Outbound Authentication for Programmable Secure Coprocessors

S.W. Smith*
Department of Computer Science/Institute for Security and Technology Studies
Dartmouth College
Hanover, NH USA 03755
sws@cs.dartmouth.edu

Technical Report TR2001-401

March 2001

Abstract

A programmable secure coprocessor platform can help solve many security problems in distributed computing. These solutions usually require that coprocessor applications be able to participate as full-fledged parties in distributed cryptographic protocols. Thus, to fully enable these solutions, a generic platform must not only provide programmability, maintenance, and configuration in the hostile field—it must also provide *outbound authentication* for the entities that result. A particular application on a particular untampered device must be able to prove who it is to a party on the other side of the Internet.

To be effective, a secure outbound authentication service must closely mesh with the overall security architecture. Our initial architecture only sketched a rough design for this service, and did not complete it. This paper presents our research and development experience in refining and implementing this design, to provide PKI-based outbound authentication for the IBM 4758 Model 2 secure coprocessor platform.

1 Introduction

How does one secure computation that takes place remotely—particularly when someone with direct access to that remote machine may benefit from compromising that computation? This issue lies at the heart of many current e-commerce, rights management, and PKI issues.

To address this problem, research (e.g., [9, 13, 14]) has long explored the potential of *secure coprocessors*: high-assurance hardware devices that can be trusted to carry out computation unmolested by an adversary with direct physical access. For example:

- An adversary can subvert *rights management* on a complex dataset by receiving the dataset and then not following the policy; secure coprocessors enable solutions by receiving the dataset encapsulated with the policy, and only revealing data items in accordance with the policy.
- An adversary can subvert *decentralized e-cash* simply by increasing a register. However, secure coprocessors enable solutions: the register lives insider a trusted box, which modifies the value only as part of a transaction with another trusted box.

Many other applications—including auctions [6], e-commerce co-servers [10], and mobile agents [15]—can also benefit from the high-assurance neutral environment that secure coprocessors provide.

One necessary step for achieving this potential is building high-assurance, programmable secure coprocessor platforms. Such work usually focuses on establishing and maintaining physical security, and on how the device can

*This paper reports design and development work performed when the author was on the research staff of IBM Watson.

authenticate code-loads and other commands that come from the outside world. However, using secure coprocessors to secure distributed computation *also* requires *outbound authentication (OA)*: the ability of coprocessor applications be able to authenticate themselves to remote parties. (Code-downloading loses much of its effect if one cannot easily authenticate the entity that results!)

Merely configuring the coprocessor platform as the appropriate entity—a rights box, a wallet, an auction marketplace—does not suffice. A signed statement *about* the configuration also does not suffice. For maximal effectiveness, the platform should enable the *entity itself* to have authenticated keypairs and engage in protocols with any party on the Internet: so that only that particular trusted auction marketplace, following the trusted rules, is able to receive the encrypted strategy from a remote client; so that only that particular trusted rights box, following the trusted rules, is able to receive the object and the rights policy it should enforce.

The Research Project. The problem of programming a device is deceptively simple until one thinks of the complexities of shipping, upgrades, maintenance, and hostile code, for a generic secure coprocessor that can be configured and maintained in the hostile field. [7] Other reports [4, 3, 8] present our experiences in bringing such a device into existence as a COTS product, the IBM 4758.

The same issues that complicate configuring the box also complicate outbound authentication. Although our initial security architecture [8] sketched a design for OA, we did not fully implement it—nor fully grasp the nature of the problem—until the Model 2 device. As with the rest of the 4758 work, we had to simultaneously undertake tasks one might prefer to tackle sequentially: identify fundamental problems; reason about solutions; design, code and test; and ensure that it satisfied legacy application concerns.

This Paper. This paper is a post-facto report expanding on this research and development experience. Section 2 discusses problems; Section 3 presents theoretical foundations; Section 4 and Section 5 present design and implementation experiences; and Section 6 suggests some directions for future work.

2 Evolution of the Problem

2.1 Background and Approach

Software Structure We start with a brief overview of the software structure of the 4758. The device is *tamper-responding*: with high assurance, on-board circuits detect tamper attempts and destroy the contents of volatile RAM and non-volatile BBRAM before an adversary can see them. The device also is a general purpose computing device; internal software is divided into *layers*, with layer boundaries corresponding to divisions in function, storage region, and external control. The current family of devices has four layers: Layer 0 in ROM, and Layer 1 through Layer 3 in rewritable FLASH.

The layer sequence also corresponds to the sequence of execution phases after device boot: initially Layer 0 runs, then passes invokes Layer 1, and so on. In the current family Layer 2 is intended to be an internal operating system; it invokes the application in Layer 3 but doesn't actually go away.

