 9, 2009

(revised March 10, 2010)

CMU-CyLab-09-003

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

TrustVisor: Efficient TCB Reduction and Attestation*

Jonathan M. McCune Yanlin Li Ning Qu Zongwei Zhou
Anupam Datta Virgil Gligor Adrian Perrig
CyLab, Carnegie Mellon University

Abstract

An important security challenge is to protect the execution of security-sensitive code on legacy systems from malware that may infect the OS, applications, or system devices. Prior work experienced a tradeoff between the level of security achieved and efficiency. In this work, we leverage the features of modern processors from AMD and Intel to overcome the tradeoff to simultaneously achieve a high level of security and high performance.

We present TrustVisor, a special-purpose hypervisor that provides code integrity as well as data integrity and secrecy for selected portions of an application. TrustVisor achieves a high level of security, first because it can protect sensitive code at a very fine granularity, and second because it has a very small code base (only around 6K lines of code) that makes verification feasible. TrustVisor can also attest the existence of isolated execution to an external entity. We have implemented TrustVisor to protect security-sensitive code blocks while imposing less than 7% overhead on the legacy OS and its applications in the common case.

1 Introduction

Current commodity operating systems and applications lack formal assurance that the secrecy and integrity of security-sensitive data are protected. The size and complexity of these systems suggest that we will not achieve the level of assurance necessary to guarantee the absence of security vulnerabilities in these systems in the near future. Even the best-engineered code contains bugs in proportion to its size [24], and available formal methods – while holding great promise for the future – are plagued by scalability challenges. Yet, the convenience and low cost of commodity systems offer unmatched appeal for both users and developers, dictating that security-sensitive workloads *will* be run on these systems for years to come.

This situation highlights the need for techniques to achieve isolated execution of security-sensitive code without breaking compatibility with legacy OSes. Indeed, in recent years many researchers have investigated approaches to execute security-sensitive code while reducing the extent to which the legacy OS and applications are included in the trusted computing base for that code [7, 8, 14, 20, 22, 29, 30, 32, 35]. We briefly lay out the design space explored by existing work and discuss the granularity of the code that is protected.

One possibility is to isolate an entire application from the OS. Several proposals are based on the use of a full-featured commodity VMM that always runs beneath the legacy OS [7, 8, 14, 32, 35]. These works achieve limited security properties because the entire application and VMM code needs to be trusted, bloating the trusted computing base (TCB) by several hundreds of thousands of lines of code. High performance is the main advantage of these approaches.

The Flicker system [22] represents the other extreme of the granularity spectrum, because it protects fine granules of security-sensitive code and adds only a few hundred lines to the TCB. Unfortunately, Flicker incurs significant performance overhead due to its frequent use of hardware support for a *dynamic root of trust for measurement* (DRTM) [2, 16].

In this paper, we aim to achieve the best of both worlds: protect small security-sensitive code blocks within a potentially malicious environment and yet achieve high performance for legacy applications. More specifically, our goal is to provide *data secrecy and integrity*, as well as *execution integrity* for security-sensitive portions of an application, executing the code in isolation from the OS, untrusted application code, and system devices. *Execution integrity* is the property that code P actually executes with inputs P_{inputs} and produces outputs $P_{outputs}$. Finally, we also enable external entities to receive *attestations* that describe the execution of security-sensitive code and optionally its parameters.

To accomplish these goals, we develop a special-purpose hypervisor, called TrustVisor, designed to provide a measured, isolated execution environment for security-sensitive code modules *without* trusting the OS or the application that invokes the code module. This environment is initialized via a DRTM-like process called the *TrustVisor Root of Trust for Measurement*, or TRTM. TRTM interacts with a software-based, "micro-TPM" (μTPM) that is part of TrustVisor and executes at high speed on the platform's primary CPU. We restrict our μTPM to providing only basic randomness, measurement, attestation, and data sealing facilities. Additional trusted computing features can be leveraged by directly interacting with the hardware TPM.

*This research was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 and MURI W 911 NF 0710287 from the Army Research Office, grant CNS-0831440 from the National Science Foundation, and by a gift from AMD. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AMD, ARO, CMU, CyLab, NSF, or the U.S. Government or any of its agencies.

We have fully implemented TrustVisor on an AMD platform, and report on its design, implementation, and evaluation. We also discuss the effort of porting several legacy codebases to take advantage of TrustVisor's protections. TrustVisor works on commodity x86 hardware with virtualization support by leveraging DMA protection [2] and 2D page walking [5]. These mechanisms enforce (IO)MMU-based protection of TrustVisor itself and application-level security-sensitive code and data from the OS, 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). Legacy OSes and applications remain compatible with TrustVisor; changes are only required to applications that wish to leverage the protected environment. TrustVisor imposes less than 7% overhead in the common case, and has a TCB of only 6351 lines of code, over half of which implements cryptographic operations for the μTPM.

TrustVisor enables many exciting applications, but is particularly well suited for implementing oracle-like properties for portions of applications. For example, the security of many cryptographic primitives is based on an assumption that an adversary has access to only a particular interface for the primitive. This assumption can be challenging to enforce in a real-world system as a result of its hierarchical privilege model and large TCB. However, when implemented on TrustVisor, the interface and consequently the attack surface can be carefully constrained.

Contributions. We design and implement a comprehensive system that enables application developers to achieve strong security guarantees for their data and code executing on commodity platforms, and to prove those security properties to an external verifier. The small TCB, efficiency, ease-of-use, and commodity hardware support distinguish our approach from previous efforts.

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 and applications. 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 consider 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 of 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 availability. 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 [12] 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 Background

We describe the hardware dynamic root of trust mechanism and the Flicker system [22], including its prerequisites, security properties, and practical shortcomings.

3.1 Dynamic Root of Trust

Dynamic Root of Trust for Measurement (DRTM) is a mechanism available with AMD's SVM extensions [2] and Intel's TXT extensions [16]. It enables the *measured launch* of a protected code module at any time during a system's operation. *Measurement* denotes computing a cryptographic hash over code before it is executed. This process amounts to reinitializing all CPUs (but not other devices) to a well-known state, computing a cryptographic hash over the relevant code region *after* memory isolation and DMA protection mechanisms are active, and *before* the launched code begins to execute. The measurement is *extended* into a Platform Configuration Register (PCR) in the system's TPM chip [33] in such a way that this measurement can be distinguished as occurring during the establishment of a DRTM (as opposed to a reboot).

This measurement process enables TPM-based remote *attestation* and data *sealing*. An attestation is a TPM-signed list of PCR values that enables an external verifier to make a security decision about the attesting platform. Sealing is a TPM function whereby data is encrypted such that it can only be decrypted if the TPM's PCRs contain pre-defined values (e.g., the values of a known-good version of a hypervisor).

We refer the interested reader to the relevant specifications for additional background on the DRTM process [2, 15, 16].

3.2 The Flicker System

Flicker [22] demonstrates that it is possible to use current trusted computing and hardware virtualization technologies to dramatically reduce the TCB for certain security-sensitive operations. Indeed, with Flicker, current commodity systems are capable of securely executing code without the need to trust the legacy OS. While a valuable proof-of-concept, several characteristics of the Flicker system render it impractical for use in situations with demanding performance (e.g., latency, throughput) requirements.

Each Flicker session takes significant time to execute an application that maintains secrets because slow TPM opera-

tions are on the system's critical path. During Flicker sessions, the user perceives that her system momentarily hangs. This user-experience can be quite disruptive and the performance impact is unacceptable on even a moderately loaded (e.g., tens of users) server. We show that much higher performance is attainable with current hardware by slightly extending the size of the trusted code, but still remaining an order of magnitude smaller than commodity VMMs.

Further, Flicker requires the security-sensitive code of interest to be custom-compiled and linked with very few external dependencies. This complicates the development process and makes debugging more difficult. Though libraries of commonly-used functions may be developed, a preferred solution is one that can protect portions of existing legacy code without modification. TrustVisor employs a registration process compatible with existing code, obtaining its advantage primarily from its ability to understand legacy OSes' memory paging structures.

