Open-Source Applications of TCPA Hardware

John Marchesini, Sean W. Smith, Omen Wild, Josh Stabiner, Alex Barsamian
Department of Computer Science
Dartmouth College
6211 Hinman, Hanover, NH 03755 USA
{First.Last}@dartmouth.edu

Abstract

How can Alice trust computation occurring at Bob's computer? Since it exists and is becoming ubiquitous, the current-generation TCPA/TCG hardware might enable a solution. When we started investigating this technology, the specification of the TCG software stack was not publicly available, and an implementation is still not; so, we designed and built an open-source platform based on Linux and commercially available TCPA/TCG hardware which would allow us to address the problem of trusting computation. Within the limits of TCPA/TCG hardware security, our solution balances what Alice needs to do to make trust judgments against what Bob needs to do to keep his system running.

Furthermore, we describe how we use our platform to harden three sample open-source applications: Apache SSL Web servers, OpenCA certification authorities, and (with SELinux) compartmented attestation to balance privacy with DRM.

To our knowledge, our project remains the only open-source TCPA/TCG platform in existence, and is also enabling trusted computing applications developed by our user community (`enforcer.sourceforge.net` reports over 1100 sourcecode downloads so far).

1. Introduction

This paper presents a snapshot of our design and experimental work in applying TCPA/TCG hardware to solve computer security problems in the real world[1].

Motivation Modern computing environments present many scenarios where Alice needs to trust that certain properties hold for a program running on Bob's machine, even though Alice may have little reason to trust Bob.

To be effective, a solution to this problem must satisfy several constraints:

- It must be *real*. It cannot be based on hardware that does not exist or is too expensive to be ubiquitous, nor on software that is still vaporware.
- It must be *practical*. It should work with standard protocols, and cannot require a significant departure from the standard software base.

Our Project We began this project by asking ourselves if we could do anything about this problem of trusting remote computation, while living within these constraints.

We started by considering TCPA. Over the last several years, the *Trusted Computing Platform Alliance (TCPA)* (now renamed the *Trusted Computing Group (TCG)*) has released a series of specifications—which, informally, are often referred to by "TCPA," the former name of the group. The TCPA design includes a *Trusted Platform Module (TPM)*—essentially, a smart-card-like chip that is mounted on a PC's motherboard and participates in the boot process—and a *TCG Software Stack (TSS)*, both of which are tied up with Microsoft's *Next-Generation Secure Computing Base (NGSCB)*.

Since the TPM described by the 1.1b specification already ships on many commodity machines, we figured it might be a good place to start. We decided to see what we could do now with the basic 1.1b tool. (TSS implementations and the successor 1.2 TPM are still unavailable.)

This Paper Section 2 presents the basic framework we built on top of Linux and TCPA/TCG hardware. Section 3 presents how we applied this framework to harden SSL Web servers and *certification authorities (CAs)*, and how we combined this framework with SELinux to harden selective software attestation. Section 4 presents our evaluation of how well our solution works. Section 5 surveys prior related work, and Section 6 concludes with some avenues for future work.

1 Portions of this submission are based on our earlier technical reports [17, 20]

2. The Basic Framework

To start with, we need a way for Alice, working within existing hardware, software, and protocols, to reach some conclusion about computation occurring on Bob's computer. The TPM gives us a basic tool (described in Section 2.1). However, this tool binds a secret to a specific full-blown software and data configuration on a given machine, which makes it difficult to deal with two problems:

- In most applications where a relying party Alice needs to authenticate a remote program P on Bob's machine, the overall software and data configuration on a platform often need to change (e.g., for upgrades), even though P remains the same.
- In current distributed security infrastructures, Alice wants to make her trust decision based on whether P proves knowledge of a long-lived private key matching a long-lived X.509 identity certificate, and Bob does not want to have to go back to a CA each time his software or data changes.

We addressed both problems by indirection. The TPM and boot process verifies that our *Enforcer* security module (described in Section 2.2) and supporting software is unmodified; the Enforcer then checks the more dynamic parts of the system against a configuration file signed by a (possibly remote) *Security Administrator*, Cathy. The TPM releases private keys to the Enforcer only when it boots correctly; but the Enforcer only releases the program private key when it satisfies the current configuration (described in Section 2.3). Thus, by delegating configuration judgment to Cathy, a CA can issue a long-lived certificate to Bob's application.