We intended the device to be a generic platform for secure coprocessor applications; hence, applications must be installable. Business forces pressured us to have only one shippable version of the device, and to ensure that an untampered device with no hardware damage can always be revived. We converged on a design where Layer 1 contains the security configuration software which establishes owners and public keys for the higher layers, and validates code installation and update commands for those layers from those owners. Layer 1 is updatable, in case we want to change its officer's public key, upgrade algorithms, or fix bugs; but is mirrored so that failures during update will not leave us with a non-functioning layer.

Authentication Approach Another business constraint we had was that the only guaranteed contact we would have with a card was at manufacture time—in particular, we could assume no audits, nor database of card-specific data (secret or otherwise), nor any provide any services to cards once they left. This constraint naturally suggested the use

of *public-key cryptography* for authentication, both inbound and outbound. For OA, we should keep a private key in tamper-protected memory, and have something create signed certificates about the corresponding public key. Because of the last-touch-at-manufacturing constraint, we can last do something cryptographic at the factory. After that, it's up to the card to prove what it is—despite changing configurations, and potentially buggy or malicious software, developers, and users.

2.2 On-Card Entities

One of the first things we need to deal with is the notion of what an on-card entity *is*.

This is tricky. Let's start with a simple case: suppose the coprocessor had exactly one place to hold software and zeroized all state with each code-load. In this scenario, the notion of entity is pretty clear: a particular code-load C_1 executing inside an untampered device D_1. The same code C_1 inside another device D_2 would constitute a different entity; as would a re-installation of C_1 inside D_1.

However, even this simple case raises a challenge: if a reload replaces C_1 with C_2, and reloads clear all state, how does the resulting entity—C_2 on D_1—authenticate itself to a party on the other side of the net? (We would start down a path of shared secrets and personalized code-loads.)

Unfortunately, the reality of building a programmable secure coprocessor to support real customer and research needs creates more complex scenarios: multiple layers from different owners, a reloadable code-loading layer, a general-purpose OS, post-boot execution phases, and developer demands that we sometimes permit secret retention across reload. Each of these adds more wrinkles. With a secret-preserving load; the "entity" may stay the same, but the code may change; with a secret-destroying load, the "entity" may change, but the code stays the same. Should an application entity "include" the OS underneath it? Should it include the configuration control layers that ran earlier in this boot sequence, but are no longer around?

Proposed future areas of work blur these boundaries even further. In an architecture that supports two or more mutually suspicious applications, should the presence (and version) of one application be part of the other entity? What about an architecture that permits applications to graft extensions onto the operating system, or one with hardware support that permits more general scenarios than successively less-trust phases?

Since we built the 4758 to support real applications, we gravitate towards a practical definition: an entity is an installation of the application software in a trusted place, identified by the underlying operating system and hardware.

2.3 Tricky Scenarios

Consider a roughly-drawn scheme: on-card entities use a certified keypair, whose private key lives in tamper-protected memory, to prove who they are to some party P in the external world. Careful reflection generates numerous scenarios that create problems

Code-Loads. Suppose entity C is the application Layer 3 in a particular device. Layer 3 may change: two possible changes include a simple code update taking the current code C_1 to C_2, or a complete re-install of a different application from a different owner, taking C_1 to C_3.

With no additional countermeasures, we're left with some disturbing facts:

- An external party P cannot distinguish between C_1 and C_2.

  If C_1 was corrupt, then party P can never be sure if they're talking to a patched version—even if some cards may never had had the bad version in there.

  If C_1 is correct but C_2 is corrupt, then party P can never be sure if they're talking to a good version, or one that had the corrupt update. The mere existence of a signed update command compromises all the cards.

- An external party P cannot distinguish between C_1 and C_3. If C_3 uses the same private key as the old application C_1, then a malicious developer can write code that pretends to be someone else's.

Code-Loading Code. Even more serious problems arise if a corrupted version of the configuration software A in Layer 1 exists. If an evil version existed that allowed arbitrary behavior, then (without further countermeasures) a party P cannot distinguish between *any* on-card entity E_1, and an E_2 consisting of a rogue Layer 1 carrying out some elaborate spoof.

Underlying Code. Problems can also arise because *underlying* code changes. Debugging an application requires an operating system with debug hooks; but since nothing works like the real thing, a reasonable development scenario is to be able to "update" back-and-forth between a version of the OS with debug hooks and a version without.

With no additional countermeasures, party P cannot distinguish between:

- the application running securely with the real OS;
- the application with debug hooks underneath it;
- the application with the real OS, but with a policy that permits hot-update to the debug version.

Internal Certification. The above scenarios suggest that perhaps a single keypair for the card may not suffice. But if we extend to schemes where one on-card entity generates and certifies keypairs for other on-card entities, we encounter more tricky scenarios. For example, suppose Layer 1 generates and certifies keypairs for the Layer 2 entity.