4 TrustVisor Design

In §4.1, we present a design overview of TrustVisor. We then offer its detailed presentation in two passes. §4.2 describes the memory protection mechanisms that provide isolation between TrustVisor, the legacy OS, zero or more security-sensitive codeblocks, and DMA-capable peripheral devices on the platform running TrustVisor. §4.3 then presents the trusted computing aspects of TrustVisor, including both the roots of trust for TrustVisor itself and the trusted computing support available to security-sensitive code.

4.1 Design Overview

A primary goal of this work is to enable the execution of self-contained security-sensitive codeblocks – called Pieces of Application Logic, or PALs – in total isolation from a legacy OS and DMA-capable devices. We further seek to initialize the isolated execution environment via a process resembling a hardware DRTM, but we want to avoid the severe performance penalty paid by Flicker (e.g., tens or hundreds of milliseconds per session [22, 23]) as a result of its dependence on hardware TPM operations and frequent use of hardware DRTM. We introduce TrustVisor's isolation mechanisms, and then its use of trusted computing.

Memory Protection. TrustVisor has three basic operating modes (Figure 1). *Host mode* refers to execution of TrustVisor code at the system's highest privilege level. TrustVisor in turn supports two guest modes: legacy and secure.

In *legacy guest mode*, a commodity x86 OS 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 device shared between TrustVisor and the untrusted legacy OS.[1]

[1]TPM chips are memory-mapped to multiple addresses, each corresponding to a different privilege level called a *locality* [33]. TrustVisor's memory protections prevent the legacy guest from accessing privileged localities.

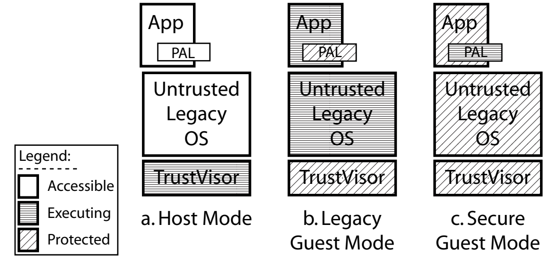


Figure 1: TrustVisor memory protections from the perspective of executing code. (a) In host mode, TrustVisor is executing in response to a trap or hypercall, and may manipulate the state of a PAL, or the untrusted legacy OS or applications. (b) In legacy guest mode, TrustVisor isolates PAL state and its own memory regions from the untrusted legacy code. (c) In secure guest mode, a PAL is executing, and TrustVisor isolates it from the memory regions of TrustVisor and the untrusted legacy OS and applications.

In *secure guest mode*, a PAL executes in isolation from the legacy OS and its applications. A PAL is identified to TrustVisor via a registration process that employs an application-level hypercall interface, with the PAL execution environment initialized by TrustVisor to a well-known, secure configuration. Note that a PAL can also be a part of the OS itself if making changes to the OS is practical. TrustVisor is orders of magnitude smaller than a full OS, thereby bolstering its ability to provide assured isolation between a PAL and all untrusted code and devices on the system. All PAL input parameters are marshaled by TrustVisor into protected memory before the PAL begins executing.

TrustVisor leverages available hardware virtualization support to provide memory isolation and DMA protection for each PAL (Figure 2). In summary, TrustVisor provides isolation by virtualizing a machine's physical memory, enforcing memory isolation between different PALs and untrusted code, and protecting against malicious DMA reads and writes.

Trusted Computing. A DRTM-like mechanism provides the valuable security properties of a known-good initial state, memory protection from DMA accesses, and integrity measurement of the launched code before it executes. We devise a suitable mechanism for PALs called the TrustVisor Root of Trust for Measurement, or TRTM. The TRTM is realized via the inclusion of a TrustVisor-managed, software *micro* TPM (μTPM) instance associated with each PAL (§4.3). The μTPM executes on the platform's primary CPU for high performance while avoiding the TCB growth required of a full software TPM implementation (e.g., vTPM [4]). The TRTM is instantiated as part of the PAL registration process, and is designed to serve as a "second-layer" dynamic root of trust, where the PAL code is isolated and measured before it is executed. The combination of the isolated environment, TRTM, and μTPM offer PALs facilities for fine-grained remote attestation and long-term protection of sensitive state with a small TCB.

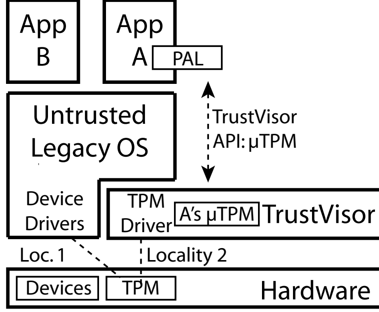


Figure 2: System architecture with TrustVisor. Applications can register PALs for execution in isolation from the untrusted legacy OS and applications. The OS remains responsible for controlling the platform's devices. The only interface exposed to a PAL by TrustVisor is that of a μTPM. The system's physical TPM is shared by TrustVisor and the OS using the TPM's locality mechanism.

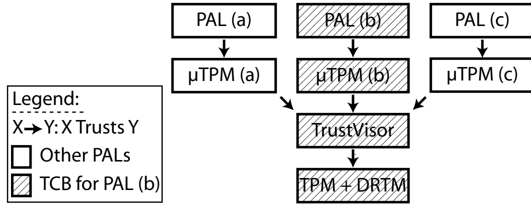


Figure 3: Trust relationships in the TrustVisor architecture.

To distinguish between legacy code and PALs, we devise a registration mechanism by which untrusted applications can register selected code and data as security-sensitive. Registration triggers the sequence of TRTM operations, including allocation of a μTPM instance and protection of the PAL's memory pages. Once registered, a PAL can be invoked multiple times without requiring a new TRTM operation. The μTPM instance provides PALs with a facility for long-term secret protection, and enables remote attestation that a particular PAL has executed.

TrustVisor enables remote attestation and long-term protected storage for PALs via the TRTM and μTPM associated with each PAL. TrustVisor is itself instantiated using the hardware dynamic root of trust mechanism, thereby reducing the TCB for TrustVisor and PALs executing thereupon, and rooting trust in TrustVisor in the platform's physical TPM. Figure 3 shows the relationship of trusted components when multiple PALs are registered. The shaded areas indicate the trusted components in the TCB for a particular PAL.

4.2 Memory Protection Mechanisms

TrustVisor enforces code and execution integrity, and data secrecy and integrity. We first describe how TrustVisor protects itself, and then show how TrustVisor provides these properties for PALs.

4.2.1 Hardware Memory Protections

TrustVisor must protect its own memory regions while also isolating PALs from each other, from the legacy OS and its applications, and from DMA-capable devices. We further wish to support unmodified legacy OSes and legacy applications. Though it is our intention for PALs to be more trustworthy than the legacy OS, PALs are still programs written by humans, and may be susceptible to compromise; e.g., specially crafted input may stimulate a latent bug in the PAL. Thus, it is prudent to prevent PALs from arbitrarily accessing other memory, as they may compromise the secrecy of data belonging to other applications, e.g., security-sensitive legacy applications or other PALs.

TrustVisor uses secure x86 hardware virtualization support to securely bootstrap itself, as well as to enforce isolation between TrustVisor itself, the legacy OS, and PALs. Efficient memory isolation and low hypervisor complexity are more readily achieved given the increasingly wide availability of 2D hardware page walkers [5] that natively support separate paging structures for virtual-to-physical address translation in guest mode, and physical-to-machine address translation in host mode (Figure 1). The memory regions accessible by DMA-capable devices can also be restricted by the hypervisor using modern platforms' IOMMUs (Input/Output Memory Management Unit).

TrustVisor configures its page tables such that guest physical memory simply excludes the machine pages that contain state that must remain inaccessible. Likewise, TrustVisor programs the system's IOMMU to prevent access to these pages by DMA-capable devices. This design enforces code integrity and data secrecy and integrity for both TrustVisor itself and PALs, since a compromised legacy OS can only manipulate the virtual CPU that is under the control of TrustVisor. Even if the malicious OS reprograms DMA-capable devices, the IOMMU will prevent access to TrustVisor or PAL memory regions.

