Dartmouth College

Dartmouth Digital Commons

Computer Science Technical Reports

Computer Science

8-14-2003

# Bear: An Open-Source Virtual Secure Coprocessor based on TCPA

Rich MacDonald
*Dartmouth College*

Sean Smith
*Dartmouth College*

John Marchesini
*Dartmouth College*

Omen Wild
*Dartmouth College*

Dartmouth Digital Commons Citation

MacDonald, Rich; Smith, Sean; Marchesini, John; and Wild, Omen, "Bear: An Open-Source Virtual Secure Coprocessor based on TCPA" (2003). Computer Science Technical Report TR2003-471. https://digitalcommons.dartmouth.edu/cs_tr/221

# Bear: An Open-Source Virtual Secure Coprocessor based on TCPA

Computer Science Technical Report TR2003-471

Rich MacDonald, Sean Smith, John Marchesini, Omen Wild
Department of Computer Science/Dartmouth PKI Lab*
Dartmouth College
Hanover, New Hampshire USA
`http://www.cs.dartmouth.edu/~pkilab/`

August 14, 2003

**Abstract**

This paper reports on our ongoing project to use TCPA to transform a desktop Linux machine into a virtual secure coprocessor: more powerful but less secure than higher-end devices. We use TCPA hardware and modified boot loaders to protect fairly static components, such as a trusted kernel; we use an *enforcer* module—configured as Linux Security Module—to protected more dynamic system components; we use an encrypted loopback filesystem to protect highly dynamic components.

All our code is open source and available under GPL from `http://enforcer.sourceforge.net/`.

## 1 Introduction

In our lab, we have been trying to produce practical solutions to the trust problems endemic to the current computational infrastructure.

In the big picture, many of these problems center on how to trust what is happening on a given computer. A party Alice may care about whether certain correctness properties hold for computation. However, computation occurs on machines. Consequently, whether Alice can believe whether a given computation has some critical correctness property depends, in general, on whether Alice can trust that machine. Is the machine really executing the right code? Has critical data been altered or stolen? Is the machine connected to the appropriate real-world entity?

In the modern computing environment, these machines have become increasingly complex and distributed. Both factors complicate this trust decision:

- *Distributed.* How can Alice know for sure what is happening at a remote machine, under Bob's control?
- *Complexity.* Given the inevitable permeabilities of common desktop machines, how can Alice even know what's happening at her own machine?

*This research has been supported in part by the Mellon Foundation, NSF (CCR-0209144), AT&T/Internet2 and the Office for Domestic Preparedness, Department of Homeland Security (2000-DT-CX-K001). This paper does not necessarily reflect the views of the sponsors. The last three authors can be reached via addresses of the form `firstname.lastname@dartmouth.edu`

The notion of *secure coprocessing* has long been touted (e.g., [26, 27, 18]) as a solution to this problem: a careful interweaving of physical armor and software protections can create a device that, with high assurance, possesses a different security domain from its host machine, and even from a party with direct physical access. Such devices have been shown to be feasible as commercial products [20, 4] and can even run Linux and modern build tools [9]. In our lab, we have explored using secure coprocessors for trusted computing—both as general designs (e.g., [13]) as well as real prototypes (e.g., [11])—but repeatedly were hampered by their relatively weak computational power. (Their relatively high cost also inhibits widespread adoption.)

In some sense, secure coprocessors offer high-assurance security at the price of low performance (and high cost). However, in industry, two new *trusted computing* initiatives—TCPA [21, 22, 23, 14] and the former Palladium [7, 5, 6] have emerged that target a different tradeoff: lower-assurance security that protects an entire desktop platform (thus greatly increasing the power of the trusted platform) and is cheap enough to be commercially feasible. Indeed, the TCPA technology has been available on various IBM platform, and other vendors have discussed availability.

Many in the field ([1] is a notable example) have criticized the technology for its potential negative social effects; others (e.g, [8, 16, 15]) have seen positive potential.

**This Project.** When we started this project, commercial applications of TCPA had been confined to Windows operating systems and closed software; so we decided to start with Linux, the TCPA specifications [23, 22], and IBM's open source driver for the TPM—and prototype a complete, fully-functional open-source solution. (Initially, this began as the first author's senior thesis.) Our goals were two-fold: both to provide a foundation for further trusted computing projects in our lab, as well as to address a gap—open source use of TCPA—that we did not see addressed in the field.