If a reload replaces corrupt OS B_1 with an honest B_2, then party P should be able to distinguish between the certified keypair for B_2 and that for B_1. However, without further countermeasures, if supervisor-level code can see all data on the card, then B_1 can forge messages from B_2.

A similar penetrated-barrier issue arises if we expect an OS in Layer 2 to maintain a private key separate from an application Layer 3, or if we entertained alternative schemes where mutually suspicious applications executed concurrently. If a hostile application might in theory penetrate the OS protections, then an external party cannot distinguish between messages from the OS, messages from the honest application, and messages from rogue applications.

Outliving the Certified. This line of thinking led us to the more general observation that, if the *certifier* outlives the *certified*, then the integrity of what the certified does with their keypair depends on the *future* behavior of the certifier.

In the case of the coprocessor, this observation has some more subtle and dangerous implications. For example, one of the reasons we centralized configuration control in Layer 1 was to enable the application developer to distrust the OS developer and request that his application (and its secrets) be destroyed, if the underlying OS undergoes an update. However, if the outbound authentication scheme has the underlying OS certify keypairs for the application, and the OS keypair lives through the update, then (without further countermeasures) an external party cannot distinguish between messages from the original application and messages (and certified keypairs) forged by the untrusted post-update OS.

3 Theory

The construction of the card suggests that we use certified keypairs for outbound authentication. The obvious approach of just sending the card out with a certified keypair does not work. The next obvious approach of having some on-card entities certify things of other on-card entities also does not quite work. Why?

3.1 Dependence

Our problems arose because entities may develope dependencies as computation proceeds. We need language to talk about this.

Definition 1 *(History, Run, $\prec$)* Let a *history* be a finite sequence of computation for a particular device. Let a *run* be some unbounded sequence of computation for a particular device. We write $H \prec R$ when history H is a prefix of run R.

Definition 2 *(Dependency)* For entities E_1 and E_2 in run R, we write:

- $E_2 \xrightarrow{data}_R E_1$ when E_1 has read/write access to the secrets of E_2
- $E_2 \xrightarrow{code}_R E_1$ when E_1 has write-privilege to the code of E_1

Let $\longrightarrow_R$ be the transitive closure of the union of these two relations.

Definition 3 *(Dependency Set)* For an entity E in a run R, define its *dependency set*:

$$\textit{DepSet}(E, R) \quad = \quad \{F : E \longrightarrow_R F\}$$

A party trying to authenticate entity E usually has some notion who E is. *DepSet*(E, R) denotes the rest of the entities whose malicious operation could fool P in this run R. If C_1 follows B_1 in the post-boot sequence, then $C_1 \xrightarrow{data}_R B_1$; if C_2 is a secret-preserving replacement of C_1, then $C_1 \xrightarrow{data}_R C_1$; if A can reburn the FLASH segment where B lives, then $B \xrightarrow{code}_R A$.

3.2 Certificate Chains

How might an external party P authenticate a message M came from on-card entity E? Typically, E signs M using the private member of a keypair *KP*, and party P obtains a *certificate chain* X that purports to support the binding of *KP* to E. P then carries out some validation algorithm *Validate* on the chain. If that yields *true*—and if the signature on M validates against *KP*—then P accepts that M came from E.

In this analysis, we'll use *certificate* in the limited sense of a signed statement binding together a public key and an identity. The certificate chain usually consists of (as the name suggests) a chain of certificates: $X_0, X_1, ..., X_n$. For $i < n$, each X_i is a signed statement attesting to the public key of the entity that signed X_{i+1}; X_n attests to the public key of the entity E. The *Validate* algorithm usually (e.g., [1]) consists of: validating that X_0 was signed by a *trust root* (an entity and keypair that party P trusts prima facie), and then validating that each X_i was signed by the public key named in X_{i-1}. (One may also see forays into time and expiration dates; PGP allows individual users to establish more flexible trust policies.)

3.3 Drawing Conclusions from Certificate Chains

What consistency rules are reasonable for validating chains from coprocessor entities?

First, we need some notion of trust. A party P usually has some ideas of which on-card applications it might trust to behave "correctly" regarding keys and signed statements, and which ones it is unsure of.

Definition 4 For a party P, let *TrustSet*(P) denote the set of entities that P whose statements P trusts.

In a more formal treatment, $E \in$ *TrustSet*(P) might imply:

- If P knows a binding of *KP* to E, and receives a statement S correctly signed with *KP*, then P concludes that E said S.
- If S is a statement regarding some domain of interest (such as statements about other keypair-entity bindings), and P believes that E said S, then P believes S.