Our earlier technical reports [17, 20] and `enforcer.sourceforge.net` site [19] give more technical details.

2.1. The TPM

We quickly review the basic functionality of the TPM that is currently available. More information on the TCPA/TCG technology can be found in our technical reports [17, 20], the TCPA specifications [37, 38, 39], and other literature [10, 22, 26], as well as `www.trustedcomputinggroup.org`.

The TPM in our commodity hardware has 16 *platform configuration registers (PCRs)*, each 20 bytes long. The TCPA PC specification reserves eight PCRs for specific purposes, leaving eight for applications. 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 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 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).

It is also possible to create keys which are bound to a specific machine configuration with the `TPM_CreateWrapKey` function. 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.

The TPM can 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 a TPM-held private key, the caller can use the `TPM_Quote` command to have the TPM use it to sign a snapshot of the current values of the 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 is wrapped, and to what PCR values.

Certification TCPA provides additional functionality for tasks like proving that a TPM is genuine and *attesting* to the software configuration of a machine. The TCPA specification—and subsequent research [27]—lays out some fairly complex procedures. However, Alice does not want to carry out a complex procedure—she just wants to verify that a remote program knows a private key matching the public key in an X.509 certificate. Upon careful reading of the specification, it appears the TPM can provide equivalent functionality. We provide a new code module that has the TPM create what it terms an "identity key pair" and then obtain an "identity certificate" from what we call YACA ("yet another CA"). This module then uses the TPM to create a wrapped key pair bound to a configuration which includes itself—and then has the TPM use the identity private key to certify that fact. Finally, the module needs to return to a standard X.509 CA (which could be the same YACA) with the identity certificate and the certificate created for this wrapped key pair, in order to obtain a standard X.509 certificate.

Threat Model The TCPA design 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 boot loader 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.

2.2. The Enforcer

Our goal is to bind a private key to program P. How do we permit Bob to carry out appropriate updates to the software that constitutes program P, without rendering this private key unavailable? How do we ensure a malicious Bob cannot roll back a patched program to an earlier version that we now know is unsafe? How do we permit a CA to express something in a certificate that says something meaningful about the trustworthiness of P over future changes—both of software as well as of more dynamic state?

Design In some sense, everything is dynamic, even X.509 key pairs. However, in current PKI paradigms, a certificate binds an entity to a key pair for some relatively long-lived period. But if this entity P is to be a remote program 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

As noted above, 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 signs a description of the medium-lived software which represents a good configuration of the medium-lived software. The Security Admin's signed description acts as a security policy for the medium-lived software. For simplicity, the Security Admin's public key can be part of the long-lived core (although we could have it elsewhere). A Security Admin's security policy could apply to large sets of machines, and in theory, the Security Admin may in fact be part of a different organization. For example, Verisign or CERT might set up a Security Admin who signs descriptions of what are believed to be secure configurations of the program(s) in question, and distributes these descriptions to a number of organizations to use as a a security policy. This approach allows one entity to bless the configurations for multiple sites without having to run all of the servers itself.

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 security policy is correctly signed, and that the medium-lived software matches this policy. The Enforcer then uses the secure storage API to retrieve and update short-lived operational data, when requested by the other software.

Our design binds the protected secrets to the Enforcer and long-lived core instead of the the medium- and short-lived components of the system. This approach alleviates the need to get a new certificate each time the medium- or short-lived components change—presumably quite often.

To prevent replay of old signed policies, 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 policy 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.)

Structure In order to make our system usable, we chose designs that coincide with familiar programming constructs. These choices may also made our system easier to build—since we could re-use existing code.

Short-Lived Data. For short-lived data, we wanted 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 [3].

So, a natural choice for short-lived data was to have the Enforcer save and retrieve keys for an encrypted loopback filesystem. (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 also provided an interface to do that.

Medium-lived Software. For the medium-lived software, we needed 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., [2, 40]).

The Security Admin (again, possibly on a different machine or part of a different organization) prepares a signed security policy of the medium-lived component; the long-lived component of our system uses this policy to verify the integrity of the medium-lived component.