4.2.2 Protection Life-Cycle for PALs

We now describe the life-cycle of a PAL, which begins when code is first identified as comprising a PAL via a registration process. We detail how TrustVisor is configured to provide code and execution integrity, and data secrecy and integrity, to PALs. We define code integrity to be the property that code P has not been modified from its intended version, and *execution integrity* to be the property that code P actually executes with inputs P_{inputs} and produces outputs $P_{outputs}$. We discuss these properties as a PAL progresses through registration, invocation, termination, and unregistration.

PAL Registration. To avoid modifying the legacy OS to support PALs, TrustVisor implements an application-level hypercall interface for registering PALs (though PALs can also be components of the OS if desired). The registration interface allows 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 design makes it the responsibility of application developers to identify the security-sensitive regions of their programs and group sets of functions into one or more PALs and untrusted portions. Essentially developers are required to perform privilege-separation.[2]

TrustVisor verifies that the specified addresses belong to the calling application's address space, and (un)marshals parameters between legacy mode and secure mode when PAL functions are invoked. The registration hypercall returns an error if the provided addresses are illegal.

While a PAL is registered, TrustVisor ensures that the machine physical pages that contain any relevant PAL state (both code and data) are unmapped from the legacy OS's guest physical memory space. Any illegal access by the untrusted application or legacy OS to read, write, or execute the PAL's registered pages will trap to TrustVisor. TrustVisor handles illegal accesses by injecting a fault (e.g., General Protection Fault, Segmentation Fault, or Bus Error) into the legacy OS, which will handle it in accordance with that OS's design (typically by terminating the offending process).

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

1. Identify which registered PAL contains the current called sensitive function.
2. Switch from legacy guest mode to secure guest mode, with secure guest mode configured so that only the pages containing this PAL are accessible.
3. Prepare the secure-mode execution environment for the called sensitive function. This includes marshaling input parameters into isolated pages available to the PAL and setting up the PAL's stack pointer.

Passing pointers in and out of a PAL requires knowing the size of the pointed-to area. (This information is provided as part of the registration call, when entry-points are enumerated.) Thus, nested pointers (e.g., a pointer to a struct that contains another pointer to a buffer) must be marshaled by PAL developers during invocation. Likewise, a PAL that wishes to output any of its state to the untrusted world can do so simply by passing it as an output parameter. Note that, despite TrustVisor's protections, PAL developers must take care to perform appropriate input parameter validation, as untrusted code may invoke a PAL with arbitrary inputs.

The application that registers a PAL is held responsible for faults or exceptions caused by the PAL. TrustVisor zeros the PAL's state and injects the fault into the legacy OS. Thus, data secrecy is maintained and applications can attempt recovery.

[2]While automatic privilege separation may be possible in some instances [6], such mechanisms are beyond the scope of this paper.

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

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

The PAL's execution state is left intact, so that the corresponding untrusted application can invoke it a second time, e.g., with different input parameters. Thus, PALs should clear their sensitive state to ensure semantic security if warranted by application requirements.

PAL Unregistration. Unregistration is normally initiated by the application that originally registered a particular PAL. However, it can also be initiated by the legacy OS if a PAL exits due to an error (e.g., a null-pointer exception). Either way, other than the PAL's output parameters, TrustVisor zeros all execution state associated with that PAL. Once all PAL state is cleared, the relevant pages are once again marked accessible to the untrusted OS.

4.3 Trusted Computing Mechanisms

Trusted computing mechanisms are used to provide two basic capabilities for TrustVisor and the PALs it supports. The first is a *sealed storage* mechanism, by which a particular PAL can encrypt data along with a policy such that the resulting ciphertext can only be decrypted by the PAL specified in the policy. The second is a *remote attestation* mechanism by which a remote party can be convinced that a particular PAL indeed ran on a particular platform (optionally with particular inputs and producing particular outputs) protected by TrustVisor. Both of these mechanisms are enabled by an *integrity measurement* process that maintains a set of measurements (cryptographic hashes) of all code in the TCB for a PAL of interest.

The security properties provided by these mechanisms ultimately stem from hardware roots of trust – the TPM chip and the platform's chipset and CPU support for dynamic root of trust. However, as it is our goal to enable arbitrarily many PALs to be registered with TrustVisor concurrently, we must provide a means to delegate the hardware root of trust to PALs as needed. This is accomplished through a software μTPM instance associated with each registered PAL. The μTPM maintains integrity measurements and enables sealed storage and attestation for a specific PAL. We detail the interactions between hardware trusted computing primitives provided by the TPM and chipset, TrustVisor, μTPM instances, and PALs.

4.3.1 Roots of Trust and Integrity Measurement

Code integrity measurement is a prerequisite for remote attestation and long-term data protection. It comprises keeping track of the cryptographic hash of all software that has been

loaded for execution in the TCB for some operation. For a particular PAL, this amounts to TrustVisor and the PAL itself. Integrity measurement provides a trustworthy source of information about what code has been loaded for execution to use in remote attestations. Further, it serves as a means for controlling access to the cryptographic keys used by sealed storage to provide long-term data secrecy and integrity on a per-PAL basis.

Two-Level Integrity Measurement. TrustVisor employs a two-level approach for integrity measurement. The physical TPM stores measurements of TrustVisor when it is invoked via hardware DRTM, and TrustVisor in turn measures each PAL when it is registered. This design is intended to avoid Flicker's performance issues and monopolization of the platform's DRTM capabilities. PAL integrity measurements are maintained in a software μTPM that exposes trusted computing and dynamic root of trust capabilities to PALs.

Every registered PAL has its own distinct μTPM instance that is created as part of the PAL registration process. However, the μTPM is not created until after TrustVisor's memory protection mechanisms are actively enforcing that no other code or devices on the platform can tamper with this PAL's memory pages. Following μTPM creation, a measurement of the PAL is extended into the μTPM. This measurement includes PAL metadata, including its size and legal entry points. This atomic (from the perspective of the PAL) isolate-then-extend sequence during registration constitutes the establishment of the TRTM. Further, the TCB includes only TrustVisor and the PAL itself (Figure 3). Note that μTPM instances are zeroed and freed whenever a PAL is unregistered, which may be during normal operation or in response to an error.

Measurements extended into a μTPM instance are stored in *micro* Platform Configuration Registers (μPCRs) within the μTPM instance. Thus, TRTM imitates the functionality of the dynamic root of trust provided by the platform's physical TPM, but with the relevant TPM operations performed in software by the μTPM. This enables multiple μTPM instances to exist concurrently, and removes the slow TPM chip from critical-path measurement and data sealing operations.

Measuring Parameters. PAL input parameters, and any outputs produced, can be measured (extended into a μPCR) if the PAL is written to do so, thereby enabling the presence (or absence) of certain inputs and outputs to serve as additional access control to sealed data, and to be attested to remote parties. A PAL written to take full advantage of these capabilities achieves the strongest execution integrity properties. This gives PAL developers maximum flexibility in managing parameters (Figure 4).

4.3.2 μTPM Functions

We describe the μTPM design that TrustVisor exposes to PALs. Many of the more sophisticated TPM functions remain useful to a system running TrustVisor and executing PALs, but they can be leveraged at the whole-system layer of abstraction. An example is the generation of Attestation Identity Keys (AIKs). Multiple AIKs can be generated by the system's physical TPM, and a particular AIK can be used when attesting to a particular PAL running on top of TrustVisor. This does not require any explicit action from the PAL or μTPM. Additionally, migrating TPM-sealed data between physical platforms is accomplished via migration of TrustVisor-level secrets. The higher-level, μTPM-sealed data will unseal perfectly on the relocated TrustVisor.

The small number of commands included in our μTPM design help to keep the TrustVisor TCB small. TrustVisor accesses the TPM chip via its Locality 2 interface during platform startup and shutdown [33], and prevents the legacy OS from accessing this interface. TrustVisor exposes Locality 1 (less privileged) access to the physical TPM chip to the untrusted legacy guest OS (Figure 2), thereby maintaining compatibility with existing TPM-based applications (e.g., the open-source TCG Software Stack [33]).