However, in July 2003, while we were completing final testing of our TCPA library, IBM published some open-source tutorial code [10]; in August, a companion article provided further discussion [17]. Hence, we decided to not delay any further, and offer:

- our TCPA code
- our modified boot loader and Linux Security Module (LSM)
- and our associated design ideas.

Our code is available for download, under GPL.

**This Paper.** Section 2 discusses our overall security goals. Section 3 reviews TCPA. Section 4 presents the design we developed, using TCPA, to achieve these goals. Section 5 discusses how we implemented this design. Section 6 discusses some avenues for future work.

# 2 Security Goals

## 2.1 Motivation: Remote Servers

When designing security systems, it's useful to start by considering what it is one actually wants to achieve.

We'll start by considering the remote case. Our lab has a focus on PKI; perhaps the most tangible instance of PKI-based trust judgment is SSL-protected Web servers. Let's consider the steps by which the current infrastructure permits Alice to make this trust judgment.

- Alice desires some service $X$.

- Alice knows, by magic, the binding between that desired service and the identity of the server $S_X$ that provides it.
- One[1] of the CAs whose public key is built into her browser as a trust root has signed a certificate binding the identity $S_X$ to a public key.
- Alice's browser correctly informs her when it has carried out an SSL handshake with a server that knows the private key $d$ matching that public key, and that the rest of the session occurs within that channel. (This is trickier than one might suspect [25].)

From this reasoning, Alice might conclude that her Web session is with service $S_X$, and that she receives service $X$. However, this conclusion also rests on two other facts with which the infrastructure does not help:

- Server $S_X$ is the only party that knows the private key $d$.
- Server $S_X$ will in fact provide service $X$, with all its implicit properties.

These facts might fail to hold, via accident or malice. By accident (such as server misconfiguration), $S_X$ might permit another party to learn the private key $d$, to change the data provided in service $X$, or to download the credit card numbers and order information that $S_X$ accumulates as part of providing $X$—but which were supposed to be confined to $X$. By malice, an unscrupulous $S_X$ might itself break such privacy or correctness rules for $X$. Alice has no way of knowing.

**Prior Work.** In prior work [12, 19], we tried to address this problem by moving the private key $d$ and the service $X$ into a secure coprocessor co-located at the server $S_X$. This technology then permits an additional step:

- In addition to binding the identity to a public key, the CA certifies that the private key $d$ lives safely inside the armored card, which uses it for $X$ and nothing else, even if $S_X$ would like to cheat.

(Alice's browser's trusted path would also have to communicate this special certification to Alice somehow.)

Even though we prototyped it, this approach never found in-the-field use. For our team, the primary obstacle was the awkwardness of the programming environment. The limited internal code space meant considerable rewriting of the standard SSL suite; our rewrites then needed to be upgraded with each upgrade of Apache, etc. Had we progressed to actual deployment, we would have confronted a different obstacle: exactly how a CA would verify the nuances of a service $X$.

**With TCPA.** A TCPA solution would let us extend this armor to the entire server. This would overcome the codespace problem, but leave us with the other obstacles. Alice needs to conclude that she'll receive service $X$ from server $S_X$, because the party on the other end of the wire knew the private key $d$, and a CA said something about that public key, $X$, and $S_X$. We can design some *hardware and software magic (HSM)* on top of TCPA to bind $d$ to something.

However, this leaves us with a question: to what can we bind $d$, that lets a CA make some statement, that enables Alice to make a reasonable trust conclusion?

A natural but naive approach is to extend the identity-keypair binding.

- The CA chooses some software suite—particular versions of the OS, Apache, mod_ssl, CGI scripts, Web pages, etc—that it regards as meeting service type $X$.

[1] Actually, it could be more than one; but she implicitly trusts that all the CAs who perform this binding do it correctly, to a public key that really belongs to server $S_X$.

- The CA certifies the keypair when the CA believes that the hardware and software magic will restrict the private key exactly to one instance of that suite.

We note that, if the CA neglects to certify anything about the Web content, then the point of the remaining HSM is not clear—we have the proverbial armored car to a cardboard box (except the cardboard box is no longer the entire server, but just the content the server is serving).

The naivete of this approach is obvious to anyone who has ever tried to deploy a system or a Web site in the real world.

- The software will not be static. For bug fixes and security patches alone, various elements of the suite will have to be upgraded (and perhaps sometimes downgraded) over time. *The promise of responsibly maintaining a secure site requires that the executable suite, considerd as a whole, be dynamic.*
- The Web content will not be static. Some of us have direct experience with industrial sites where much content would change daily.

