Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear

Computer Science Technical Report TR2003-476[1]

John Marchesini, Sean Smith, Omen Wild, Rich MacDonald
Department of Computer Science/Dartmouth PKI Lab Dartmouth College
Hanover, New Hampshire USA

Abstract. Over the last few years, our group has been working on applications of secure coprocessors—but has been frustrated by the limited computational environment and high expense of such devices. Over the last few years, the TCPA (now TCG) has produced a specification for a *trusted platform module (TPM)*—a small hardware addition intended to improve the overall security of a larger machine (and tied up with a still-murky vision of Windows-based trusted computing). Some commodity desktops now come up with these TPMs.

Consequently, we began an experiment to see if (in the absence of a Non-Disclosure Agreement) we could use this hardware to transform a desktop Linux machine into a virtual secure coprocessor: more powerful but less secure than higher-end devices. This experiment has several purposes: to provide a new platform for secure coprocessor applications, to see how well the TCPA/TCG approach works, and (by working in open source) to provide a platform for the broader community to experiment with alternative architectures in the contentious area of trusted computing.

This paper reports what we have learned so far: the approach is feasible, but effective deployment requires a more thorough look at OS security.

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 a given computation. However, computation occurs on machines. Consequently, whether Alice can believe that 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, which complicates this trust decision. How can Alice know for sure what is happening at a remote machine which is under Bob's control? Given the inevitable permeabilities of common desktop machines, how can Alice even know what's happening at her own machine?

In this arena, a promising tool is *secure coprocessing* (e.g., [30, 43, 44]): 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 [6, 34] and can even run Linux and modern build tools [13]. In our lab, we have explored using secure coprocessors for *trusted computing*—both as general designs (e.g., [23]) as well as real prototypes (e.g., [15])—but repeatedly were hampered by their relatively weak computational power. Their relatively high cost also inhibits widespread adoption, particularly at clients.

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 have emerged: the *Trusted Computing*

[1] This version supersedes TR2003-471 and fixes typos found in the December 4, 2003 version.

Platform Alliance (TCPA) (now renamed the *Trusted Computing Group (TCG)* [22, 37, 38, 39]) and Microsoft's *Palladium* (now renamed the *Next Generation Secure Computing Base (NGSCB)* [7, 8, 9, 10]). These also seem to be tied up with Intel's *LaGrande* initiative [35].

Many in the field ([1, 28] are notable examples) have criticized these initiatives for their potential negative social effects; others (e.g, [12, 24, 25]) have seen positive potential. (Felten [11] and Schneider [27] give more balanced high-level overviews.)

These new initiatives 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 platforms, and other vendors have discussed availability. Some academic efforts [19, 21, 36] have also explored alternative approaches in this "use a small amount of hardware security" space, but no silicon is available for experiments.

This Project. This project started when we made this link. The IBM 4758 coprocessor was too constraining. This new TCPA hardware came installed on our IBM Netvista 8310, and is intended to secure an entire desktop. Can we use the TCPA specifications [38, 39] and this hardware to build a virtual secure coprocessor—that is, to transform the desktop into a more powerful but less secure cousin of the 4758?

This project had several goals. As noted, we wanted to provide a foundation for further trusted computing projects in our lab. We also wanted to address a gap—open source exploration of the TCPA/TCG trusted computing space—that we did not see the field addressing. Rather than just waiting for the dominant software vendor to release its architecture as a fait accompli, researchers could actively experiment with alternative architectures; real technology could replace (or at least enhance) idealogical ranting. Finally, this project—"outside the fence" of a Non-Disclosure Agreement—could provide an independent evaluation of the TCPA/TCG hardware approach.

This Paper. Section 2 discusses our overall security goals. Section 3 reviews the building blocks that the TCPA TPM offers us. Section 4 presents the design we developed, using these building blocks, to achieve these goals. Section 5 reports on our experiences implementing this design. Section 6 evaluates this experience. Section 7 concludes with some avenues for future work.

2 Secure Coprocessor Platforms

2.1 Motivation

We want secure coprocessor platforms because we see many scenarios where Alice and Bob can benefit from Alice being able to trust computation at Bob's computer—even if Bob is motivated to subvert it.

For example, consider the case where Bob operates a Web site S_X offering some service X (e.g., selling bicycle parts, or providing health insurance quotes) with some specific security properties (e.g., Bob won't reveal Alice's credit card number or health information). How can Alice know that Bob's site will provide these properties? Server-side SSL binds a keypair to the identity of Bob's site; server-side crypto hardware can strengthen the claim that Bob's private key stays at Bob's site, but how can Alice bind it to X? (This could give a marketing advantage to Bob: "you can trust my service, because you don't have to trust me.") Moving both the key and the service X into a secure coprocessor co-located at Bob's site provides a potential solution. In addition to binding the identity to a public key, the SSL CA could certify that the private key lives inside the coprocessor, which is used for X and nothing else, even if Bob would like to cheat. [17, 31]

The literature provides many additional scenarios, including client-side examples. For example, Alice might like to use an "e-wallet" payment application [43] from Bob—but this requires that Bob can trust that Alice cannot use a debugger to increase the balance stored in the application.