The software μTPM interface that TrustVisor exposes to PALs includes the following TPM-like functions:

1. HV_Extend for measuring code and data,
2. HV_GetRand for obtaining random bytes,
3. HV_Seal and HV_Unseal for sealing and unsealing data based on measurements, and
4. HV_Quote to attest to measurements in μPCRs.

The secrecy and integrity of μTPM-sealed data is protected by symmetric cryptographic primitives performed in TrustVisor. 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, or monopolize the TPM's scarce non-volatile (NV) RAM, 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 detail the design of TrustVisor's μTPM interface.

HV_Extend. TrustVisor allocates memory from its own address space for the μPCRs in the μTPM for each PAL. PALs 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 PAL. The semantics of HV_Extend are identical to those of the hardware TPM's TPM_Extend: Given a measurement $m \leftarrow SHA-1(data)$, a particular μPCR is extended as follows: $\mu PCR_{new} \leftarrow SHA-1(\mu PCR_{old}||m)$.

HV_GetRand. PALs rely heavily on cryptography because all access to non-volatile storage or network communication involves data travelling through the potentially malicious legacy OS. Thus, it is essential that PALs have a good source of random numbers for generating keys and nonces. HV_GetRand returns the requested number of bytes using a pseudo-random number generator (PRNG) seeded by randomness from the system's hardware TPM. The interface exposed to PAL code is identical to that of the hardware TPM's TPM_GetRand. The PRNG enables HV_GetRand to dramat-

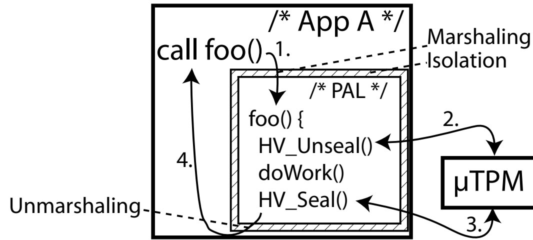


Figure 4: TrustVisor and μTPM-based protections for a PAL containing function `foo`. (1) Input parameters are marshaled by TrustVisor from untrusted code into the PAL. (2) The PAL can invoke the μTPM Unseal command to decrypt previously created secrets. (3) After the PAL serves its purpose, sensitive state can once again be sealed using the μTPM. (4) Outputs from the PAL are unmarshaled back to untrusted code.

ically outperform the corresponding TPM_GetRand by executing on the platform's main CPU, since in the common case no low-speed hardware TPM operations are required.

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 [33]. The primary difference is that instead of authorizing decryption based on values stored in the physical PCRs in the system's TPM chip, these functions operate based on the values in the μPCRs in the μTPM instance maintained by TrustVisor for a particular PAL. HV_Seal gives PALs 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.

HV_Seal outputs a ciphertext that should be included as one of the inputs to a later call to HV_Unseal. It is the responsibility of the untrusted application code to maintain this ciphertext on a non-volatile storage device (e.g., hard disk) for future retrieval, if it is desired that the data survive system reboots or multiple register-unregister cycles of the same PAL. Note that data sealed on one physical TPM cannot be unsealed on a different physical TPM. However, we do allow data sealed by the μTPM associated with one PAL to be unsealed by the μTPM associated with another PAL. This provides the ability to establish a secure channel between multiple PALs. Figure 4 shows a PAL using μTPM-based sealed storage to protect data across multiple registration cycles.

The data sealed by a μTPM is protected using authenticated encryption [18] with keys maintained by TrustVisor itself. TrustVisor protects its own secrets using cryptographic keys sealed by the TPM to the PCR containing the DRTM measurement of TrustVisor. Thus, during TrustVisor boot, these keys are unsealed in a call to the physical TPM. Likewise, any changes in these keys must be re-sealed using a call to the physical TPM prior to system shutdown.

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 keep the TCB small. HV_Quote uses a single RSA identity keypair μAIK across all PALs (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 PALs running thereupon. 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.

4.3.3 Attestation and Trust Establishment

Attestation enables a remote entity to establish trust in TrustVisor, and subsequently in PALs protected by TrustVisor. Building on the two-level integrity measurement mechanisms described in §4.3.1, 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 μTPM-based attestation to demonstrate that TRTM was employed to launch a particular PAL with TrustVisor-enforced isolation. Thus, the ultimate root of trust in a system running TrustVisor stems from TPM-based attestation to the invocation of TrustVisor using hardware DRTM.

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 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 AIK by some authentic mechanism, such as pre-configuration by an administrator 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.

μTPM-Generated Attestation. An attestation from TrustVisor consists of an HV_Quote operation, along with additional measurement metadata[3] 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

[3]The nuances of validating untrusted measurement lists using trustworthy TPM-style measurement aggregates are be beyond the scope of this paper. IBM's IMA discusses one possible mechanism [28].

Remote Party (RP):	has AIK_{public}, expected hash(TrustVisor) = $\hat{H}$
TV:	TPM_Extend(PCR[18], $h(\mu AIK_{public})$)
RP:	generate $nonce, n1 \leftarrow h(1\|\|nonce), n2 \leftarrow h(2\|\|nonce)$
RP → App:	$n1, n2$
App → PAL:	$n2$
App:	q1 ← TPM_Quote(PCR[17,18], $n1$)
PAL:	q2 ← HV_Quote(μPCR[0], $n2$)
App ← PAL:	q2
RP ← App:	q1, q2
RP:	if ($\neg$Verify($AIK_{public}, q1, n1$) $\vee\ q.PCR17 \neq h(0\|\|\hat{H})$ $\vee\ q.PCR18 \neq h(0\|\|h(\mu AIK_{public}))$ $\vee \neg$Verify($\mu AIK_{public}, q2, n2$)) then abort
RP:	μPCR array represents a valid PAL run.

Figure 5: Attestation protocol. Remote Party verifies that a particular attestation represents a legitimate run of a PAL.

TrustVisor is untrusted, then no trusted environment can be constructed using 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 PAL. Thus, the verifier can learn precisely which PAL 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 PAL 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 as described in §4.3.2.

Note that the verifier can confirm precisely which PAL executed, and that a PAL constructed to measure its inputs and outputs enables the verifier to learn that the execution integrity of this PAL is intact. Figure 5 illustrates the attestation protocol used to convince an external verifier that a particular PAL ran on a particular system with TrustVisor's protections.

5 Implementation

We now describe our implementation of TrustVisor. Currently TrustVisor is AMD-specific, but its design applies equally well to widely available Intel systems that include support for both 2D page walks and dynamic root of trust. TrustVisor is a tiny hypervisor that leverages modern x86 hardware virtualization with the latest Nested Page Table (NPT) support and either a Device Exclusion Vector (DEV) or full IOMMU (e.g., AMD's [2]) support: (1) to keep the software TCB small and (2) to maintain binary compatibility with various legacy x86 OSes. We have developed a full, stable implementation of TrustVisor as described in §4, though our implementation currently lacks SMP support. We run our experiments on an off-the-shelf Dell PowerEdge T105 (§6).

We present our implementation in the same order that we presented TrustVisor's design: memory protection mechanisms for TrustVisor and PALs first (§5.1), then trusted computing mechanisms including our μTPM implementation (§5.2). Note that this means TrustVisor's steady-state operation is presented before its boot-up using the trusted computing mechanism dynamic root of trust.

5.1 Protecting TrustVisor and PALs

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 more privileged CPU protection level (*ring* on x86) than the Linux kernel. However, to protect itself and PALs, 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 PALs. Finally, we explain how TrustVisor enables a protected environment for PAL execution.

5.1.1 Memory Isolation for TrustVisor

To achieve memory isolation, TrustVisor virtualizes the guest OS's physical memory using the 2D nested page table (NPT) hardware feature provided by AMD SVM. The NPTs are maintained by TrustVisor in host mode, while the guest OS continues to maintain its own page tables to translate guest virtual addresses to guest physical addresses (i.e., the guest OS need not be aware that it is virtualized). At runtime, 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 an 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 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 accesses 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 statically set up during initialization. TrustVisor also uses similar mechanisms to protect PALs. However, due to the registration feature TrustVisor exports for PALs, those protections have to be set up dynamically at runtime. We describe the details below.