In some sense, everything is dynamic, even server keypairs. However, in current PKI paradigms, a certificate binds an entity to a keypair for some relatively long-lived period. But if this entity is to be something a like a Web server with particularly identity offering some type of service, the entity will have to change in ways that cannot be predicted at the time of certification.

### 2.2 Requirements

From the above discussion, what can we conclude about what our Hardware and Software Magic should provide?

- **Secure Storage.** We need to provide a way to store data that's accessible only to a specific software entity.

  Besides confidentiality, we also need to provide freshness and integrity: the data the entity stores should be the data it retrieves.
- **Authentication.** This entity should be able to prove who (and what) it is, based on a certificate issued for a long-lived keypair.
- **Maintenance.** We need to allow the configuration of this entity—its software (including all security-relevant software on the platform) and operational data—to be able to undergo authorized modifications, while still retaining access to its secrets.
- **Usability.** Our platform should look and act like as much like a standard open-source platform. The HSM should be easy to use!

Obviously, the security of our platform will be depend on the security of the commodity software and hardware tools we start with. These tools—particular TCPA—will thus dictate the attack model.

## 3 TCPA Background

### 3.1 Overview

A multi-vendor consortium, the *Trusted Computing Platform Alliance* produced a series of specifications for a hardware addition to the standard computing platforms that adds some additional security functionality. The term *TCPA* has come to be identified with the design that this alliance produced.

The Microsoft initiative formerly known as Palladium also seeks to use a hardware addition for additional security functionality. The exact relation between TCPA and the former Palladium is not clear; one suspects that at some point in the TCPA design process, Microsoft decided to withdraw and build their own variant.

In TCPA, the basic idea is to add a *Trusted Platform Module (TPM)* to the machine; this TPM then assists in authenticating the software configuration and providing a credential store. In the TCPA-enabled IBM machines currently available, the TPM is a smart-card like chip mounted on the motherboard; rumor has it that vendors are incorporating the TPM into the CPU itself.

Unfortunately, TCPA was designed by committee, and it shows. The specification documents [23, 22] are large, complex—and buggy and incomplete, as we discovered. (We found it interesting that the specification text often trumpets "end of informative comment" but then keeps going.) A subsequent book from HP engineers [14] helps somewhat. (As noted earlier, we did not have the benefit of the recent IBM tutorial material [10, 17].)

In this section, we try to present the basic building blocks that TCPA gives us.

## 3.2 The Trusted Platform Module

The heart of the TCPA design is the TPM. The TPM provides services to its hosting machine. The TPM also provides services to two special entities:

- the *owner* (not necessarily the machine owner or user), who authorizes commands via an HMAC derived from a secret 20-byte key.
- an *operator* who authorizes commands via *physical presence*, which the specs suggest might be via jumper cables.

The initial set of challenges facing a TCPA experimenter is how to wade through the relevant commands in order to enable the TPM and take ownership.

## 3.3 PCRs and Hashes

The TPM has a series of *platform configuration registers (PCRs)*. Each is 20 bytes long, the length of a SHA-1 hash.

These PCRs are initially zeroized at boot time; the machine can then *extend* a given PCR by writing a value $v$; the TPM will concatenate this $v$ to that PCR's current value, hash the concatenation, and store the result in the PCR. This extension feature permits a single PCR to record (essentially) an arbitrary length sequence of values. This feature also provides a convenient "ratcheting" feature: adversarial software cannot roll back a PCR to a value it held earlier during the execution.

The TPM can perform SHA-1 hashing. During boot time, the BIOS measures itself and reports that to the TPM. (Hence, the BIOS must be trusted; the specifications refer to this as *Root of Trust Management (RTM)*.) The BIOS feeds the *Master Boot Record (MBR)* to the TPM to hash before passing control to it. Subsequent software components are expected to hash their successors before loading them. These hashes are stored in PCRs.

The TPM we use has 16 PCRs; the TCPA PC specification reserves eight of them for specific purposes, leaving eight for us.

## 3.4 Credential Storage

The TPM provides a *protected storage* service to its machine.

From the programming perspective:

- One can ask the TPM to *seal* data, and specify a subset of PCRs and target values; the caller also specifies a 20-byte authorization code for this object. The TPM returns an encrypted blob (with an internal hash, for integrity checking).
- One can give an encrypted blob to the TPM, and ask it to *unseal* it. The caller must prove knowledge of the 20-byte authorization code for this object. The TPM will release the data only if the PCRs specified at sealing now have the same values they had when the object was sealed (and if the blob passes its integrity check).