Long-lived Core. Another question was 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.

Security Admin. As noted, our design provides a level of indirection: the Security Admin defines an updatable security policy; the long-lived core ensures that critical secrets are maintained only as long as the rest of the system matches this policy.

Implementation Experience We built our Enforcer as an 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 2000 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. Full details of this implementation are available in our preliminary reports.

The Enforcer uses the `/etc/enforcer/` directory to store its signed policy, public key, etc. (Having the kernel store data in the filesystem is a bit uncouth, but was the best solution 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 policy's signature at load-time; if compiled into the kernel, the Enforcer verifies it when the root filesystem is mounted.

At run-time, the Enforcer hooks all inode permission checks (which happen as a file is opened). The Enforcer calculates a SHA-1 of the file and compares it to the SHA-1 listed in the policy; if the values do not match, it reacts according the option: log the event to the system log, fail the call, or panic the system. Tapping each inode read operation would be better from a security standpoint, in that it would check the file's integrity each time the file is read. While this would alleviate any TOCTOU issues which arise between opening a file and another party writing to it, it would also be quite expensive and would still not work for things like log files.

```
4d37d651b ... 4a57afd4f2 deny file.txt
```

Figure 1. The above example line from the Enforcer's security policy lists a SHA-1 hash over the contents of the file `file.txt`. Should the Enforcer calculate a different hash value for the file, it will fail the request.

Additional Tools. We wrote a number of small executables which make some of the TCPA calls necessary for attestation—`TPM_MakeIdentity` and `TPM_ActivateIdentity`. We also wrote some utilities to produce the security policies, and for each file covered by the policy, the Security Admin can specify what should happen if its integrity check fails: log, deny, or panic (see Figure 1). We used an open-source big integer package [12] to produce a rudimentary key generation (2048-bits), signing tool, and stripped-down verification tool (which was included in the Enforcer kernel module).

2.3. Trust

Linux with the TPM and our Enforcer LSM enables in practice what prior work only enabled in theory: a way to bind a general-function desktop or server program—including its configuration and operational data—to a long-lived private key.

If someone tampers with a file on the server which is guarded by the Security Admin's security policy, the program will not be able to prove knowledge of the private key to the relying party Alice. A CA who wants to certify the "correctness" of such a platform essentially certifies that the long-lived core operates correctly, and that the named Security Admin will have good judgment about future maintenance. (Essentially, this approach generalizes the "epoch" idea of outbound authentication in the IBM 4758 [34].)

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, and the Enforcer (already verified) ensures that the current system matches the Security Admin's signed policy.

The operational data of the program is controlled by various users, per Bob's policy. 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.

Figure 2 at the end of this paper sketches how this trust flows.

3. Applications

Our platform enables us to bind a private key to a program P at Bob's computer, such that:

- Bob can still maintain and upgrade P and its environment, as long as it complies with a security policy signed by the Security Admin Cathy, and
- A relying party Alice can deduce from a long-lived X.509 certificate that, within the physical security limits of the TPM, the wielder of this private key is still P, in this trustworthy configuration.

Furthermore, our platform works in a nicely decentralized way. Bob merely obtains the platform and installs the software; the Enforcer then uses the TPM credentials to obtain a certificate for itself from an appropriate CA.

Once our platform reached some level of stability, we applied it to three problem scenarios.

3.1. SSL Web Servers

First, 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 will not 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 key pair to the identity of Bob's site; server-side cryptographic 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. Our lab prototyped this approach in prior work [14, 33]. However, this previous approach overlooked how to map a long-lived key pair onto short-lived configurations. Must the Web site go back to the CA with each new upgrade of Apache or modification to the Web pages or scripts?

With our platform, we solve this problem as follows. The Security Admin signs descriptions of what she believes are secure configurations of Apache, `mod_ssl`, etc. The TPM checks the Enforcer, which checks that the current server configuration matches the Security Admin's description. The OS (already checked) determines who can change what Web content and when; this content is saved in the loopback file system that the Enforcer protects. 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 violation of the Security Admin's policy, the loopback is immediately 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.

We have implemented this application, and it has been available for download since late 2003.