Security dependence depends on the run; entity and trust do not. This leads to a potential conundrum: suppose in run R, $C \longrightarrow_R B$, $C \in \mathit{TrustSet}(P)$, but $B \notin \mathit{TrustSet}(P)$. Then P cannot reasonably accept any signed statement from C, because B may have forged it.

This situation suggests the following rule:

Definition 5 *(Consistency)* A certification scheme is *consistent* when, for any entity E, party P, run R, and chain X_E allegedly from E:

$$\mathit{Validate}(P, X_E) \implies \mathit{DepSet}(E, R) \subseteq \mathit{TrustSet}(P)$$

One might also turn the implication the other way.

Definition 6 *(Complete)* A certification scheme is *complete* when, for any entity E, party P, run R, and correctly signed chain X_E actually from E:

$$\mathit{DepSet}(E, R) \subseteq \mathit{TrustSet}(P) \implies \mathit{Validate}(P, X_E)$$

In the context of OA for coprocessors that cannot be opened or otherwise examined, it's reasonable to impose the restriction: on-card entities carry their own chains, and an external party decides validity based on this chain and the party's own list of trusted entities.

Definition 7 Let $\mathit{Chain}(E, H)$ denote the certificate chain presented by entity E after history H.

Definition 8 *(Trust-set)* A *trust-set* certification scheme is one where the *Validate* algorithm is deterministic on the variables $\mathit{Chain}(E, H)$ and $\mathit{TrustSet}(P)$.

These definitions equip us to formalize a fundamental observation:

Theorem 1 Suppose an on-card entity E uses a trust-set certification scheme that is consistent and complete, and suppose two histories H_1, H_2 are prefixes of runs R_1, R_2 respectively. Then:

$$\mathit{DepSet}(E, R_1) \neq \mathit{DepSet}(E, R_2) \implies \mathit{Chain}(E, H_1) \neq \mathit{Chain}(E, H_2)$$

Proof Suppose $\mathit{DepSet}(E, R_1) \neq \mathit{DepSet}(E, R_2)$ but $\mathit{Chain}(E, H_1) = \mathit{Chain}(E, H_2)$. Let party P have $\mathit{DepSet}(E, R_1) \subseteq \mathit{TrustSet}(P)$ but $\mathit{DepSet}(E, R_2) \not\subseteq \mathit{TrustSet}(P)$.

Since this is a trust-set certification scheme and $\mathit{Chain}(E, H_1) = \mathit{Chain}(E, H_2)$, party P must either accept or reject the chain in both runs. If party P accepts in run R_2, then the scheme cannot be consistent; But if party P rejects in run R_1, the scheme cannot be complete. □

Theorem 1 implies that if:

- on-card entities are going to carry around their own certificates,
- the nature of coprocessor architecture forces dependencies,
- and we need to accommodate a range of potential opinions about trusted entities,

then certificate chains had better name the dependency set.

(We note that the above definitions and theorem could also be extended to include the notion of *time* as a parameter to *Validate*.)

4 Design

4.1 Breaking Unwanted Dependencies

Our first step developing an OA scheme that enables reasonable conclusions is to trim away unnecessary dependencies.

Vertical Separation Let B, C be Layer i, Layer $i+1$ respectively. The post-boot sequence gives us $C \longrightarrow_R B$, which we felt was unavoidable. [1] But other direction should be avoidable; and we used hardware to avoid it.

- **Ratchet Locks.** To provide high-assurance separation, we developed ratchet locks: an independent microcontroller tracks a counter, reset to zero at boot time. The microcontroller will advance the ratchet at the main coprocessor CPU's request, but never roll it back. Before B invokes the next layer, it requests an advance.
- **Protected Memory.** To ensure $B \overset{data}{\not\longrightarrow}_R C$, we reserved a portion of battery-backed RAM for B—and used the ratchet hardware to enforce access control.
- **Protected Code.** To ensure $B \overset{code}{\not\longrightarrow}_R C$, we write-protect the FLASH region where B is stored. The ratchet hardware restricts write privileges only to the designated prefix of this sequence.

Horizontal Separation Another set of solutions addressed code-loading and ownership changes.

- **Updates to the code-loading layer.** As discussed elsewhere, we felt that centralizing code-loading and policy decisions in one place enabled cleaner solutions to the trust issues arising when different parties control different layers of code. Suppose Layer 1 entity A_1 is reloaded with A_2. It's unavoidable that $A_2 \overset{code}{\longrightarrow}_R A_1$. But to avoid $A_1 \overset{data}{\longrightarrow}_R A_2$, we take these steps as an atomic part of the reload:
  - A_1 generates a keypair for its successor A_2;
  - A_1 uses its current keypair to sign a *transition certificate* attesting to this change of versions and keypairs;
  - A_1 destroys its current private key.
- **Updates to higher layers.** For higher layers, we established a two-step process: establishing ownership for an unowned layer, and loading code into that layer. At every boot, Layer n's private memory is cleared unless it's owned and contains undamaged code. Code reloads will preserve a layer's memory only if that layer's owner explicitly permits that—and the card is in a position to verify that permission. This breaks dependency across successive installations of code at that layer, as well as across all other loads the owner requested.