TPM protected storage can thus bind secrets to a particular software configuration, if the PCRs reflect hashes of the elements of this configuration. The TPM also has the ability to save and report the PCR values that existed when an object was sealed.

This protected storage design suffers from an apparent oversight: if the caller would like to change the value of one the PCRs in the release policy for an object (for example, because the OS kernel has been upgraded), the caller must export the object in plaintext to the host (before upgrade, while the old PCR policy is satisfied), then re-save it.

Internally, the TPM provides protected storage by building up a tree of private keys, starting with an internal *storage root key (SRK)*. Nodes themselves are protected blobs; the arbitrary data items themselves are leaves in this tree. As a side-effect, protected data items are limited to the length of the RSA modulus: 2048 bits.

Many of the TPM storage commands thus deal with manipulation of these keys.

The TPM has the ability to perform RSA operations internally. Besides enabling management of the key tree, this feature permits the TPM to do private-key operations with stored objects that are private keys (if the PCRs and authorization permit this) without exposing the private keys to the host platform.

## 3.5 Data Integrity Registers

Research into booting a system securely has a long history (e.g., [2, 3, 24]). TCPA builds on this history, but gives special definitions to terms that might otherwise sound synonymous:

- In TCPA, *authenticated boot* is when the system can prove what software actually booted on the system (e.g., by proving knowledge of a secret bound to PCRs that reflect the boot sequence). (Some researchers also use *trusted boot* for this concept.)
- In TCPA, *secure boot* is when the TPM actually prevents the platform from booting if the software sequence does not match some specified hashes.

The TCPA literature is emphatic about this special meaning of “secure boot” but does not give any details on how the TPM actually causes the platform to stop booting, how much of the software sequence is checked, or how to tell if the TPM in our IBM machine actually does this.

However, to support this vague functionality, the TPM includes *data integrity registers (DIRs)*, 20 bytes long, to hold the critical hashes. Writing to a DIR requires owner authorization; reading can be done by anyone.

The TPM in our IBM Netvista 8310 has one DIR; as far as we can determine, it does not actually do anything.

### 3.6 Other Functionality

TCPA provides additional functionality, for tasks like proving a TPM is authentic, *attesting* to the software configuration of a machine, and for manufacturer-assisted maintenance. Proving authenticity is a multi-step process, in order to dissociate use of a TCPA machine from purchase of a TCPA machine.

### 3.7 Adversary Model

So, what does the TPM protect against?

TCPA cannot protect against fundamental physical attacks. If an adversary can extract the core secrets from the TPM, then they can build a fake one that ignores the PCRs. If an adversary can manage to trick a genuine TPM, during boot, to storing hash values that do not match the code that actually runs (e.g., perhaps with dual-ported RAM), then secrets can be exposed to the wrong software. If the adversary can manage to read machine memory during runtime, then they may be able to extract protected objects that the TPM has unsealed and returned to the host.

However, the TPM can protect against attacks on software integrity. If the adversary changes the BIOS or critical software on the hard disk, the TPM will refuse to reveal secrets; the software so verified can then verify (via hashes) data and other software. Potentially, the TPM can protect against runtime attacks on software and data, if onboard software can hash the attacked areas and inform the TPM of changes.

Note that, unless we take additional countermeasures, the TPM design appears to permit a class of *replay* attacks. Suppose a protected object has value $v_0$ at time $t_0$ and value $v_1 \neq v_0$ at time $t_1 > t_0$. If the adversary makes a copy of the hard disk at time $t_0$, the adversary can restore the value $v_0$ by powering down the system and loading the old copy. For some applications, this attack can have serious ramifications (e.g., it might permit the adversary to restore revoked privileges or spent e-cash, or roll back a security-critical software upgrade).

## 4 Architecture

How can we use the elements of Section 3 to satisfy the requirements of Section 2?

### 4.1 Secure Storage

We can use the TPM's protected storage services to bind stored secrets to a given software entity in a specific configuration.

Since keeping an RSA private key inside the TPM provides an extra level of protection that some programmers might want to exploit, we should expose that option.

Although the TPM provides confidentiality and integrity for stored data, it does not provide freshness (as Section 3.7 above discussed). In order for the trusted software entity to verify that its stored secrets are fresh, it needs a place to store something that the adversary cannot rewrite. In the general case (e.g., without adding a 4758 or multiple TCPA platforms), the only place we have is the one DIR in our platform.

Using the DIR for this purpose requires that a trusted piece of software on the platform itself know the owner authorization code. Of course, this code itself could be saved as a TPM protected object; any commands that a genuine remote owner would need to authorize could be done via proxy: the owner authorizes to the software, which in turn constructs the command.