2.2 Desired Properties

What should a secure coprocessor platform provide in order to enable such applications?

Security. First, the platform must enable a remote relying party to make a reasonable decision to trust an application running on the platform. This requires that the platform actually provide for integrity and freshness of executing code—and integrity, freshness, and confidentiality of data—against a local adversary. This also requires that the platform enable the relying party to distinguish between this application in its trustworthy environment and a malicious impostor.

Usability. Second, developers must be able to use this platform to build and deploy real applications.

This requires that the platform must be easily programmable with modern tools. The platform must also allow easy maintenance and upgrade of its software. Finally, the platform must permit installation and upgrade of software to happen at the end user's site—since forcing software developers to ship hardware stifles innovation, limits user choice, and complicates trust [33].

Interaction. As our prior work has shown, these properties have subtle interactions. For example, how can the platform preserve confidentiality of data—and the ability to authenticate software—if it also permits software upgrade [32]?

2.3 Our Prior Work

Platform. In our prior work, we used the IBM 4758 [34] as our platform (we called this the "Moose" platform due to its high level of armor). The 4758 provided the necessary security properties—including attestation (which the 4758 design called *outbound authentication*). However, its usability was mixed. It provided maintenance and upgrade, end-user installation, and (as noted) an experimental version ran Linux. However, the size and power of the computational environment was extremely limited, developers needed to have public keys signed by IBM, and the attestation scheme was fairly complex.

Experiments. Above, we sketched an approach to use secure coprocessors to address the insider attack problem for Web servers.

We used the 4758 to prototype this approach, but it 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.

Design Problem. Had we progressed to actual deployment, we would have confronted a different obstacle. In the standard SSL model, the relying party (e.g., the end user) expects to make a judgment based on a long-lived certificate speaking about a long-lived server keypair. As we sketched above, we can transform a more complex attestation scheme into this simple model by requiring the SSL CA to use the more complex scheme to verify the right properties about the software that owns the private key, and then issue a simpler-to-use certificate (we call this the *yet-another CA (YACA)* approach).

However, our approach overlooked how to map a long-lived keypair onto short-lived configurations. Must the Web site go back to the CA with each new upgrade to Apache or modification to the Web pages or scripts?

3 TCPA/TCG Building Blocks

What basic building blocks does TCPA give us?

3.1 Main Features

As discussed earlier, the TCPA is a multi-vendor alliance that produced a series of specifications for a hardware addition to the standard computing platforms which adds some additional security functionality. The term *TCPA* has come to be identified with the design that this alliance produced—and has recently become obsolete, as the alliance changed its name to TCG. (We call our platform based on this design "Bear," since it lives in the same space as Moose.)

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.

Unfortunately, TCPA was designed by committee, and it shows. The specification documents [38, 39] 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 Hewlett Packard engineers [22] helps somewhat; toward the end of our development, some additional IBM tutorial material appeared [14, 26].

The Trusted Platform Module. The heart of the TCPA design is the TPM, which provides services to its hosting machine. The TPM also provides services to a special entity: the *TPM Owner* (not necessarily the machine owner or user), who authorizes commands via an HMAC derived from a secret 20-byte key.

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.

At the time we developed our system, the most recent version of the "Main Specification" was version 1.1b, and the most recent version of the "PC Specific Implementation Specification" was version 1.0. The TPM hardware we had in our Netvista (mostly) conformed to these versions of the specifications. We did not have access to the Trusted Software Stack (TSS) specification, as it was only recently released [40].

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 (similar to the 4758's ratchet locks [34]): 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, and the hashes are stored in PCRs.

Our TPM has 16 PCRs; the TCPA PC specification reserves eight of them for specific purposes, leaving eight for applications.

Protected 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 also 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).

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.

It is also possible to create keys which are bound to a specific machine configuration with `TPM_CreateWrapKey`. This alleviates the need to create a key and then seal it, allowing both events to be performed by one atomic operation.

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.

Internally, the TPM provides protected storage by building up a tree of private keys, starting with an internal *storage root key (SRK)*, which is created during the `TPM_TakeOwnership` command. 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 private-key operations internally. Besides enabling management of the key tree, this feature permits the TPM to do private-key operations with other stored objects that happen to be private keys (if the PCRs and authorization permit this) without exposing the private keys to the host platform.

One special use of a TPM-held private key is the `TPM_Quote` command. If the caller is authorized to use that key, the TPM will sign a snapshot of the current values of PCRs.

Another useful feature of a TPM-held key is exposed via the `TPM_CertifyKey` call. This function allows a TPM-held private key to sign a certificate binding a TPM-held public key to its usage properties, including whether it's wrapped, and to what PCR values.