5.1.2 PAL Registration

Application developers must explicitly register and unregister the PAL(s) for their application (recall §4.2.2). Both registra-

tion and unregistration consist of a hypercall with parameters to describe the PAL to be registered. These hypercalls are intercepted directly by TrustVisor without legacy OS awareness using the VMMCALL instruction.

We have developed simple build-process linker scripts to automate the process of placing sensitive code and regular code on separate pages, as well as allocating pages for a PAL's data and parameters. There are six types of sensitive memory pages: PAL entry point code pages, PAL-private code pages, code pages shared between PALs and untrusted applications, PAL data pages, PAL runtime stack pages, and PAL parameter marshaling pages. All of the functions that contain intended entry points to PAL code are collected and linked into an explicit entry-point region that cannot be shared. PAL-private code regions are used to hold all read-only, unshared, PAL-specific code. The shared code region includes routines that may be called by the untrusted applications or other PALs. Sharing is only allowed for *read-only* pages, with shared pages commonly resulting from Linux's copy-on-write functionality during process forking, and from memory mapping and demand paging common code pages for multiple instances of the same binary executable or library. Part of the PAL build process isolates the PAL's initialized and uninitialized data into a dedicated PAL data region, and further allocates additional pages for use as the PAL's stack and as the PAL-accessible location for marshaled input and output parameters. We note that there is no explicit PAL heap. We implement dynamic memory allocation for PALs as a stand-alone library that can be optionally linked into each PAL, that from the perspective of the build process simply includes a large (the size of the heap) static buffer.

During registration, TrustVisor accepts the start address and the size of each page region of the PAL, a list of valid entry points, and information describing the input and output parameters for each entry point. TrustVisor performs three steps to set up the protections for a PAL during registration. First, TrustVisor collects all the physical pages that correspond to each page region by walking the current guest page tables. Note that TrustVisor needs to check that all the permission bits of the guest page table entries during page table walking are consistent with the intended permissions of each page region. This prevents a malicious application from taking advantage of TrustVisor to violate the permissions set by a well-behaved OS, e.g., by attempting to register read-only application pages as writable PAL pages. TrustVisor also needs to save the base address of the current guest page table structure from the guest's CR3 register as part of an indicator that can be used to identify this PAL in the future. Second, TrustVisor sets up permissions for all the corresponding machine physical pages in the NPT structures. All of the corresponding machine pages (except for any shared code pages) are marked as not accessible from the guest. TrustVisor also sets up DEV protection for those pages, to prevent malicious DMA accesses. Third (now that isolation is configured), TrustVisor creates a μTPM instance dedicated to the newly registered PAL, and performs the first measurement of the PAL's non-data pages to instantiate the TRTM.

We leverage Linux's copy-on-write feature to generate multiple copies of non-read-only PAL pages and pages containing PAL entry points. During registration, one byte on each page is written with its current value to force Linux to make a duplicate using copy-on-write. This requires code pages (such as the pages containing the PAL entry points) to be temporarily marked writable during registration. For performance reasons, whenever changing permissions in the NPT, TrustVisor changes between 2 MB and 4 KB NPT granularities as necessary. Essentially, 2 MB pages are used to map contiguous regions 2 MB or larger, since this will consume only a single TLB entry. 4 KB pages are used to map smaller regions, such as a PAL's stack pages.

Note that any attempt by the untrusted legacy OS or its applications to write to any registered page, or to read from any non-shared registered page, will cause a nested page fault (NPF) that will be caught by TrustVisor. If an overlapping registration of non-shared pages is attempted, the registration hypercall will return a failure code to the calling guest application. Valid calls to PAL entry points are allowed, but all other illegal accesses will be prevented by TrustVisor. Our current prototype halts at this point to aid debugging, but a production implementation should inject a fault (e.g., SIGSEGV, bus error) into the legacy guest so that it can reclaim resources from the misbehaving process.

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 PAL and the current CR3 in the guest have already been registered. If so, sensitive data pertaining to the PAL is zeroed, including the data region of the PAL and the corresponding μTPM inside TrustVisor. Finally, protections are removed from the NPT and DEV for all the physical pages corresponding to this PAL. All registration information for this PAL is removed from TrustVisor's state.

5.1.3 Sensitive Environment Switching

TrustVisor needs to transparently get control of the system when any sensitive function is called by the application, and when the sensitive function returns to the calling application. TrustVisor switches between legacy mode and secure mode at those points and marshals the relevant parameters. TrustVisor's memory virtualization implementation based on NPTs makes this interposition straightforward. In legacy mode, the pages that belong to the registered PAL are marked as inaccessible. This guarantees that when the application running in legacy mode attempts to execute the sensitive code or touch the data inside the PAL, the CPU will generate a nested page fault and trap into TrustVisor. TrustVisor uses the page permissions of the page(s) containing PAL entry points to guarantee a trap to TrustVisor whenever a sensitive function is called. Note that the valid entry points for a PAL must not be on a shared page.

Analogously, in secure mode, all the pages that are not part of the current PAL are inaccessible to it, and the PAL will cause a nested page fault (that will be caught by TrustVisor) whenever they are touched (read, written, or executed). Therefore, TrustVisor will always get control during transitions between legacy mode and secure mode. The input data available to a PAL is marshaled by TrustVisor, and is verified by TrustVisor's parameter checking. (Recall that TrustVisor will return a failure code from the registration hypercall if parameter checking fails.)

We now describe the operations that TrustVisor performs to switch from legacy mode to secure mode to guarantee the execution integrity of the PAL (in response to a trap as described above). First, TrustVisor configures the PAL code in secure mode to run in ring 3 with interrupts disabled. Any exceptions generated by the PAL code will be caught by TrustVisor and interpreted as an illegal action performed by the PAL (e.g., a null-pointer dereference or divide-by-zero).

Second, TrustVisor configures the NPTs such that only the physical pages belonging to this PAL are accessible from the guest. However, since we run PALs within a subset of the current application's execution environment, we also need to let the guest have access to some critical system resources, such as the GDT, the LDT, and guest page tables that are used to translate addresses for the GDT, LDT and PAL. Thus, in the third step, TrustVisor configures the NPTs so that pages containing these critical system resources are accessible from the guest with read-only permission.

In the fourth step, TrustVisor verifies that the system bit in each of the guest page table entries corresponding to TrustVisor and the critical system resources is correctly set, so that the PAL running in ring 3 cannot write any information to the pages containing the critical system resources, or read any information from the pages for which the PAL does not have read permission. Note that well-behaved Linux will already have the system bit set for these pages, but with our attacker model a rootkit may have modified them arbitrarily.

Finally, TrustVisor must ensure that the PAL page mappings configured during registration cannot be subsequently changed (e.g., re-ordered) by the legacy OS. Each PAL's guest physical pages are inaccessible to the legacy guest due to the NPT configuration during PAL registration, so the contents of the pages themselves are protected from illegal modification. However, we must still ensure that the virtual-to-physical address translation of registered PAL pages has not been changed since registration. This verification will prevent a malicious OS from compromising PAL code integrity by making clever changes to PAL page tables, e.g., changing the virtual address of an entry point code page to point to the physical address of a PAL-private code page.

The steps described above show that TrustVisor sets up a highly restricted, secure environment for executing PALs. TrustVisor also marshals input and output parameters, copies the memory regions corresponding to these parameters into the PAL's parameter marshaling pages, saves the legacy OS stack pointer, and initializes the stack pointer within the PAL. The pages allocated for use as the secure mode stack and parameter storage are identified during registration and need to have been allocated by the untrusted code prior to PAL registration. However, before copying the memory regions corresponding to input parameters, TrustVisor needs to check that those memory regions are readable by the guest. If the parameter is a reference, TrustVisor also needs to check that the memory region of that parameter is writable by the guest. These operations will prevent a malicious application from passing incorrect parameters to TrustVisor and tampering with the memory permissions set by the OS. Note that despite TrustVisor's protections, semantic security for PALs depends on PALs performing responsible input parameter handling, the details of which are beyond the scope of this paper.