3.2. Certification Authorities

Secondly, consider the case where Bob's program P is a CA he operates. Many parties may want to have assurance that Bob's CA only uses its private key in accordance with the policies and practices established by this program, running on this operating system. These parties include:

- relying parties who depend on the correctness of the certificates Bob issues, for the correctness of their own applications;
- a bridge CA or higher-level CA that issues a certificate attesting to the trustworthiness of Bob's CA;
- Bob himself, if (as we have seen frequently) his CA is at risk: online, in a shared machine room, and operated by an already overworked staff.

With our platform, we solve this problem as follows. The Security Admin signs descriptions of what she believes are secure configurations of OpenCA [4] and any other related code, such as Apache for online Web-based CAs. The TPM checks the Enforcer, which checks that the server configuration matches this description. We have implemented the CA private key in two ways. In our initial approach, we set it up as an item within the encrypted loopback file system as in the Apache case. In our second approach, we set it up as a TPM-protected RSA credential that never leaves the TPM. We have written an OpenSSL engine that wraps around the TPM-protected credential, which more elegant and extensible.

We have implemented this application; it will be available for download as soon as testing is complete. We are also exploring incorporating threshold cryptography (in the spirit of [44, 45]), perhaps using low-cost programmable cryptographic tokens in addition to the TPM.

Furthermore, if Bob's CA is itself being certified by a higher-level CA, then that higher CA can act as the Security Admin, thereby further easing the maintenance pres-

sure on Bob. We hope to take this approach, and combine our Enforcer/OpenCA code with a "CA-in-a-box" service we will be offering in conjunction with the *Higher Education Bridge CA (HEBCA)* our laboratory will be operating.

3.3. Compartmented Attestation

Finally, consider the case where Bob is a consumer, running a program P whose authenticity and integrity is of concern to a remote stakeholder Alice. The canonical instantiation of this scenario is *digital rights management (DRM)*: Bob purchases licensed content (such as music), and Alice would like ensure that Bob uses this content only in a program that makes illicit use sufficiently difficult for her tastes. Other instantiations include consumer Bob running a banking program that bank Alice wants to verify, or municipality Bob running a online voting program that watchdog Alice would like to verify.

TCPA/TCG The potential of TCPA/TCG for content applications has generated much controversy (e.g, [29]). With the 1.1b TPM, to use the content he has purchased, Bob may have to expose *everything* on his machine to Alice—even programs and data that have little to do with the application in question. Alice might even choose to deny services and rights to Bob, if he has a competitor's product installed.

Unfortunately, TCPA/TCG itself appears insufficient to solve Alice's problems—we are still at the mercy of a corrupt superuser. 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. Even the 1.2 TPM's attempts to localize PCR contexts appear to suffer from this problem.

SELinux To address some of these problems in operating systems, the National Security Agency (NSA) created *Security-Enhanced Linux (SELinux)*, an LSM and a set of programs for Linux, intended to provided mandatory role-based access control, described by a complex and rigorous policy document. This policy assigns roles to subjects and types to objects; after assigning roles and types, the policy document then further describes how subjects in each role can interact with objects of each type. The access controls in SELinux apply not only to filesystem objects but also to memory, network resources, and devices, and are orthogonal to the controls in a standard Unix environment [16]. Therefore, compatibility with existing applications and security structures is maintained. However, since they are mandatory access controls, the system runs under the assumption that unless specific permission is given to a role with regard to a particular data object, that object is completely protected from that role. In this way, sensitive data, whether in a filesystem object or in memory, can be protected from a "root-spy." Even if a program is compromised in a way that it is inadvertently privileged as root (as is common with, say, buffer-overrun attacks), the program is confined to role it was running in and so its ability to compromise other programs is contained.

SELinux can thus provide *software compartments*: confining software programs so that they cannot arbitrarily spy on or modify another program's data except in policy-prescribed, presumably safe ways such as (if we allow) by cut-and-paste or inter-process communication. Traditional operating systems also attempt to do this, but these efforts are frustrated by the fact that there are really only two levels of privilege, user and root, and root is all-powerful. Plus, the access-control checking is often so messily strewn throughout the kernel that one is rarely sure whether it is working at all. SELinux provides far finer granularity, a restricted root, and a central access-control checking module.