Many protection schemes are possible. A scheme in the spirit of the the DIR's alleged use is to maintain a *freshness table* of hashes (or otherwise concise expressions) of the most recent versions of appropriate objects. When an entry in the freshness table is updated, we save the hash of the updated table in the DIR.

However, this creates a problem: an adversary who can get root access can learn the owner authorization code, and then completely subvert the freshness defense. An interim solution here is to provide this freshness checks only for data modified at boot-time:

- The authorization code is bound, via PCRs, to a trusted boot-time entity.
- If this entity needs to update the DIR, it unseals the code, uses it, and securely erases it from RAM.
- The entity then extends a key PCR hash with a known value—so the PCR value now reflects "this entity, but after it has put away its secrets."

It is interesting to note the similarity of this approach to the "ratchet locks" in the IBM 4758 [20].

An effective, simple, elegant solution to freshness of run-time user storage is an area of future work. (Some combination of hashing, DIRs, and the PCR values at sealing might work.)

## 4.2 Authentication and Maintenance

We consider the next issues together:

- How do we permit the software that constitutes this entity to be maintained, while retaining the entity's TPM secrets?
- How do we permit a CA to express something in a certificate that says something meaningful about the trustworthiness of this entity over future changes—both to software as well as to more dynamic state (e.g., Web pages)?

Everything is dynamic. To address this problem, we decided to organize system elements by how often they change:

- the relatively long-lived core kernel
- more medium-lived software
- short-lived operational data

We then add two additional items to the mix:

- a remote *security admin*, who controls the medium-lived software configuration, via public-key signatures.
- an *enforcer* software module that is part of the long-lived core

The security admin provides a signed description of the medium-lived software. For simplicity, the public key can be part of the long-lived core (although we could have it elsewhere). A security admin's signed descriptions could apply to large sets of machines. In theory, the security admin may in fact be part of a different organization; e.g., Verisign or CERT might set up a security admin who signs descriptions of what are believed to be secure configurations of Apache and SSL on Linux.

The TCPA boot process ensures that the long-lived core boots correctly and has access to its secrets. The enforcer (within the long-lived core) checks that the security admin's description is correctly signed, and that the medium-lived software matches this description. The enforcer then uses the secure storage API to retrieve and update short-lived operational data, when requested by the other software.

Since these protected secrets are bound to the enforcer and long-lived core, we avoid the TPM update problem.

To prevent replay of old signed descriptions, the security admin could include a serial number within each description, as well a "high water mark" specifying the least serial number that should still be regarded as valid. The enforcer saves a high-water mark as a field in the freshness table; the enforcer accepts a signed description only if the serial number equals or exceeds the saved high-water mark; if the new high-water mark exceeds the old, the enforcer updates the saved one. (Alternatively, the enforcer could use some type of forward-secure key evolution.)

A CA who wants to certify the "correctness" of such a platform essentially certifies that:

- The long-lived core operates correctly.
- The named security admin will have good judgment about future maintenance.

## 4.3 Usability

How should we flesh out the above elements? In order to make our system usable, we should try to choose designs that coincide with familiar programmer constructs. (If possible, these choices may also make our system easier to build—since we can re-use existing code!)

**Short-Lived Data.** For short-lived data, we want to give the programmer a way to save and retrieve non-volatile data whose structure can be fairly arbitrarily.

In systems, the standard way that programmers expect to do this is via a filesystem. A *loopback filesystem* provides a way for a single file to be mounted and used as a filesystem; an *encrypted loopback filesystem* allows this file to be encrypted (and presumably integrity-protected) when stored.

So, a natural choice for short-lived data is to have the enforcer save and retrieve keys for an encrypted loopback filesystem, and retain its hash in the freshness table. (A remaining question is how often an update should be committed.)

Since the TPM provides a way to use RSA private keys without exposing them, we should also provide an interface to do that.

**Medium-lived Software.** For the medium-lived software, we need a way for a (remote) human to specify the security-relevant configuration of a system, and a tool that can check whether the system matches that configuration.

We chose an approach in the spirit of Tripwire. The security admin (again, perhaps on a different machine) prepares a signed description of the configuration of this medium-lived component; the long-lived component of our system will use this signed description to verify the integrity of the medium-lived component.

(We considered also performing this function with an encrypted loopback filesystem, but then decided that the relevant aspects of the security configuration would be too hard to handle that way.)