Diagonal Separation. The card may destroy a layer's secrets because a code-load violated that layer's owner's policy, or because of other disaster. How should such destruction relate across layers? We decided to preserve a lower layer's secrets, but destroy a higher layer's.

4.2 Chaining along the Dependency

4.2.1 Linear Chains

Trimming dependency down to a linear chain enables a nice certification scheme. Suppose E_0 is a CA and $E_1, ..., E_k$ is a sequence of entities in history $H \prec R$ such that:

- For $i \geq 1$, $\mathit{DepSet}(E_i, R) = \{E_j \ : \ 0 \leq j < i\}$

[1] With unknown software, and only one chance to get the hardware right, we did not feel comfortable with attempts to restore the system to a more trusted state, short of reboot.

- *Chain*(E_1, H) is a certificate from E_0.
- For $i > 1$, *Chain*(E_i, H) is a certificate from E_{i-1} appended to *Chain*(E_{i-1}, H).

This linear structure gives rise to a simple algorithm for validation:

- *Validate*$(P,$ *Chain*$(E_1, H))$ is true when $\{E_0, E_1\} \subseteq$ *TrustSet*(P) and the certificate is correctly signed.
- For $i > 1$, *Validate*$(P,$ *Chain*$(E_i, H))$ is true when *Validate*$(P,$ *Chain*$(E_{i-1}, H))$ is true, $E_i \in$ *TrustSet*(P), and the the E_i certificate is correctly signed.

This is a trust-set certification scheme, since party P decides *Validate* based only the chain and *TrustSet*(P); this scheme is also consistent and complete, since party P accepts the certificate if and only if *DepSet*$(E_k, R) \subseteq$ *TrustSet*(P).

Layer 1 If we group the ROM Layer 0 with the hardware, and describe the Layer 1 entity as "this software, on this hardware," then the above separation schemes organize the the Layer 1 dependency into a linear chain, pointing backwards in time: each version depends on the previous version.

This convenient organization enables clean construction of a consistent, complete, trust-set certification scheme for Layer 1 entities: the certificate chain consists of the original factory certificate, and the sequence of transition certificates.

Subsequent Layers, Conveniently. If officers requested that all reloads destroyed secrets, then the linear dependency sequence would extend upwards to Layer 2 and Layer 3 (much as Figure 16 from [8] implies).

For example, suppose A_1, B_1, C_1 are the layer entities. Then *DepSet*(B_1, R) is just *DepSet*(A_1, R) with A_1; and *DepSet*(C_1, R) picks up B_1. The dependencies form a linear sequence; the vertical, horizontal, and diagonal separations of Section 4.1 preserve this fact across reloads.

4.2.2 Subsequent Layers, in Reality

Unfortunately, this simple design cannot accommodate the reality of the development and application environments we targeted.

- **Selective Preservation.** Some developers wanted their layer's data preserved across updates Some don't. Some want the their data preserved, but still regard the installation with the new code as a different entity.
- **Retroactive Paranoia.** Even full-preservation developers reserved the right to, post-facto, suddenly develop negative opinions about certain versions of code—and be able to verify whether any of those nefarious entities had snuck in and out.
- **Careless Verification.** One experienced application architect insisted that users and developers will not pay attention to details of certificate paths, and (unless care was taken) would happily accept anything that validates to any root, for any purpose.
- **Penetration Risk.** Although the current architecture forces applications to trust the device's OS, we felt that developers would also demand "retroactive paranoia" here: to, later on, discover that a certain version of the OS code could be penetrated by a malicious application, and to (post-facto) try to determine if that had happened.
- **Upstart Developers.** Where possible, we should maximize the credibility our architecture can endow on applications from small developers unable to assure the public of the integrity and correctness of their applications (e.g., through code inspection, formal modeling, etc).

4.3 Dealing with Reality

Can we preserve the nice chain-scheme, while accommodating these limitations?

4.3.1 Lifetimes and Dependency

The conflicting concepts that developers have about what exactly happens to their on-card entity when code update occurs leads us to think more closely about entity lifetimes.

> **Definition 9** *(Configuration, Epoch)* A Layer N *configuration* is the maximal period in which that Layer is runnable, with an unchanging software environment. A Layer N *epoch* is the maximal period in which the Layer can run and accumulate state. If E is an on-card entity in Layer N,
>
> - E is an *epoch-entity* if its lifetime extends for a Layer n epoch.
> - E is a *configuration-entity* if its lifetime extends for a Layer n configuration.