Merging SELinux and Enforcer Our design combines these two techniques—our TCPA-based Enforcer with SELinux—to provide *compartmentalized attestation*. The TPM checks SELinux, its policy, and the Enforcer. The SELinux policy keeps applications of interest in their own compartments, and denies other applications and root the ability to interfere with these compartments. The Enforcer module can use a key pair to testify about (and certify a key pair for) the contents of just one compartment; we architected this as a separate key-wielder service that communicates to the compartments. Alice can have assurance that the attestation she receives really pertains to the compartment in question, and that the Enforcer with SELinux will confine her data to just that compartment; Bob can have confidence that nothing outside of that compartment and above Enforcer/SELinux will be communicated to Alice.

Our Experience Currently, we have SELinux running and confining the XMMS music player from root manipulation.

Official NSA documentation of SELinux is scant, extremely theoretical in nature, and offers little in the way of practical "how-to" information to the intermediate or even advanced Unix user as to how to begin. The bulk of papers published about SELinux do not contain much helpful information on configuring the kernel, and papers which describe non-trivial accomplishments in SELinux (such as Gosselin and Schommer's work confining the Apache web server [8]) are so focused on the details of the accomplishment that broad details such as configuration and installation are left out. Even doing trivial tasks such as adding a role or type involve complex manipulations of the many policy files. On one occasion, a mistake in describing a policy somehow resulted in a system that was virtually unusable and needed to be formatted and reloaded. This demonstrates the dangers of working with an overly complex, overly secure system!

The policy language is robust and expressive, but is also cumbersome to learn and use [31]. It is not always clear how to state the security goals of the system and then build a policy which accomplishes those goals. In fact, other researchers (Jaeger et al.) have noted that "the policy implies the security goals of the system—we only learn that certain subjects can only perform certain operations on certain objects" [13].

Status Getting the Enforcer and SELinux to cooperate was a challenge. The first obstacle we faced involved LSM stacking under Linux. The SELinux LSM has the restriction that it must be the primary module in the stack, which would force the Enforcer to be second. This configuration leaves no room for the standard "capabilities" module which is used for default Linux security (e.g., file permissions checking and enforcement). Our first task was to get the functionality of the capabilities module into the Enforcer, as this would free up a slot in the module stack. We then had to rewrite parts of the SELinux LSM so that it does not hook the `inode_permissions` call, thus allowing the Enforcer to handle those calls. (Further analysis here might be necessary.)

4. Evaluation

How well did our project work?

4.1. 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 trying to apply a good 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.

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, 128MB 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.

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.

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.

4.2. Impact

However, another measure of effectiveness is the impact our project has had already. Ours is the first integration of TCPA with a non-trivial application in the open world, and (as noted above) appears to remain the only open-source TCPA platform. The basic framework and Apache application has been available for open-source download since 3Q2003. Statistics from `enforcer.sourceforge.net` report over 1100 sourcecode downloads so far, and email contacts indicate that the project is being used in a number of European projects, as well as generating interest among a number of corporations.

Our CA and SELinux applications are currently being prepared for open-source release.

5. Related Work

Besides presenting a number of novel applications which utilize the TCPA/TCG technology, this paper also extends a number of earlier ideas and previous work.

In some ways, our work extends earlier research into *secure coprocessors* (e.g., [32, 42, 43]). Secure coprocessors have been shown to be feasible as commercial products [5, 35] and can even run Linux and modern build tools [9], and even provided versions of the TCPA/TCG properties of safe computation, sealed storage, and even attestation [34].

Our lab has investigated and developed a number of applications for the IBM 4758 secure coprocessor including hardening Apache [14, 33] (as noted earlier), enhancing privacy [11], hardening S/MIME [23], and exploring new PKI architectures [18, 41]. Many of these projects were repeatedly hampered by the 4758's relatively weak computational power and lack of space. Such specialty devices typically lag behind desktops in terms of functionality and power which, along with their relatively high cost, inhibits widespread adoption, particularly at clients.