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 PAL. This prevents a malicious application from using a PAL to attempt control-flow attacks.

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. Note that a PAL cannot return heap data in our current implementation – a buffer for such outputs needs to have been allocated by the untrusted application and passed as an input parameter to the PAL. Then, TrustVisor updates the NPTs to mark all the pages that are not part of the PAL as accessible from the guest, and sets the PAL 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 PAL and continue to run. Note that a PAL that makes an explicit call (as opposed to a return) to untrusted code will be terminated as described in §4.2.2.

5.2 Trusted Computing Implementation

We describe how TrustVisor initializes itself using dynamic root of trust to achieve a trusted boot process. We then describe how μTPM instances are implemented in TrustVisor.

5.2.1 Trusted Boot

We use AMD's SKINIT instruction to create a *dynamic root of trust* to bootstrap TrustVisor starting from an initially untrusted system state (§3.1). We note that bootstrapping a hypervisor with DRTM is its intended function. Contemporary projects for booting with DRTM include Kauer's Open Secure Loader [19] and Intel's "tboot".[4]

We now explain how we create an unbroken chain of trust from TrustVisor's launch to the execution of PALs from within Linux applications (currently we have tested v2.6.21 of the Linux kernel with the Fedora Core 6 patchset, and v2.6.27 with the Ubuntu 8.10 patchset).

[4] `http://tboot.sourceforge.net/`

TrustVisor is invoked in a three-step process by the bootloader (e.g., grub). First, an untrusted loader we have developed called TLoader relocates the Linux kernel and initial ramdisk so that they will be able to execute when invoked as a guest by TrustVisor. TLoader also relocates the trusted initialization portion of TrustVisor so that it is aligned on a 64 KB boundary – a requirement for SKINIT.

TLoader's final operation is to launch TrustVisor by invoking 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, the initialization portion relocates the runtime portion to the top of physical memory, so that TrustVisor can present the illusion to the untrusted guest OS (e.g., Linux) that the system is equipped with slightly less (see §6) physical memory (RAM). Note that this design significantly reduces hypervisor complexity since guest physical addresses are also machine physical addresses. Before invoking the runtime portion, the initialization portion sets up AMD's Device Exclusion Vector to provide DMA protection for the runtime memory region. 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 sets up a Virtual Machine Control Block (VMCB) for the legacy guest OS and prepares access to the necessary resources for the corresponding VM. It then boots Linux inside the VM. Thus, the runtime TCB comprises only the runtime portion of TrustVisor, which is verified by a chain of trusted software since SKINIT.

5.2.2 μTPM Implementation

Our μTPM implementation is part of TrustVisor. TrustVisor maintains three long-term secrets using TPM sealed storage. 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-CBC with 128-bit keys and HMAC-SHA-1 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 2048-bit RSA signing keypair, and is used when a PAL invokes HV_Quote. A unique array of 8 μPCRs is allocated for each PAL, and used in the HV_* family of operations from §4.3.2. The data structures used to enforce the required μPCR values during HV_Unseal are identical to those employed by the physical TPM [33].

Dbg	Init.	Runtime					.h
	C + ASM	Core	μTPM	RSA	Lib	Total	
507	773 + 128	1943	619	2339	1580	6481	2790

Table 1: Lines of code in C, assembly, and header files.

6 Evaluation

We present the TCB size of our TrustVisor implementation (§6.1). We then present the performance impact on a legacy system running on TrustVisor (§6.2), since these results explain the basic hardware virtualization overhead intrinsic to the design of TrustVisor. We also evaluate the performance for PALs on TrustVisor and compare it with Flicker (§6.3).

Our experimental platform is a Dell PowerEdge T105 with a Quad-Core AMD Opteron running at 2.3 GHz. Our current implementation of TrustVisor allocates 2 GB of RAM to the Linux kernel and supports only a uniprocessor guest. Additional cores and RAM are unused. Our server runs the 32-bit version of the Fedora Core 6 Linux distribution for the experiments that follow, although no kernel modifications were made other than including "nosmp" on the kernel command line. We have successfully booted other kernels, e.g., v2.6.27 with Ubuntu's patchset. Since TrustVisor is binary-compatible with the legacy OS, experiments with and without TrustVisor are run on the identical nosmp kernel image. For network benchmarks, we connect another machine via a 1 Gbps Ethernet crossover link and run the T105 as a server.

6.1 Trusted Computing Base

We evaluate how our implementation maintains a small TCB and compatibility with unmodified legacy software.

Reduced TCB. We use the sloccount[5] program to count the number of lines of source code in TrustVisor (see 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 described in §5.2.1, which is measured by SKINIT and initializes the protected environment. Finally, the runtime code is responsible for providing the guarantees for PALs as described in §4. We further divide the runtime code into core functionality (including TrustVisor's basic NPT-based protection framework, PAL 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 7889 lines of C and assembly code (the sum of the debug, initialization, and runtime code). The runtime TCB is about 6481 lines, which includes 3919 lines of RSA and other libraries. This is the full extent of the software TCB for TrustVisor, which places it within the reach of formal verification and manual audit techniques.

[5]http://www.dwheeler.com/sloccount/

Compatibility. As a security hypervisor, TrustVisor virtualizes 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 32-bit 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 PALs with TrustVisor's protections.

6.2 Performance of Legacy Software

TrustVisor only receives control as a result of a hypercall or trap (recall §5). 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 PALs.

OS Microbenchmarks. We use the lmbench suite to measure the overhead of different OS operations when running on top of TrustVisor. Figure 6(a) 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 MMU and TLB functionality – components which are highly sensitive to the hardware performance of NPT. We note that these overheads are likely to decrease on future platforms as hardware virtualization support matures.

Application Benchmarks. We execute both compute-bound and I/O-bound applications with TrustVisor. For compute-bound applications, we use the SPECint 2006 suite. For I/O-bound applications, we select a range of benchmarks, including building the Linux kernel, Bonnie,[6] Postmark [17], netperf,[7] and unmodified Apache web server performance.

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 web server 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 Figures 6(b) and 6(c). Most of the SPEC benchmarks show less than 3% performance overhead. However, there are two benchmarks with over 10%, and two more with 29% and 37% overhead. We attribute this high overhead to paging operations performed with the current hardware's NPT support, and expect that performance will improve as NPT hardware matures. For I/O application benchmarks, sequential access to very large files incurs the highest overhead – over 20%. We also expect this overhead to diminish with newer NPT hardware. All of the other benchmarks show less than 7% overhead.

[6] http://www.textuality.com/bonnie/
[7] http://netperf.org/

(a) Registration hypercall overhead with varying PAL sizes.

Registration				Unregistration			
4K	16K	32K	64K	4K	16K	32K	64K
31	112	220	435	1.09	1.17	1.44	1.62

(b) Varying PAL input parameter size.

Parameter marshaling				
0K	4K	8K	16K	32K
25	92	152	279	536

Table 2: PAL setup overhead microbenchmarks (in μs). Avg. of 100 runs with negligible variance.

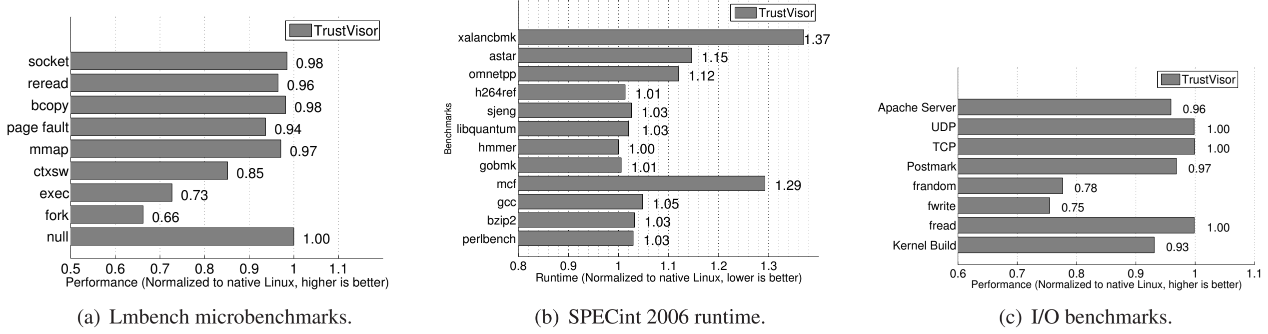
6.3 Performance of PALs

We present micro- and macro-benchmarks to evaluate sources of PAL overhead and application-level impact, respectively.