Data Integrity Registers. Research into booting a system securely has a long history (e.g., [2, 5, 41]). TCPA builds on this history, but gives special definitions to terms that might otherwise sound synonymous. In TCPA lingo, *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 contrast, *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 TPM Owner authorization; reading can be done by anyone. The TPM in our Netvista has one DIR; as far as we can determine, it does not actually do anything.

3.2 Attestation

TCPA provides additional functionality, for tasks like proving a TPM is genuine, *attesting* to the software configuration of a machine, and for manufacturer-assisted maintenance.

Credentials. The specification and documentation (such as Chapter 5 in [22]) discusses a suite of credentials: the *endorsement credential*, typically from the TPM manufacturer, testifies that the TPM is genuine; the *platform credential*, typically from the machine's manufacturer, testifies that the particular TPM has been correctly incorporated into a design which conforms to the TCPA specification; the *identity credential*, typically from a *Privacy CA* (TCPA's YACA), states that a specific instance of a trusted platform is a trusted platform with a genuine TPM; the *conformance credential* is a reference to a document that vouches that a particular design of TPM and platform meets the TCPA specification.

Endorsement Keys. For our work, the only thing that our TPM appeared to come with was the endorsement key: a 2048 bit RSA keypair that can either be generated on-chip via the `TPM_CreateEndorsementKeyPair` call or injected by the manufacturer. If the key is generated on-chip, the `CEKPUsed` flag is set to true inside the TPM. Once the key is inside the TPM, either by generation or injection, the `TPM_CreateEndorsementKeyPair` call will never succeed again.

The public portion of the endorsement key (named `PUBEK`) would be included in the endorsement credential (except the Netvista does not appear to have this credential). It can also be read from the TPM by the `TPM_ReadPubek` call. The TPM Owner can disable this call for non-owners via `TPM_DisablePubekRead`, but can still access the key via `TPM_OwnerReadPubek`.

The private portion of the endorsement key (named `PRIVEK`) is used for decryption only. It is never used to generate a digital signature, and it is not used to decrypt arbitrary data. As far as we can tell, it is only used in two operations: `TPM_TakeOwnership` which establishes the TPM Owner and SRK for the platform, and `TPM_ActivateIdentity`, which we discuss below.

Creating an Identity Keypair. The TCPA specification defines an attestation *identity* as a statement that an entity with knowledge of a given secret has previously proven that it is an assembly of hardware and software that complies with the TCPA specification [22].

Quickly summarized[2], the attestation bootstrap essentially consists of five steps:

1. The TPM Owner calls `TPM_MakeIdentity`. This generates a new keypair, and creates an *identity-binding*. This binding is essentially a structure containing the new key, a name for the new entity (any arbitrary string), and the public key of the YACA that will vouch for this identity. This structure is then signed with the freshly generated private key.

2. The TPM Owner calls `TSS_CollateIdentityRequest`. This function would be implemented in the TSS—but at the time of this writing, we are unaware of any public implementations of the TSS; the specification was just released [40]. However, the concept of what this function achieves is quite simple. It packs the identity-binding created above and the credentials for the platform (endorsement, platform, and conformance) into a structure for shipment to the Privacy CA.

[2]For more detail, see Section 9 of the specification [39] and Chapter 5 in the Hewlett Packard book [22].

3. The identity-binding and credentials are given to the Privacy CA. The Privacy CA inspects the identity-binding to ensure that it is the requested CA; if so, the Privacy CA checks the signatures on all three credentials.

   Assuming everything is valid, the Privacy CA still has no way of knowing that the identity-binding was generated by the platform described by the credentials. So, the Privacy CA generates a successful response (called an *attestation credential*), generates a symmetric key, and encrypts the attestation credential with the symmetric key. The symmetric key is then encrypted with the the PUBEK (found in the endorsement credential), the idea being that only the corresponding PRIVEK will be able to retrieve the session key and hence, the attestation credential.

4. Upon receiving the Privacy CA's response, the TPM Owner calls `TPM_ActivateIdentity`, which instructs the TPM to verify that the response was intended for that TPM, and if so, to decrypt the session key (using PRIVEK).

5. The TPM Owner can then call `TSS_RecoverTPMIdentity`, which will recover the decrypted attestation credential from the TPM.

To put it in a nutshell, with the help of the TPM Owner's HMAC key, our TPM can generate a new keypair and sign a request binding this new public key to an arbitrary identity. The YACA can respond by sending a certificate for this new public key—but encrypted so that only the TPM (again, with the help of the TPM Owner's HMAC key) can decrypt it.

Using an Identity Keypair. According to the documentation, the TPM will only use an identity private key for specific functionality, such as `TPM_Quote` and `TPM_CertifyKey`.