These previous struggles led us to investigate emerging TCPA/TCG technology [22, 37, 38, 39], and develop

early versions of our platform [17, 20]. During our initial development, we learned of a number of interesting concurrently-developed projects from IBM research [10, 26] in the same space. IBM has since developed a an alternate Linux-based attestation scheme for TCPA [27]. Some academic efforts [15, 21, 36] have also explored alternative approaches in this "use a small amount of hardware security" space, but no silicon is available for experiments yet.

Many in the field ([1, 30] are notable examples) have criticized TCPA for their potential negative social effects; others (e.g, [7, 24, 25]) have seen positive potential. (Felten [6] and Schneider [28] give more balanced high-level overviews.)

6. Conclusions and Future Work

Modern computing presents many scenarios—such as remote Web applications, or the policies and practices of a CA, or whether a consumer will violate DRM restrictions, or whether voting software operates untampered—where a party Alice needs to trust properties about a computation running on Bob's machine. In this paper, we have reported our work on building a systematic framework to address these problems in a practical way, in the real world—by using currently common TCPA/TCG hardware, and building open-source Linux solutions that lets Alice draw conclusions by verifying ownership of a long-lived private key, and lets Bob maintain and update his system. As we noted earlier, this project yielded the first non-trivial application of TCPA hardware in the open world, and remains (we believe) the only open-source TCPA-based platform.

Many issues lie outside the scope of this submission, such as: a security analysis of overall system, a security analysis of some specific issues with the 1.1b TPM (such as freshness, TOCTOU risks, replay attacks, and vulnerability to root), and our general experiences trying to turn the specification and off-the-shelf hardware into something useful. Our preliminary technical reports contain further thoughts

In ongoing work, we plan to combine the SELinux protection from root-almighty (of Section 3.3), with the TPM-held private keys (of Section 3.2), as part of our general toolkit. In future work, we plan to explore how a CA might use X.509v3 extensions or attribute certificates to communicate the additional hardness properties of an Enforcer-protected private key, as well as to migrate our project to the 1.2 TPM when it becomes commercially available.

Acknowledgments

The authors are extremely grateful to Rich MacDonald for his help in crafting our original TCPA experiments. We are also 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.

References

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

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

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

[4] C. Covell and M. Bell. OpenCA Guides for 0.9.2+. `http://www.openca.org/openca/docs/online`.

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

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

[7] 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.

[8] M. J. Gosselin and J. Schommer. Confining the apache web server with security-enhanced linux. MITRE Technical Report, June 2001.

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

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

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

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

[13] T. Jaeger, R. Sailer, and X. Zhang. Analyzing Integrity Protection in the SELinux Example Policy. In *12th Usenix Security Symposium*, August 2003.

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

[15] 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.

[16] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. NSA/NAI Technical Report, February 2001.

[17] R. Macdonald, S. 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.

[18] J. Marchesini and S. Smith. Virtual Hierarchies - An Architecture for Building and Maintaining Efficient and Resilient Trust Chains. In *NORDSEC2002 - 7th Nordic Workshop on Secure IT Systems*, November 2002.

[19] J. Marchesini, S. Smith, O. Wild, and R. MacDonald. Open source, gpl download site for source. `enforcer.sourceforge.net`.

[20] J. Marchesini, S. Smith, O. Wild, and R. Macdonald. Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Computer Science Technical Report TR2003-476, Dartmouth College, December 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] R. Sailer, X. Zhang, T. Jaeher, and L. van Doorn. Design and Implementation of a TCG-Based Integrity Measurement Architecture. Technical Report RC23064, IBM Research, 2004.

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

[29] S. Schoen. Trusted computing: Promise and risk. `http://www.eff.org/Infra/trusted_computing/20031001_tc.php`, October 2003.

[30] 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.

[31] S. Smalley. Configuring the SELinux Policy. NSA/NAI Technical Report, January 2003.

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

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

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

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

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

[41] G. Vanrenen and S. Smith. Distributing Security-Mediated PKI. In *1st European PKI Workshop Research and Applications*. Springer-Verlag, 2004.

[42] B. 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.

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

[44] S. Yi and R. Kravets. MOCA:Mobile Certificate Authority for Wireless Ad Hoc Networks. In *2nd Annual PKI Research Workshop*, 2002.

[45] L. Zhou, F. B. Schneider, and R. V. Renesse. COCA: A Secure Distributed Online Certification Authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.