6.3.1 PAL Microbenchmarks

We evaluate the overhead when TrustVisor receives control in 5 cases (Tables 2 and 3): (a) when an application registers a PAL, (b) when any function inside the PAL is called, (c) when a function inside the PAL finishes execution and returns to the application, (d) when an application unregisters a PAL, and (e) when a PAL calls any μTPM function. We use microbenchmarks 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. TrustVisor's overhead has four 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 PALs or switches between guest legacy mode and guest secure mode, it will walk the page tables in the guest, change permissions in the NPTs, and perform some TLB operations. The bigger the PAL, the more overhead is incurred. (3) Integrity measurement during registration uses SHA-1 to hash the PAL pages containing executable code. (4) Parameter marshaling will incur memory copy overhead between the untrusted application and PAL.



(a) Lmbench microbenchmarks. (b) SPECint 2006 runtime. (c) I/O benchmarks.

Figure 6: Performance impact of TrustVisor compared to native Linux.

	Extend	Seal	UnSeal	Quote
Native Linux	24066	358102	1008654	815654
TrustVisor	533	11.7	12.6	21000

Table 3: TPM vs. μTPM microbenchmarks (in μs). Avg. of 100 runs with negligible variance.

	HMAC		Sign	
	Avg	Stdev	Avg	Stdev
TrustVisor	0.059	0.003	5.071	0.018
Flicker	62.644	0.181	67.461	0.008

Table 4: HMAC and sign PAL overhead performed using TrustVisor vs. using Flicker (in ms). Avg. of 100 runs.

Table 2(a) summarizes the overhead of PAL registration, and Table 2(b) summarizes the overhead of marshaling parameters during PAL execution. We compare these results to the same operations performed on native Linux, where appropriate. We choose four PAL sizes for registration and unregistration. For PAL 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 PAL is about 31 μs. With a larger PAL, the overhead increases by 27 μs per 4 KB page. This is expected because, during registration, the integrity measurement overhead (cause (3)) outweighs other overheads (causes (1) and (2)). The unregistration overhead is reasonable – less than 1.5 μs, and along with increasing PAL size, the elapsed time of unregistration only slightly increases. For PAL execution, the overhead of switching between guest legacy mode and guest secure mode is about 25 μs without parameters, and increases by about 65 μs with each 4 KB page of parameters. The switching overhead increases proportionally to the size of the marshaled parameters because causes (4) and (2) are more significant than cause (1). Note that there is no additional performance penalty when PAL functions run in secure guest mode unless they invoke μTPM operations.

μTPM Overhead. μTPM functions can only be used by hypercalls when the PAL is running in 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, we distinguish between these two overheads in our results.

Table 3 summarizes the results for μTPM operations. We compare all the results to the corresponding operations on native Linux with Flicker [22], which both depend on the hardware TPM.

6.3.2 PAL Macrobenchmarks

HMAC and Sign. Two simple tasks that require a secret key are computing message authentication codes (MACs) and digital signatures. We implemented a routine to compute a HMAC-SHA-1 over a 1000 byte payload using a 512-bit key as both a PAL run using TrustVisor and a PAL run using Flicker. Likewise, we implemented a routine to perform a digital signature using a 1024-bit RSA key over a 20-byte hash value as both a PAL run using TrustVisor and a PAL run using Flicker. Our results are shown in Table 4. TrustVisor outperforms Flicker by several orders of magnitude for the HMAC operations, and by more than one order of magnitude for the sign operations.

OpenSSH. Here we evaluate the overhead induced by TrustVisor on OpenSSH 4.3p2 as modified for use with Flicker [22]. We ported the security-sensitive portions to run in a PAL using μTPM operations. We compare native SSH performance with Flicker- and TrustVisor-induced overheads, executing all versions on our Dell PowerEdge T105.

We modified the Flicker-protected code to use the hardware TPM's Non-Volatile RAM facility for protecting the sensitive state, instead of the hardware TPM's Sealed Storage facility. This considerably improves Flicker's performance, as TPM_Unseal averages nearly 1 second, whereas TPM NV_Read on our machine executes in 15 ms on average with negligible variance. However, NV-RAM does impose scalability issues for Flicker, as there are only a few KB of NV-RAM available in today's TPMs [33]. Thus, the performance results for our Flicker-based runs should be considered a best-case for Flicker.

We define Connect-to-Prompt to be the time elapsed between establishment of the TCP connection and prompting

	Native	Flicker	TV
Connect-to-Prompt	110	1316	1260
Prompt-to-Shell	0	131	11

Table 5: SSH server-side password processing overhead. Note that both the Flicker and TrustVisor Connect-to-prompt figures include the time to generate a hardware TPM_Quote. Avg. of 100 runs.

(a) **Connect-to-Prompt**

	Time (ms)	
Operation	Fli	TV
DRTM	14	0
Key Gen	196	199
Seal	15	0
TPM sharing	64	-

(b) **Prompt-to-Shell**

	Time (ms)	
Operation	Fli	TV
DRTM	14	0
Unseal	15	0
Decrypt	4	6
TPM sharing	64	-

Table 6: SSH server side overhead breakdown for each protected session. The standard deviation on all measurements is negligible, except key generation at 97 and 107 for Flicker and TrustVisor, respectively. Avg. of 100 runs.

the client user for their password, and Prompt-to-Shell to be the time elapsed between password entry and the user being presented with a shell on the remote system. Table 5 compares these overheads between unmodified SSH, Flicker-protected passwords, and TrustVisor-protected passwords. Table 6 presents the relative overheads caused by Flicker and TrustVisor.

SSL-Enabled Web Server. Here we evaluate the overhead induced by TrustVisor on a modified SSL-enabled Apache web server. The goal of this application is to protect the web server's long-term private SSL signing key. We build the web server from source using Apache v2.2.14 and OpenSSL v0.9.8l after porting the security-sensitive portions to run in two PALs. To create our PALs, we replaced some of the RSA operations performed in OpenSSL with equivalent calls to functions provided by the embedded cryptography library PolarSSL[8] v0.12.1. We describe the porting process in more detail in §6.4. The first PAL runs when the Apache server starts and tries to import the long-term private signing key. Instead of reading the private key from a file, the first PAL generates the private key and encrypts it using the μTPM sealed storage operations. The private key is sealed based on the expected measurement of the second PAL, so that only our second PAL will be able to unseal it. The second PAL's responsibility is to use this private key to sign the appropriate SSL handshake messages. Thus, the second PAL runs in response to incoming client connections during SSL session establishment.

We run the Apache web server in two modes: single process mode, and prefork mode. In prefork mode, the server creates multiple child processes (not threads) in advance and assigns incoming client connections to different idle processes. In our implementation, the web server needs to register the second PAL *after* it preforks child processes so that

[8] http://polarssl.org/

Test Scenarios	Concurrent Transactions	Perf (txns/second)		
		Vanilla	TV	Full
Single	1	26.60	24.06	22.96
	5	37.91	37.13	34.57
Prefork	5	53.71	53.53	48.49
	50	57.84	57.31	51.35
	100	58.05	58.03	51.29
	200	58.04	58.07	51.08

Table 7: SSL-based web server performance. Results represent the average number of transactions per second of 10 trials with negligible variance. The Apache Benchmark (ab) issues 10,000 transactions per trial with the specified number of concurrent transactions to the server. In each transaction, a 74-byte index page is transferred from the server to the Apache Benchmark client after an SSL connection is established. RSA keys are 1024 bits long.

each child process can have its own instance of the second PAL, i.e., each child process registers its own PAL. We then evaluate the performance of our modified Apache web server using the Apache Benchmark (ab) included in the Apache distribution to perform HTTPS transactions with varying levels of transaction concurrency.