**Long-lived Core.** Another question is how to structure the enforcer itself. The natural place was as an LSM—besides an being the standard framework for security modules in Linux, this choice also gives us the chance to mediate (if the LSM implementation is correct) all security-relevant calls—including every inode lookup.

We envisioned this enforcer module running in two step: an initialization component, checking for the signed configuration file and performing other appropriate tasks at start-up, a run-time component, checking the integrity of the files in the medium-lived configuration.

# 5 Implementation

## 5.1 Library

The first step in making this real was wading through the specifications to understand what the functionality of the TPM, and then—with the open source driver—writing code to actually exploit that functionality.

This component took the longest amount of calendar time. Using the TPM is not a simple matter of a function call: sessions and HMAC'd data must be formatted in the proper way; the formats are the wrong Endian value for an X86 platform; and mistakes often generated a non-informative error code.

The library we generated is similar to the one IBM concurrently developed: we translate high-level, human-understandable calls into properly formatted TPM commands, which we then send to the TPM. As noted, our code—with extensive comments—is available via GPL.

We use this code to initially take ownership of the TPM; we also use it when the system—once we've reached run-time and have an OS—needs to communicate to the TPM.

## 5.2 Boot Loader

Once our TPM was functional, the next step is to integrate it into the boot process.

Recall that the TPM is (from one view) essentially a credential store that guards secrets based on PCR values. If we want to ensure that secrets are available only to a specific trusted software configuration at run-time, we need to make sure that the TPM can witness—via hashes in the PCRs—each element in the sequence of executables that leads to that configuration.

The first step in this chain is BIOS. Per the PC-specific TCPA specification [22], BIOS in a TCPA-enabled PC will report itself to the TPM. BIOS will also hash the *master boot record (MBR),* and report this to the TPM, before passing control to it. (Be sure to have the latest BIOS update before starting experiments here; we lost some time due to the older version that shipped with our machine.)

The next step in the chain is to modify the first-stage bootloader in the MBR to SHA-1 hash the next component and report this to the TPM, before passing control. Currently, we store this hash in PCR 8; we used the memory-present TPM driver from IBM.

We started with the LILO loader; so we modified `first.b` (the MBR in LILO) to hash `second.b`. Doing this in assembly, to fit within the tight confines of the MBR and handle the TPM endianness requirements, was tricky.

In our current prototype, we run our TCPA-enabled LILO from a floppy. This decision stemmed from two reasons.

- The first was codespace—and TPM bugs. An MBR is 512 bytes; but a hard disk MBR also contains other data, and does not give us the full 512 for code. This would not have been a problem, except the TPM in our machine did not appear to actually support the `TCPA_HashLogExtendEvent()` call—we kept getting a "call not implemented" error. The workaround—replacing this call with a sequence of calls—pushed us over the limit for the hard disk MBR. (The floppy gives us the full 512 bytes.)

- The second is more pragmatic. This is alpha-quality software, and bugs in LILO can destroy the MBR, making the system unbootable without a rescue disk. With LILO on the floppy, hard disk MBRs (and the entire `/boot` directory) do not require any changes, and disasters are avoided by removing the floppy and booting as usual.

We modified the second stage bootloader to hash the compressed kernel footprint and place that hash in a PCR. (Our implementation currently places this value in PCR 9.) The rest of the boot process can continue as normal.

At this point, PCRs 0-9 now witness that this particular kernel was booted in a trusted fashion. (The vendor uses 0-7; and PCR 8 and PCR 9 have our new hashes, noted above.) If the boot process is modified, these PCRs will contain different values.

## 5.3 Admin Tools

We wrote some Perl scripts to produce the configuration files, and used an open-source bigint package to produce a rudimentary keygen (2048-bits) and signing tool. For each file, the security admin can specify what should happen if its integrity check fails: log, deny, or panic.

We also used this produce a stripped-down verification tool, for inclusion in the enforce kernel module (discussed below).

## 5.4 Enforcer LSM

As mentioned, we built our enforcer as a LSM, for the 2.4 kernel with the LSM 2.4.20-1 kernel patch. The initial prototype is about 1000 lines of code. Our code is set up either to be compiled into the kernel or to be loaded as a separate module; the former makes sense for real deployment; however, the latter makes experimentation easier.

The enforcer uses the `/etc/enforcer/` directory to store its signed configuration file, public key, etc. (Having the kernel store data in the filesystem is a bit uncouth, but seemed the best solution here, and is not completely unprecedented.)

When the kernel initializes the enforcer, it registers its hooks with the LSM framework. If built as a loadable module, the enforcer verifies the configuration file's signature now; if compiled into the kernel, the enforcer verifies it when the root filesystem is mounted.