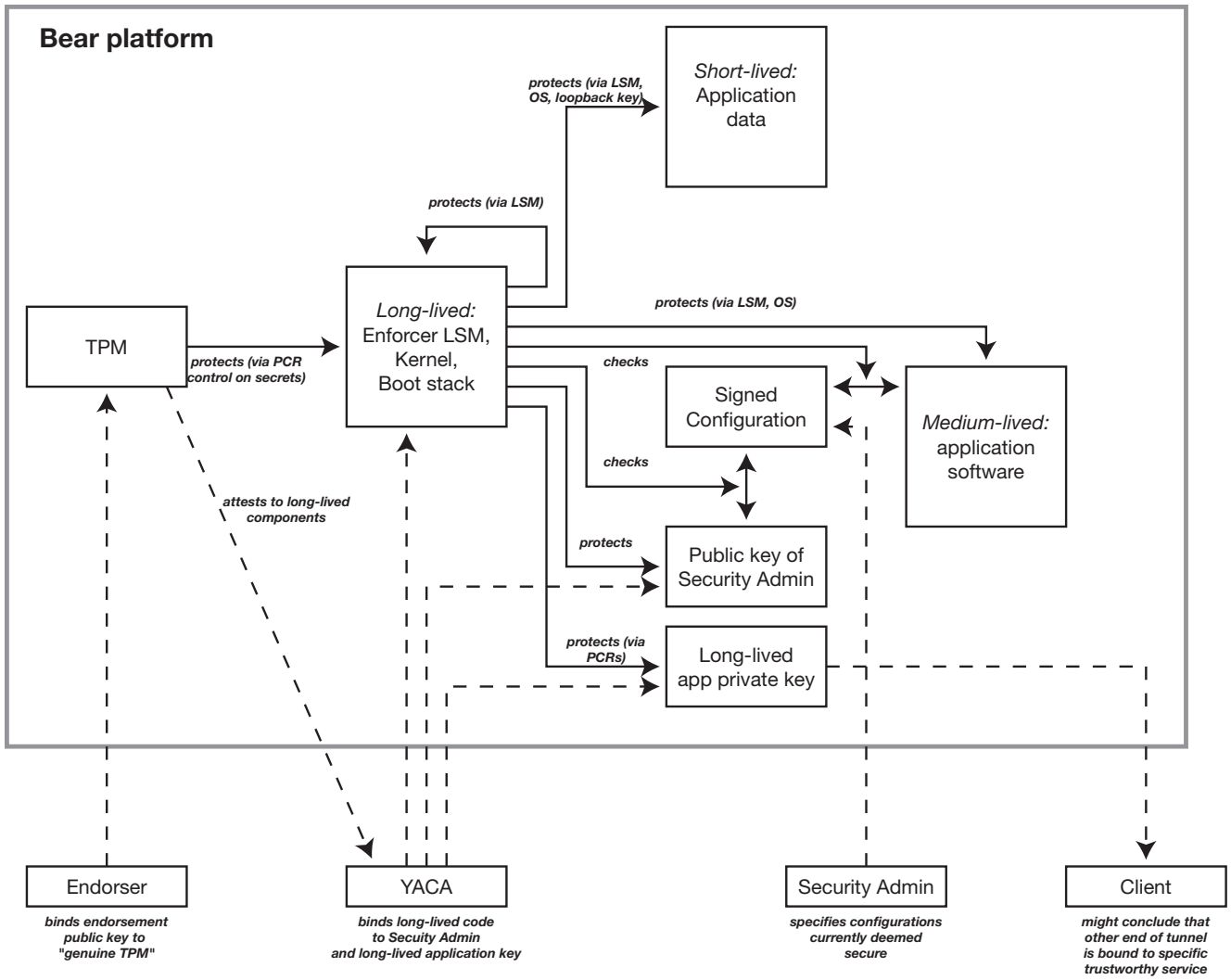

Figure 2. Sketch of the flow of protection and trust in our platform. The solid lines depict relationships between the platform components, and the dashed lines represent trust relationships between platform components and off-platform entities such as humans and CAs. To enable a client to make a trust decision about dynamic content based on a long-lived application key pair, 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."