3.3 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 or malicious DMA), 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 many attacks on software integrity. If the adversary changes the BIOS or critical software on the hard disk, the TPM will refuse to reveal secrets. Otherwise, the verified software 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). We consider some countermeasures to this class of attacks in Section 4.1.

4 Putting the Blocks Together

How can we use these building blocks to assemble a virtual secure coprocessor?

Let us revisit the motivating example. We'd like to set up Linux, and then Apache with some sensitive service, on our TPM-equipped desktop. We want to use a long-lived SSL certificate to convince a remote client that they can trust this service—even if we would like to subvert it.

This requires that we consider how to use the TPM for secure storage of data (Section 4.1); how to structure the software so that the TPM can meaningfully bind a long-lived SSL keypair to dynamically changing content (Section 4.2); and how we can use the TPM to convince a remote relying party, such as the SSL CA, of this binding (Section 4.3).

4.1 Secure Storage

We can use the TPM's protected (non-volatile) 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 3.3 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 TPM Owner authorization code. Of course, this code itself could be saved as a TPM protected object; any commands that a genuine remote TPM Owner would need to authorize could be done via proxy: the TPM 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 TPM Owner authorization code, and then completely subvert the freshness defense. An interim solution here is to provide this freshness check 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." (Again, it is interesting to note the similarity of this approach to the "ratchet locks" in the IBM 4758.)

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 Software Architecture

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 of software as well as of more dynamic state (e.g., Web pages)?

4.2.1 False Start

One natural approach is to use the TPM to bind the SSL private key to the entire server configuration. The SSL 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; the CA certifies the keypair when the CA believes that the TPM magic will restrict the private key exactly to one instance of that suite.

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. For one thing, 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, considered as a whole, be dynamic.*

Furthermore, the Web content will not be static. Some of us have direct experience with industrial sites where much content would change daily. If the CA neglects to certify anything about the Web content, then the point of the remaining system 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).

4.2.2 Our Approach

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 a particular identity offering some type of service, the entity will have to change in ways that cannot be predicted at the time of certification. 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; and *short-lived* operational data

We add two additional items to the mix: a remote *Security Admin*, who controls the medium-lived software configuration via public-key signatures, and 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, and the named Security Admin will have good judgment about future maintenance.

Figure 1 (in Appendix A) sketches how this trust flows.

4.2.3 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 programming 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 when stored [4].

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 previous work on kernel integrity (e.g., [3, 42]).

The Security Admin (again, possibly 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 choice was as a *Linux Security Module (LSM)*—besides 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 and insmod call.

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

4.2.4 Protection from Local Attacks

The above structure may enable the relying party to trust in the identity of the kernel and Enforcer installed on our server. However, trusting the identity does not imply trusting the software itself!

Clearly, this software should be vetted for compliance with good practice in input validation and other defenses against penetration. However, we should call attention to two other issues that require further examination.

First, we need to protect the software and its secrets from the "root" user. For example, a simple test on Linux shows that, without further countermeasures, the root user can manipulate the memory space of other processes with a debugger. A more serious examination of this issue is required. (We avoided it in the 4758, since it had no notion of external user, root or otherwise.)

Second, as noted in Section 3.3, another limitation of the TPM involves a *Time-Of-Check-Time-Of-Use (TOCTOU)* issue between when the hash is checked and the code (or data) is executed/used. One potential avenue for such an attack involves waiting until an integrity check for some program/data item occurs, and then injecting code/data into that process's address space via DMA over a Firewire bus [29]. This attack would appear to require an IEEE1394 card that supports the IEEE1394 OHCI specification for hosts to allow other hosts access to physical memory, and patched system software (the firewire driver, which doesn't exist for Linux yet)—which the Enforcer should catch.

4.3 Bootstrapping Trust

How do we prove to the SSL CA that we have a properly configured kernel/Enforcer suite in our server?

With the 4758's outbound authentication API, we could just install this software, which could then have the platform generate a keypair certified (via a chain of certificates going down through the code-loading firmware and back to IBM) to belong to that software on that device.

Upon careful reading of the specification, it appears the TPM can provide equivalent functionality. Our Enforcer module (with an arbitrary TPM Owner—perhaps even itself, for the freshness trick) has the TPM create an identity keypair and obtain an identity certificate with a YACA. Our module then uses the TPM to create a wrapped keypair bound to a configuration which includes itself—and then has the TPM use the identity private key to certify that fact. The module then needs to return to an SSL CA (which could be the same YACA) with the identity certificate and the certificate created for this wrapped keypair, in order to obtain a standard SSL certificate.

4.4 Example: Apache

In our motivating example, the platform might enable in practice (except for the concerns of Section 4.2.4 and Section 4.3) what our prior work only enabled in theory: a way to secure the entire server end of an SSL tunnel.