A Layer n epoch-entity consists of a sequence of Layer n configuration-entities. This sequence may be unbounded—since any particular epoch might persist indefinitely, across arbitrarily many configuration changes.

> **Definition 10** *(Sequence Set)* Suppose E is a Layer n epoch-entity.
>
> - Let *Sequence*(E, H) be the sequence of Layer n configuration-entities in history H.
> - Let *Sequence*(E, R) be the sequence of Layer n configuration-entities in the entire run R.

If $H \prec R$, then *Sequence*$(E, H) \subseteq$ *Sequence*(E, R). However, this inequality may very well be strict: since H is just a prefix, many configuration changes may still come.

Given developer constraints, we have two views of Layer n in a particular run R: as a configuration-entity E_C and as an epoch-entity E_E, satisfying $E_C \in$ *Sequence*(E_E, R).

- $E_E \xrightarrow{data}_R E_C$, because entity E_C *is* the epoch-entity E_E during that configuration.
- However, we can ensure that $E_C \not\xrightarrow{data}_R E_E$ if we take care to declare a portion of the Layer n protected memory for the configuration-entity only—and if we ensure that this memory is indeed destroyed as part of configuration change.

4.3.2 Certification Scheme

Since we do not know a priori what these applications will be doing, we felt that application keypairs needed to be created and used at the application's discretion. From the software architecture, this means that Layer 2 does this work (since it's easier at run-time and the Layer 1 protected memory is locked away before Layer 2 and Layer 3 run)

However, developers want to think of applications both as epoch-entities and as configuration-entities. Can we build a Layer 2 CA that will enable the nice chain-scheme of Section 4.2.1 for both types of applications?

Clearly Bad Example. If this CA B_{CA} outlived the Layer 2 configuration, then our certification scheme *cannot be both consistent and complete.* Suppose C_1 is a configuration-entity on top of B_1; B_1 changes to B_2 but both are in B_{CA}; and party P trusts C_1, B_1 but not B_2. (Such parties will likely exist, for developers who worry about configurations.) For the scheme to be complete, P should accept certificate chains from C_1—but that means accepting a chain from B_{CA}, and $B_{CA} \longrightarrow_R B_2 \notin$ *TrustSet*(P).

Subtly Bad Example. The above example fails because the application entity has a CA whose dependency set is larger than the application's. Limiting the CA to the the current Layer 2 configuration-entity eliminates this issue, but fails to address the penetration risk issue from Section 4.2.2. Parties who come to believe that a particular OS can be penetrated by an application can end up with the current B_{CA} depending on future application loads.

Our Solution. Our final design avoids these problems by having the Layer 2 CA B_{CA} live *exactly* as long as the Layer 3 configuration. We reserve part of the Layer 2 protected memory for the private key for B_{CA}; upon any change to the Layer 3 configuration, Layer 1 destroys the old B_{CA} private key, generates a new keypair, and certifies it to belong to the new B_{CA} for the new Layer 3 configuration.

This gives us a trust-set scheme for both epoch-applications and configuration-applications, that is both complete and consistent.

5 Implementation

5.1 System Model

With this certification scheme, the Layer 3 application sees an outbound authentication *manager* (resident in Layer 2) that maintains a store of *certificate-keypair objects (CKOs)*, and provides various services for the application to interact with them.

These services include:

- *generating and certifying* a new CKO (the certificate indicates whether the application specified the private key should persist to end of the current configuration, or all the way to the end of the epoch);
- *operations with the private-key* of a specified application-level CKO; and
- providing *certificate lists and certificates* so the application can easily extract those necessary to form a supporting chain for a particular signature. (To ease chain construction, we provide the list as an indexed array, whose entries include the index of that certificate's signing certificate.)

The store includes all the Layer 1 certificates, including the original device certificate, as well as the public key of the factory CA (so the application can authenticate other device applications).

Automatic Destruction. The layers underneath the application ensure that the store changes in accordance with significant state changes of the device.

When the Layer 3 configuration ends, the private keys in any current Layer 3 configuration keypairs need to be destroyed along with the OA Manager's current private key; and a new OA Manager keypair needs to be generated and certified. When the Layer 3 *epoch* finally ends: the private keys in any current Layer 3 *epoch* keypairs need to be destroyed; we also take this opportunity to delete any old inactive Layer 2 and Layer 3 certificates that persist.

The Manager itself needs to gracefully recognize and respond to these changes (e.g., by noticing that it has a new keypair object, and that many other objects have changed to *inactive*, "no private key" status).

5.2 Coding

Our implementation consists of two components:

- the security configuration code in Layer 1 needs to do the appropriate generation, certification, and destruction as an atomic part of the various configuration changes, and
- the OS that IBM ships in Layer 2 needs to pick up the pieces that Layer 1 leaves it, and provide the appropriate services to Layer 3.