At run-time, the enforcer hooks all inode lookups. (These happen as a file is opened.) We check the file's integrity via the configuration file; if the integrity fails, we react according the option: log the event to the syslog, fail the call, or panic the system.

We developed the enforcer under *user-mode Linux (UML)*, which worked very nicely—each bug that appeared under UML also showed up with the real system, and vice-versa, We ran basic functional tests—showing that modifying the configuration file, public key, signature, or any protected file actually causes the appropriate reaction. We also ran 36 hours of continuous stress-tests; the code showed no signs of crashing or leaking memory.

Performance testing (against 1G of data and about 60,000 files) shows a 25% slowdown on boot, and a 11% slowdown at subsequent runtime.

## 5.5 Storage Services

As of this writing, we plan to install the following in the second alpha release.

First, we'll implement a simple freshness table as discussed above, for items updatable only at boot-time by the enforcer. This means that the owner authentication code will be a sealed object, for the enforcer to access; this also means that when the enforcer finishes its initialization, it extends PCR 9 in a known way to indicate that the system is still founded on the trusted enforcer, but its core secrets are put away.

This freshness table will store two items: a hash of the security admin public key, and the current high-water mark of the security admin's configuration files. The signed configuration files include entries for serial number, high-water mark, and optional new public key. At start-up, the enforcer will verify data:

- It checks that its freshness table hashes to the DIR.
- It checks that the stored public key hashes to the entry in the freshness table.
- It checks that the serial number in the configuration file is not below the high-water mark in the freshness table.
- (It also verifies the signature, naturally.)

The enforcer may also modify data:

- If the high-water mark in the configuration file exceeds the stored one, the enforcer needs to update the stored one.
- If configuration file included a new public key, the enforcer needs to store that—and update the hash in the freshness table.

These changes need to be flushed back to the DIR.

This way, we permit old public keys and old signed configurations to be revoked.

The enforcer also needs a suicide option, to make sure all secrets are unaccessible in the case of panic. Extending core PCRs should do the trick.

We then plan to expose the rest of the TPM calls in our library, for use by user-level code, such as encrypted loopback filesystem, as it sees fit.

## 5.6 Long-term Vision

In the long-term, we would like this platform to balance security and practical issues. For example, one motivation was to enable in practice what WebALPS only enabled in theory: a way to secure the other end of an SSL tunnel.

The TPM testifies to the long-lived component: the hardware and BIOS, the kernel and current enforcer, and the security admin's current public key.

The security admin then testifies to the medium-level software (e.g., the particular versions of Apache, mod_ssl, etc) necessary for the system to run securely. The enforcer (already checked) ensures that this configuration matches the system.

The Web content is controlled by various users. These users are authenticated via the kernel and medium-level configuration that has already been testified to. Their content is saved in a protected loopback filesystem, ensuring that it was valid content at some point. (We still need an adequate solution to freshness.)

The SSL CA verifies that it trusts the enforcer and kernel, and the judgment of the security admin. The relying party then has reason to trust the other end of the transaction.

# 6 Future Work

As we noted, this is a preliminary report about a work-in-progress. Many areas remain for future work. We quickly discuss some:

- **Freshness.** As noted earlier, it would be nice to have an elegant, effective solution for freshness of secrets stored at run-time, and by user-level code. This would also raise the issue of how often changes should be committed—for example, should *each* change to a loopback filesystem update a DIR somehow?
- **Personalization.** We have not yet thought deeply about a how a virgin system becomes configured and under the control of a security admin.

  So far, we have waved our hands and assumed that what Peter Gutmann terms the "Baby Duck Model" (the system imprints on the first public key it sees) would suffice.
- **Attestation.** We have also not addressed the task of enabling a remote party to verify that a blessed kernel and enforcer are running on a particular system. For now, we assume that the TPCA attestation framework should work.
- **Sufficiency of Configuration Checking.** Right now, the enforcer only protects against modifications of content of the files that the security admin deemed critical. The intention is that the integrity of the kernel/enforcer plus the integrity of this code is enough to ensure the system is working as advertised, to the best of the security admin's knowledge. We plan further thought as to whether this is sufficient.
- **Run-time Defenses.** Again, the enforcer only looks at modifications to file contents. Adversaries may mount many other types of attacks; since the long-term goal is a virtual secure coprocessor, it would be interesting to extend the enforcer to detect additional types of tampering.