The idea is that if someone tampers with a binary on the server which is listed in the Security Admin's configuration file, no one will be allowed to establish an SSL session with the Web server. Conversely, if a client successfully establishes a connection with the SSL server, it has reason to believe that the server has not been tampered with in a way which is unacceptable to the Security Admin to which the CA delegated configuration judgment.

In our scheme, the TPM testifies directly, through use of PCRs, to the long-lived components of our server: 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 (i.e., the particular versions of Apache, mod_ssl, configuration, etc) necessary for the system to run securely. The Enforcer (already checked) ensures that this configuration matches the current 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.

Last, the SSL private key lives inside of the encrypted loopback filesystem. A symbolic link places a reference to the key in a place where Apache would normally look for it. Should the Enforcer detect a

tamper, the loopback is unmounted. The result is that the symbolic link is broken and Apache can no longer access the private key, and can therefore no longer establish SSL connections.

(Again, Figure 1 in Appendix A sketches this structure.)

5 Implementation Experience

We quickly discuss some of our implementation experiences; our preliminary report [20] has more details.

TPM Library. 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.

Boot Loader. As noted earlier, the first step in this chain is the BIOS. Per the PC-specific TCPA specification [38], The BIOS in a TCPA-enabled PC will report itself to the TPM. The 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. 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 in part from 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.)

Admin Tools. We wrote some utilities to produce the configuration files, and used an open-source big integer package to produce a rudimentary key generation (2048-bits) and signing tool [16]. For each file, the Security Admin can specify what should happen if its integrity check fails: log, deny, or panic. We also used this to produce a stripped-down verification tool, for inclusion in the Enforcer kernel module.

Enforcer LSM. As mentioned, we built our Enforcer as a LSM, for the 2.6 kernel (or a 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, the Enforcer registers its hooks with the LSM framework. If built as a loadable module, the Enforcer verifies the configuration file's signature at load-time; if compiled into the kernel (i.e., for normal use), the Enforcer verifies it when the root filesystem is mounted.

At run-time, the Enforcer hooks all inode lookups (which happen as a file is opened). It checks the file's integrity via the configuration file; if the integrity fails, it reacts according the option: log the event to the system log, 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.

Attestation Tools. We wrote a number of small executables which make some of the TCPA calls necessary for attestation—`TPM_MakeIdentity` and `TPM_ActivateIdentity` (as discussed in Section 3.2). Since no Privacy CA or TSS exists, we had to simulate some of their functionality in order for the calls to work. Specifically, we had to extract `PUBEK` and use it to encrypt a session key which is the input to `TPM_ActivateIdentity`.

What's Next. We are currently working on the following enhancements to our prototype.

First, we are extending to the Enforcer to do the same level of integrity checking performed by Tripwire. Currently, the Enforcer only uses the contents of the file (via a SHA-1 hash) to determine integrity. The Tripwire tool uses a number of other things to determine integrity, such as directory contents, permissions, file access times, and other items contained in the file inode structure [18].

Second, we are implementing a freshness table as discussed in Section 4.2.2, for items updatable only at boot-time by the Enforcer. This means that the TPM Owner authentication code will be a sealed object for the Enforcer to access; this also means that when the Enforcer finishes initialization, it extends a PCR in a known way to indicate that the system is 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's 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, that the stored public key hashes to the entry in the freshness table, and 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 the 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.

6 Evaluation

Actually building a system gave us an opportunity to learn some unique lessons about the TCPA technology—lessons that could not be learned until we began implementation. Building a working system not only allowed us to measure its performance, but revealed a number of subtleties about the TCPA technology in general.

6.1 Performance

We benchmarked the Enforcer to get an accurate idea of the performance impact it has versus running a typical Apache installation. We wanted to benchmark a realistic system and workload so we decided to protect Apache's data in the loopback filesystem and calculate how much the Enforcer slows down Apache's ability to serve pages.

In keeping with the theme of a realistic benchmark, we acquired the static Web pages of all of the Athletics departments at Dartmouth as well as the Apache log of all the hits against those pages on a weekday. The

dataset was 19,623 files with a total size of 664 MB. The log file consisted of 20,741 URLs, of which 15% were requests for files that did not exist.

Machine Setup. The machine running the benchmark program was a dual processor Intel Xeon CPU running at 2.00GHz with 512 MB of memory, and running Linux kernel version 2.4.20-ac1 with Debian's "unstable" distribution. The machine running the Enforcer was an IBM Netvista 8310 desktop machine with a Pentium 4 CPU at 2.00GHz, 128M of memory, one IDE hard drive, running Linux kernel 2.6.0-test7 (no preempt), and Debian's "unstable" distribution. Each machine had a 100 megabit full-duplex Ethernet network, plugged directly into the same switch.

Benchmark Setup. When the Enforcer's database was built, 156 of the hashes (out of the 19,623 total) were intentionally modified to be incorrect. This allowed us to see that the Enforcer was actually working because it would log a message every time one of these files was accessed.