Table 7 shows our experimental results. We compare our web server (denoted *Full*) with a web server without any PALs registered and running on the same OS on bare metal without TrustVisor (*Vanilla*), and also a web server without any PALs registered but running on the same OS on top of TrustVisor (*TV*).

6.4 Porting Effort

We designed TrustVisor's registration mechanisms to be minimally invasive when porting existing applications to take advantage of the security properties afforded to PALs. However, we have not implemented a privilege-separation or modularity-analysis mechanism. The relative challenge associated with porting an application to include one or more PALs is closely related to the level of privilege separation and modularity existing in the application's architecture.

Separated Programs. Porting security-sensitive application modules to TrustVisor is straightforward if the program is already privilege-separated and modular. Ordinary code will execute as a PAL, 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.

Legacy Programs. Programs that were written without attention to privilege separation or modularity can be challenging to port to include meaningful PALs. We faced the greatest porting challenge with Apache + OpenSSL. Our original intention was to identify the modules in OpenSSL that manipulate the web server's private SSL key and register them as one or more PALs. This proved to be difficult due to OpenSSL's extensive use of function pointers and adaptability to different cryptographic providers, e.g., smart cards. We resorted to replacing the relevant RSA calls with calls to the embedded cryptography library PolarSSL.

7 Discussion

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

7.1 Formal Verification

Datta et al. [10] show that support for DRTM is a viable means for building a system with code and execution integrity, and data secrecy and integrity protection. 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 PALs. We plan to build on the results of Datta et al. to prove the security properties of the TrustVisor design [10]. We also plan to verify the TrustVisor implementation using software model checking methods [9].

7.2 Applications of Externally Verifiable Execution

Many applications requiring protection of a secret or private key will benefit from the reduced TCB of operating on that key exclusively within a PAL 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 will be invoking the PAL to sign or decrypt selected messages. The actual value of the private key will remain secret. Thus, the worst that could happen is that a PAL may become an encryption or signing oracle. Even if this attack is successful, it may be possible to avoid the need to revoke the affected key, which is significant given the challenges that have long plagued certificate revocation in practice.

Many enterprises today limit costs 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 PALs and serve as inline reference monitors [13] or wrappers around third-party software.

7.3 Optimizations / Future Work

We have already identified several optimizations that are not implemented in our prototype but that will further reduce the overhead imposed by TrustVisor or increase its applicability. The first is multi-processor support, and the second is support for recursive virtualizability,[9] so that TrustVisor does not monopolize the use of hardware virtualization features. Finally, there is no need for TrustVisor to run at all in the absence of registered PALs. TrustVisor should have support for unloading itself while it is not needed, and re-launching underneath an OS on-demand. Intel's P-MAPS serves as a proof-of-concept that this is readily achieved [26].

Additional features that may be valuable for PAL development include timeouts and monotonic counters. A timeout is useful to terminate a PAL that has entered an infinite loop. The TPM does include limited monotonic counter support, but per-PAL counters may simplify replay-attack defenses for μTPM-based sealed storage.

[9] VirtualBox (`http://virtualbox.org`) and Blue Pill (`http://bluepillproject.org/`) support this today.

8 Related Work

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

Intel has recently announced a Processor-Measured Application Protection Service called P-MAPS [26]. P-MAPS potentially offers the following features: (1) Isolation of the application's runtime memory from other software on the platform, (2) Encapsulation of the application data memory such that only code in the measured application pages can access the data, and (3) Prevention of circumvention of any function entry-points exposed in the application code. P-MAPS is claimed to be 2500x smaller than a commodity OS, though code size numbers are not offered. The P-MAPS hypervisor is claimed to launch underneath a running guest OS in 300 ms. This system is similar to TrustVisor at a high-level; however, insufficient detail is available to conduct a careful comparison.

Singaravelu et al. extract the security-sensitive portions of three applications into AppCores and execute them on the Nizza microkernel architecture [30]. While compelling, the trusted kernel contained on the order of 100,000 lines of code, which is an order of magnitude larger than TrustVisor. A more recent result is seL4, a formally verified microkernel [20]. While this work represents a significant step forward, it remains unclear whether it is appropriate for use in conjunction with a legacy OS.

Software-based fault isolation [21, 31, 34] and control flow integrity [1] are mechanisms that insert inline reference monitors. Unfortunately, all of these systems ultimately depend on the security of the underlying OS remaining intact, and cannot tolerate a compromise of the system at this low level. In our system, only TrustVisor is trusted to this extent.

Xen supports virtual TPMs for VMs [4]. Each vTPM instance includes all of Xen, a domain 0 OS, and a software TPM emulator that implements the full suite of TPM functions in its TCB. Though vTPM exposes more features than our μTPM, 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, a reduced μTPM interface, 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, 14, 28], 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 so

much code that one cannot make any statements regarding security properties. Researchers have also shown that the Trusted Computing Group's Static Root of Trust for Measurement [19, 25] can be readily compromised.

Researchers have developed systems to reduce the requisite level of trust in OSes (e.g., CHAOS [7], Overshadow [8], and others [11, 35]). However, the protection granularity in these systems is too coarse to provide strong security properties, because the entire application is in the TCB, as is a hypervisor that is larger than TrustVisor. sHype is an extension to the Xen VMM to enforce coarse-grained Mandatory Access Control policies between VMs [27], but it still includes the full Xen hypervisor in the TCB.

Seshadri et al. develop SecVisor, a small hypervisor that protects kernel code integrity [29]. However, SecVisor cannot protect against many classes of existing vulnerabilities in the protected kernel. TrustVisor is also a small hypervisor, but it sandboxes the legacy OS and provides a trusted environment in which to execute PALs in isolation from the legacy OS and its applications, thereby attaining a much smaller TCB for sensitive code. The protections offered by SecVisor could also be implemented using TrustVisor as the hypervisor, thereby providing defense-in-depth.

9 Conclusion

TrustVisor is a small hypervisor that enables isolated execution of Pieces of Application Logic (PAL) with a TCB containing only the TrustVisor runtime and the PAL itself. This system enforces code and execution integrity, and data secrecy and integrity for PALs. TrustVisor enables fine-grained attestations to the PAL's execution. TrustVisor supports unmodified legacy OSes 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 *Proc. ACM Conference and Computer and Communications Security (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 *Proc. IEEE Symposium on Research in Security and Privacy (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*, 2006.

[5] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2008.

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

[7] 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 VMM. Technical Report FDUPPITR-2007-0801, Fudan University, 2007.

[8] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008.

[9] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[10] 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.

[11] P. Dewan, D. Durham, H. Khosravi, M. Long, and G. Nagabhushan. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proc. Spring Simulation Multiconference*, 2008.

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

[13] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE S&P*, 2000.

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. ACM Symposium on Operating System Principles (SOSP)*, 2003.

[15] D. Grawrock. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[16] Intel Corporation. Intel trusted execution technology – software development guide. Document number 315168-005, June 2008.

[17] J. Katcher. Postmark: A new file system benchmark. Technical Report TR-3022, NetApp, 1997.

[18] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall, 2008.

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

[20] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *ACM SOSP*, 2009.

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

[22] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proc. ACM European Conference in Computer Systems (EuroSys)*, 2008.

[23] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How low can you go? Recommendations for hardware-supported minimal TCB code execution. In *ASPLOS*, 2008.

[24] S. C. Misra and V. C. Bhavsar. Relationships between selected software measures and latent bug-density. In *Proc. Conference on Computational Science and Its Applications*, Jan. 2003.

[25] A. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy. TCG inside? A note on TPM specification compliance. In *Proc. Scalable Trusted Computing Workshop*, 2006.

[26] R. Sahita, U. Warrier, and P. Dewan. Dynamic software application protection. Intel Corporation, Apr. 2009.

[27] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure hypervisor approach to trusted virtualized systems. Technical Report RC23511, IBM Research, 2005.

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

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

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

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

[32] R. Ta-Min, L. Litty, and D. Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *ACM SOSP*, 2006.

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

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

[35] J. Yang and K. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proc. ACM Conference on Virtual Execution Environments (VEE)*, 2008.