  (An adversary that succeeds at run-time in modifying the enforcer code itself or its configuration file might also have some success.)
- **Credential Use, in Practice.** As real applications get written and run, it would be interesting to see how many make use of an encrypted loopback filesystem for secure storage, and how many instead make use of the TPM sealed storage calls directly.
- **Performance.** We have not yet even begun to consider issues of tuning performance, such as marking files that have passed the integrity check—but clearing this flag if we catch an operation that might modify the file.

As noted, one of the first things we want to do is port our Apache/SSL-based Dartmouth CA to this platform, to see whether these hooks are sufficient for a real application, whether the performance hit under real loads is significant, and what additional security the resulting the system achieves.

# Acknowledgments

The authors are grateful to Dave Challener and Ryan Cathecart at IBM for answering questions.

# References

[1] R. Anderson. TCPA/Palladium Frequently Asked Questions.
`http://www.cl.cam.ac.uk/users/rja14/tcpa-faq.html`.

[2] W. Arbaugh, D. Farber, and J. Smith. A Secure and Reliable Bootstrap Architecture. In *IEEE Symposium on Security and Privacy*, pages 65–71, 1997.

[3] P. Clark and L. Hoffmann. BITS: A Smartcard Protected Operating System. *Communications of the ACM*, 37:66–70, 1994.

[4] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smith, L.van Doorn, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34:57–66, October 2001.

[5] P. England, J. DeTreville, and B. Lampson. Digital Rights Management Operating System. United States Patent 6,330,670, December 2001.

[6] P. England, J. DeTreville, and B. Lampson. Loading and Identifying a Digital Rights Management Operating System. United States Patent 6,327,652, December 2001.

[7] P. England and M. Peinado. Authenticated Operation of Open Computing Devices. In *Information Security and Privacy*, pages 346–361. Springer-Verlag LNCS 2384, 2002.

[8] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS Support and Applications for Trusted Computing. In *9th Hot Topics in Operating Systems (HOTOS-IX)*, 2003.

[9] IBM Research Demonstrates Linux Running on Secure Cryptographic Coprocessor. Press release, August 2001.

[10] IBM Watson Global Security Analysis Lab. TCPA Resources. `http://www.research.ibm.com/gsal/tcpa`.

[11] A. Iliev and S.W. Smith. Privacy-Enhanced Credential Services. In *2nd Annual PKI Research Workshop*. NIST, April 2003.

[12] S. Jiang, S.W. Smith, and K. Minami. Securing Web Servers against Insider Attack. In *Seventeenth Annual Computer Security Applications Conference*, pages 265–276. IEEE Computer Society, 2001.

[13] M. Periera. Trusted S/MIME Gateways, May 2003. Senior Honors Thesis. Also available as Technical Report TR2003-461, Department of Computer Science, Dartmouth College.

[14] S. Pearson, editor. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall, 2003.

[15] D. Safford. Clarifying Misinformation on TCPA. October 2002.

[16] D. Safford. The Need for TCPA. October 2002.

[17] D. Safford, J. Kravitz, and L. van Doorn. Take Control of TCPA. *Linux Journal*, pages 50–55, August 2003.

[18] S.W. Smith. Secure Coprocessing Applications and Research Issues. Technical Report Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory, August 1996.

[19] S.W. Smith. WebALPS: A Survey of E-Commerce Privacy and Security Applications. *ACM SIGecom Exchanges*, 2.3, September 2001.

[20] S.W. Smith and S. Weingart. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks*, 31:831–860, April 1999.

[21] Trusted Computing Platform Alliance. TCPA Design Philosophies and Concepts, Version 1.0, January 2001.

[22] Trusted Computing Platform Alliance. TCPA PC Specific Implementation Specification, Version 1.00, September 2001.

[23] Trusted Computing Platform Alliance. Main Specification, Version 1.1b, February 2002.

[24] J.D. Tygar and B.S. Yee. Dyad: A System for Using Physically Secure Coprocessors. In *Proceedings of the Joint Harvard-MIT Workshop on Technological Strategies for the Protection of Intellectual Property in the Network Multimedia Environment*, April 1993.

[25] E. Ye and S.W. Smith. Trusted Paths for Browsers. In *11th USENIX Security Symposium*, August 2002.

[26] B.S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, May 1994. Also available as Computer Science Technical Report CMU-CS-94-149, Carnegie Mellon University.

[27] B.S. Yee and J.D. Tygar. Secure Coprocessors in Electronic Commerce Applications. In *1st USENIX Electronic Commerce Workshop*, pages 155–170. USENIX, 1995.