Results. More complete data is available in Appendix B. Essentially, the performance for an Apache SSL server with all of the content in an encrypted loopback filesystem, using the TPM to protect the server's private key, and using the Enforcer for integrity checking is quite good. The slowdown is around 6.8% compared to a standard Apache SSL server—i.e., content is not in a loopback, no TPM is involved, and the Enforcer is not used at all.

6.2 Issues

During the course of implementing our system, we discovered a number of issues with the TCPA technology (in addition to the replay and DMA attacks and software update problems mentioned earlier).

Attestation Bootstrapping. In the current specification, the way that attestation is bootstrapped is rather complex. Much to our surprise, the identity-creation process requires authorization of the TPM Owner and requires the TPM endorsement private key to retrieve the certificate for a new identity keypair, but does *not* require the TPM to attest to anything about the software that controls this keypair. It takes additional scouring of the documentation to reveal the path (Section 4.3) by which the identity keypair can testify to a wrapped keypair, which a second round with a YACA can turn into a regular certificate.

The complexity of this process troubles us. In security, one should be careful about trusting something that is too big to fit into one's head. It takes a long time to find the right combination of commands and properties from the specification. What *other* combinations are present? Are there any combinations that enable functionality the designers did not intend? A more concise expression of policy and correctness properties—perhaps verified by formal methods—would greatly enhance the trustworthiness of the TCPA/TCG design.

What is frustrating is that a small change to the TPM might greatly reduce this complexity. For example, if the `TPM_MakeIdentity` command bound the key it creates to the PCR values and included a testament about those values, and `TPM_ActivateIdentity` would not decrypt the certificate unless the identity key was created at the TPM, then the Privacy CA (or any CA, such as our SSL CA) could endorse that the key was generated by a specific TPM *in a specific configuration*, without additional rounds and keypairs.

Monotonic Storage. Multiple processes that interleave at run time cannot rely on the TPM to protect their secrets from each other. We can envision a system where, when a process gets context, its own measurements (i.e., the hash values of the software it is concerned about) are placed in the PCRs, and decisions are based these measurements. When the next process gets context, the PCRs could be cleared, and the new process's measurements are placed in the PCRs. (Indeed, some of the academic hardware alternatives seem to lean in this direction.)

However, the `TPM_Extend` operation ensures that the old PCR value is used in the calculation of the new PCR value, providing a one-way ratcheting mechanism. As this ratchet prevents a reset of any PCR, it makes the above scenario impossible. Using the TPM for context-specific storage does not seem possible—leaving us depending on OS protections.

7 Conclusions and Future Work

We began this project with two goals in mind: to provide a foundation for further trusted computing projects, and to address a gap—open source uses of TCPA.

We have met the first goal to a large extent: we have shown that it appears possible to transform a common Linux desktop machine into a virtual secure coprocessor by using TCPA specifications and hardware.

We have met the second goal of providing an open-source use of the TCPA technology. Further, we have actually built such a system, configured to bind an Apache Web server's SSL private key to the service it provides. We have also illustrated designs and protocols which would achieve our goal of giving clients a higher level of assurance about the SSL server they are connecting to.

The dynamic nature of TCPA, TCG, and industrial trusted computing activity provides a wealth of opportunity for future work in this area. We quickly discuss some avenues:

- **Freshness.** As noted earlier, we're looking at solutions for freshness of secrets stored at run-time, and by user-level code. This also raises the issue of how often changes should be committed—for example, should *each* change to a loopback filesystem update a DIR somehow?
- **Sufficiency of Configuration Checking.** Right now, the Enforcer only protects against modifications of content of the files that the Security Admin deemed critical. As discussed in Section 5, we are working toward implementing the same level of checking as Tripwire. 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.)
- **Protection from Root.** We are currently examining other solutions such as the NSA's SELinux which, when used in conjunction with the Enforcer, may prevent this type of attack.
- **Applications.** Once we complete the Bear platform, we can proceed with the applications we initially had in mind when we started—including looking at heterogeneous systems composed of Bear and Moose together.

All of our code is under the GPL and is available at `enforcer.sourceforge.net`. To date, the code has been downloaded over 500 times. We continue to invite community participation.

Notes

A preliminary version of this paper appeared as a technical report TR2003-471 in August 2003 [20].

Acknowledgments

The authors are grateful to Matthias Bauer, Ryan Cathecart, Dave Challener, and Neal McBurnett for helpful suggestions.

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. Marchesini, Smith and Wild can be reached via addresses of the form `firstname.lastname@dartmouth.edu`.

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] S. Beattie, A. Black, C. Cowan, C. Pu, and L. Yang. CryptoMark: Locking the Stable door ahead of the Trojan Horse, 2000.

[4] D. Bryson. The Linux Crypto API - A User's Perspective. `http://www.kerneli.org/howto/index.php`, May 2002.

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

[6] 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.

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

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

[9] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. *Computer*, pages 55–62, July 2003.