Full support for OA shipped with all Model 2 family devices. Furthermore, the Layer 1 component underwent full formal modeling and testing, as part of the FIPS 140-1 Level 4 validation of the Model 2 hardware and security layers.

The Threat of Backwards Compatibility. As our original architecture paper sketched, we intended full outbound authentication support in the device from the beginning. However, the Model 1 family of 4758 devices shipped with just the Layer 1 chain, for the mundane reason that we needed to meet the product deadline imposed by the legacy IBM cryptographic accelerator line, which (at the time) did not require this feature.

The potential that we might upgrade deployed Model 1 cards to include OA complicated our coding. We needed to live within the hardware restrictions of both devices: the smaller sizes and layouts of protected BBRAM in Model 1 limited what we could store; the larger FLASH sector sizes in Model 2 eliminated any chance of establishing special certificate stores. Furthermore, this potential meant that we needed to allow for an OA-aware Layer 1 to be loaded into a pre-OA card, and figure out and process the pre-OA certificate information there. The eventual decree that OA would only go out with the new devices was welcome news.

Storage Areas. We had four interesting transitions: epoch and configuration ends, for Layer 2 and Layer 3. Since the Model 1 device did not use the Layer 2 and Layer 3 BBRAM regions, we seized them, divided each in two to get four regions, declared one for each of the interesting transitions, and ensured that Layer 1's initial clean-up code enforced the appropriate rules.

However, what needs to be destroyed—a non-empty set of private keys—and what needs to be stored is much larger than these small protected regions. New material may be generated for code-loads or other changes to any of the three rewritable layers, and may also be generated during application run-time. We extended the virtual storage area by storing a MAC key and a TDES encryption key in each protected region. We stored the ciphertext for new material wherever we could: during a code-change, that region's FLASH segment; during application run-time, in the Layer 2-provided PPD data storage service. The OA Manager start-up code needs to check all the FLASH segments to see where the latest ciphertext store is, extract the run-time certificates from PPD, and digest all this information. This digestion includes recognizing that a previously active keypair has been demoted to inactive, by noticing that the entity TDES/MAC keys no longer extract the private key.

Atomic Actions. Now that we have places to store things and destroy them, we needed to extend the Layer 1 transition engine to perform the appropriate generations and certifications. Here we benefitted from earlier groundwork: the fact that the initial Layer 1 was designed for potential FIPS validation led to a clean layout of state changes; our security paranoia led to a nice layout of staging, then atomically committing, state changes. The multiplicity of keys and identities added some wrinkles. For example, if Layer 1 decides to accept a new Layer 1 load, we now also need to to generate a new OA Manager keypair, and certify it *with the new Layer 1 keypair* as additional elements of this atomic change. We needed two passes before commitment, one to determine everyone's names should the change succeed, and another to then use these names in the construction of new certificates.

Data Structures and Naming. Naming the configuration and epoch entities in a way that users and developers was challenging—particularly since the initial architecture was designed in terms of parameters such as code version and owner, and the notion of "entity" is much murkier. We eventually included all the identifying information that was possibly relevant, but minimized redundancy—what's in the Layer 1 certificate is not repeated in the OA Manager certificate, and what's in the OA Manager certificate is not repeated in the application certificates. We simplify the developer's view by, in the API documentation, using "epoch" and "configuration" only for the Layer 3 variety—the only one they care about.

As lamented elsewhere, the commercial 4758 family has no notion of secure time—since the hardware real-time clock may drift, and since the hardware permits the application to set actual clock value. As a consequence, the security configuration code tracks time by *boot count*: a non-decreasing counter advanced each time the device is booted. We name epochs and configurations by the boot-count in which they started; we also track the count of how many configurations have occurred within this epoch.

5.3 Addressing the Issues

This design meets the criteria of Section 4.2.2:

- **Selective Preservation.** Applications can select either configuration or epoch lifetimes for their keys.
- **Retroactive Paranoia.** A party can determine the sequence of Layer 3 configurations so far in a particular epoch by authenticating the current configuration, and requesting to the see the sequence of OA Manager certificates for each previous configuration. By design, these are not deleted, and are linked together by count. An untrusted configuration within this epoch cannot hide its existence.
- **Careless Verification.** To mitigate this risk, we do not provide a "verify this chain" service—applications must explicitly walk the chain. Also, we gave different families of cards different factory roots, to avoid the risk of carelessly accepting the wrong entity.
- **Penetration Risk.** To mitigate this risk, we bound the OA Manager entity to a Layer 3 configuration.
- **Upstart Applications.** To mitigate this risk, we retain the private keys within the OA Manager, and only provide private-key services to the application.