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

[11] E. Felten. Understanding Trusted Computing. *IEEE Security & Privacy*, pages 60–62, May/June 2003.

[12] 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.

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

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

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

[16] D. Ireland. BigDigits multiple-precision arithmetic source code. http://www.di-mgt.com.au/bigdigits.html.

[17] 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.

[18] G. Kim and E. Spafford. The Design and Implementation of Tripwire: a File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29. ACM, ACM Press, 1994.

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, November 2000.

[20] R. Macdonald, S.W. Smith, J. Marchesini, and O. Wild. Bear: An Open-Source Virtual Secure Coprocessor based on TCPA. Computer Science Technical Report TR2003-471, Dartmouth College, August 2003.

[21] P. McGregor and R. Lee. Virtual Secure Co-Processing on General-purpose Processors. Technical Report CE-L2002-003, Princeton University, November 2002.

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

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

[24] D. Safford. Clarifying Misinformation on TCPA. http://www.research.ibm.com/gsal/tcpa/tcpa_rebuttal.pdf, October 2002.

[25] D. Safford. The Need for TCPA. http://www.research.ibm.com/gsal/tcpa/why_tcpa.pdf, October 2002.

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

[27] F. Schneider. Secure Systems Conundrum. *Communications of the ACM*, 45(10):160, October 2002.

[28] S. Schoen. Who Controls Your Computer? Electronic Frontier Foundation Reports on Trusted Computing. http://www.eff.org/Infra/trusted_computing/20031002_eff_pr.php, October 2003.

[29] H. Shimokawa. Remote Gdb With Firewire. http://kerneltrap.org/node/view/145/334, April 2002.

[30] 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.

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

[32] S.W. Smith. Outbound Authentication for Programmable Secure Coprocessors. In *Computer Security—ESORICS 2002*, pages 72–89. Springer-Verlag LNCS 2502, October 2002.

[33] S.W. Smith, E. Palmer, and S. Weingart. Using a High-Performance, Programmable Secure Coprocessor. In *Financial Cryptography*, pages 73–89. Springer-Verlag LNCS 1465, 1998.

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

[35] N. Stam. Inside Intel's Secretive 'LaGrande' Project. http://www.extremetech.com/, September 19, 2003.

[36] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant processing. In *In Proceedings of the 17 Int'l Conference on Supercomputing*, pages 160–171, 2003.

[37] Trusted Computing Platform Alliance. TCPA Design Philosophies and Concepts, Version 1.0. http://www.trustedcomputinggroup.org, January 2001.

[38] Trusted Computing Platform Alliance. TCPA PC Specific Implementation Specification, Version 1.00. `http://www.trustedcomputinggroup.org`, September 2001.

[39] Trusted Computing Platform Alliance. Main Specification, Version 1.1b. `http://www.trustedcomputinggroup.org`, February 2002.

[40] Trusted Computing Group Software Stack Specification announced for development of standards-based applications what will protect data, help secure platforms, September 16, 2003. Press release.

[41] 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.

[42] L. van Doorn, G.Ballintijn, and W. Arbaugh. Signed Executables for Linux. Technical Report UMD CS-TR-4259, University of Maryland, June 2001.

[43] 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.

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

Appendix A: Trust Flow Sketch

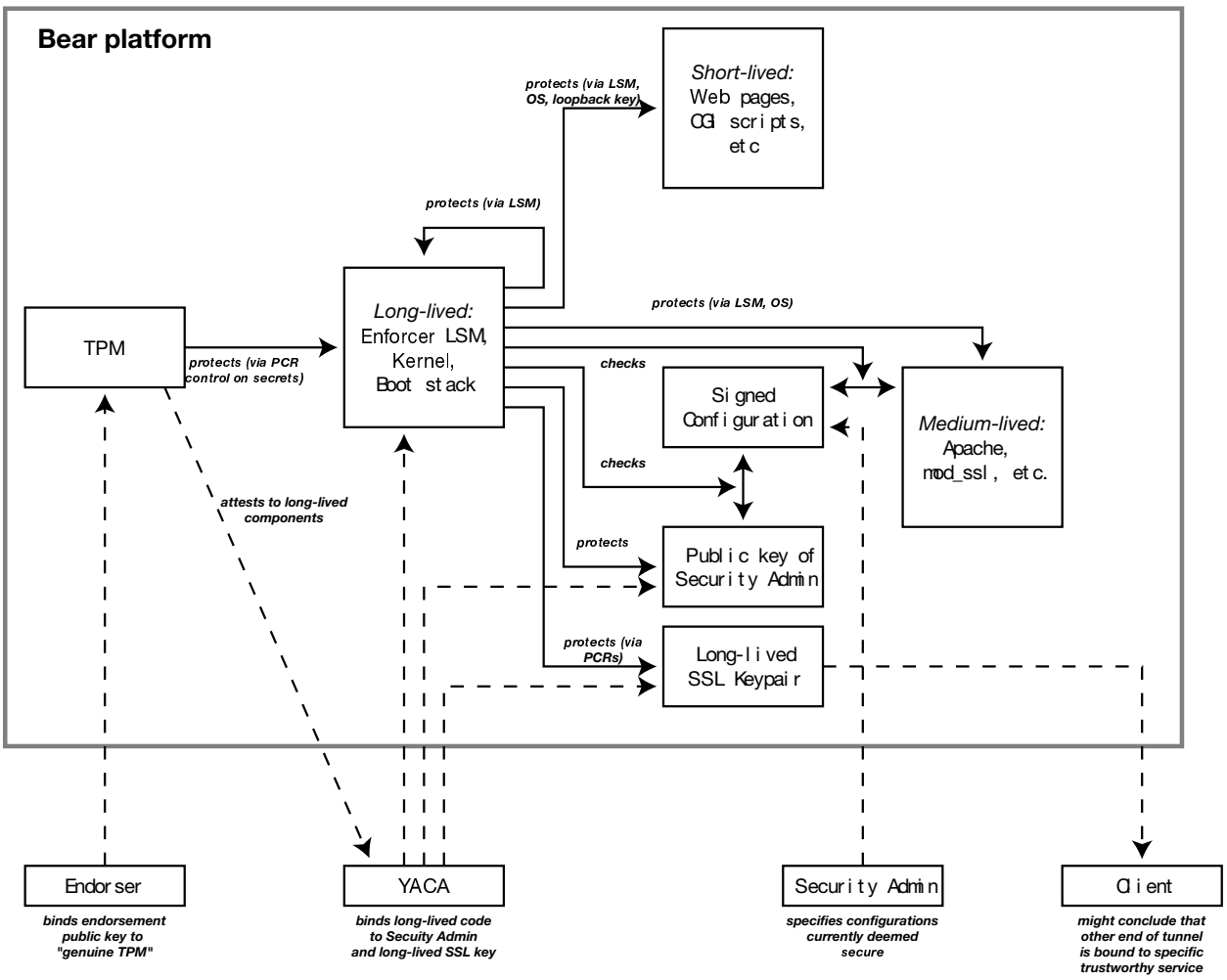

Figure 1: Sketch of the flow of protection and trust in our platform, configured as a Web server. To enable a client to make a trust decision about dynamic content based on a long-lived SSL keypair, we introduce indirection between the core long-lived components and the more dynamic components. Our intention is, like the IBM 4758, the TPM/Linux platform would let the end user buy the hardware, which could authenticate these components to "Yet Another CA."

Appendix B: Performance Results

We ran a total of eight different benchmarks: straight HTTP requests, HTTP with the Enforcer, HTTP with content in the loopback filesystem, and HTTP with the Enforcer and the content in the loopback filesystem. The other four tests were the same, except they were HTTPS requests.

Protocol	No Enforcer No Loopback	Enforcer No Loopback	No Enforcer Loopback	Enforcer Loopback
HTTP	0.0%	1.5%	39.4%	48.5%
HTTPS	0.0%	1.2%	3.7%	6.8%

Table 1: Percentage of slowdown.

For the HTTP benchmarks (which executed quite quickly) each child repeated the URLs three time for a total of 932,940 hits per run. For the HTTPS benchmarks (which executed very slowly) each child only did one run through the URLs for a total of 310,980 hits per run. We ran each test three times and averaged the results. The average size of each Apache request was 22.768 k. Table 1 shows the results.

Protocol		Enforcer	Enforcer/Loopback
HTTP	%CPU	32.18%	29.80%
	%I/O	52.38%	68.84%
	Sum	64.56%	98.65%
HTTPS	%CPU	83.97%	81.12%
	%I/O	14.33%	18.01%
	Sum	98.30%	99.13%

Table 2: Division of the workload.

Table 2 was generated by running `vmstat` on the Enforcer machine while the benchmark was taking place. The percentage of CPU usage ("% CPU") was calculated by adding the 'us' (user) and 'sy' (system) vmstat fields, while the percentage of time spent waiting on I/O ("% I/O") was taken from the 'wa' (waiting I/O) field. From this data we conclude the HTTP benchmarks were mostly I/O bound while the HTTPS benchmarks were CPU bound.

The actual impact of the Enforcer is slight—only 1.5% for an I/O load and 1.3% for a CPU load. As the amount of CPU work goes up relative to the amount of data loaded from disk, the impact of the Enforcer should diminish.

The impact of putting data into the loopback filesystem is more significant, as can be seen in Table 1. For an I/O load the performance impact is 39.4%. For a CPU load the impact was only 3.7%.

The impact of putting the files on the loopback and then protecting them with the Enforcer resulted in the greatest slowdown. For the I/O load the slowdown was 48.5% and for a CPU load the slowdown was 6.8%.