By defining and separating entities, and considering how trust and certification interact, this design also addresses the issues of Section 2.3.

6 Alternatives and Future Work

We quickly enumerate some avenues for future research and reflection:

- **Alternative Software Structure.** Our OA design follows the 4758 architecture's sequence of increasingly less-trusted entities after boot. Some current research explores architectures that dispense with this limiting assumption, and also dispensing with the 4758 assumptions of one central loader/policy engine, and of a Layer 2 that exists only to serve Layer 3. It would be interesting to explore OA in these realms.
- **TCPA.** Since our work, the *Trusted Computing Platform Alliance* [11] has published ideas on how remote parties might gain assurance about the software configuration of remote desktop machines. It would be interesting to explore the interaction of our OA work with this now-timely topic, as well as the longer history of work in securely booting desktops [2, 12]
- **Standardized Certificate Formats.** As a consequence of the development process, our OA design uses 4758-specific certificate formats. Using a broader standard, such as SPKI, would enhance interoperability.
- **Witnesses.** The analysis and design presented in this paper assumes an authority making a statement about an entity at the time a keypair is created. Long-lived entities with the potential for run-time corruption suggest ongoing integrity-checking techniques. It would be interesting to examine OA in light of such techniques.
- **Formal Semantics for PKI.** Throughout this design process, we continually found ourselves reasoning about and explaining what a particular signature or certification scheme *meant*. (Indeed, one might characterize the entire 4758 architecture process as "tracing each dependency, and securing it.") This is an area of ongoing research for PKI in general (e.g., [5]). It would be interesting to examine the impact of such work in the context of secure coprocessors, where PKI-based OA may be the only way to verify what's going on inside the untampered box.

Our experience with the OA architecture in the 4758, like other aspects of this work, balanced the goals of enabling secure coprocessing applications while also living within product deadlines. OA enables Alice to design and release an application; Bob to download it into his coprocessor; and Charlie to then authenticate remotely that he's working with this application in an untampered device.

Outbound authentication allows third-party developers to finally deploy coprocessor applications, like Web servers [10], that can be authenticated by anyone on the Internet. Perhaps the most exciting avenue for future work will be the emergence and deployment of such applications.

Acknowledgments

The author gratefully acknowledges the comments and advice of the greater IBM 4758 team; particular thanks go to Mark Lindemann, who co-coded the Layer 2 OA Manager, and Jonathan Edwards, who tested the API and transformed my design notes into manuals and customer training material. The author is also grateful for the comments and advice of the Dartmouth PKI Lab on new research issues here, and the continual probing questions from Leendert Van Doorn and the current IBM Research team.

References

[1] C. Adams, S. Lloyd. *Understanding Public-Key Infrastructure.* Macmillian. 1999.

[2] W. A. Arbaugh, D. J. Farber, J. M. Smith. "A Secure and Reliable Bootstrap Architecture." *IEEE Computer Society Conference on Security and Privacy.* 1997.

[3] J. Dyer, R. Perez, S.W. Smith, M. Lindemann. 'Application Support Architecture for a High-Performance, Programmable Secure Coprocessor." *22nd National Information Systems Security Conference.* October 1999.

[4] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smtih, L. van Doorn, S. Weingart. "The IBM 4758 Secure Coprocessor: Overview and Retrospective." *IEEE Computer*, to appear, 2001.

[5] J. Howell and D. Kotz. *A Formal Semantics for SPKI* (extended version). Computer Science Technical Report TR2000-363, Dartmouth College. March 2000. (An earlier version appeared in ESORICS.)

[6] A. Perrig, D. Song, J. D. Tygar, S.W. Smith. "SAM: A Flexible and Secure Auction Architecture Using Trusted Hardware." *1st International Workshop on Internet Computing and Electronic Commerce.* IEEE Computer Society, 2001.

[7] S. W. Smith, E. R. Palmer, S. H. Weingart. "Using a High-Performance, Programmable Secure Coprocessor." *Proceedings, Second International Conference on Financial Cryptography.* Springer-Verlag LNCS, 1998.

[8] S.W. Smith, S.H. Weingart. "Building a High-Performance, Programmable Secure Coprocessor." *Computer Networks (Special Issue on Computer Network Security)*. 31: 831-860. April 1999.

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

[10] S. W. Smith. *WebALPS: Using Trusted Co-Servers to Enhance Privacy and Security of Web Interactions.* Research Report RC 21851, IBM TJ Watson Research Center. October 2000.

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

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

[13] B.S. Yee. *Using Secure Coprocessors.* Ph.D. thesis. Computer Science Technical Report CMU-CS-94-149, Carnegie Mellon University. May 1994.

[14] B.S. Yee and J.D. Tygar. "Secure Coprocessors in Electronic Commerce Applications." *1st USENIX Electronic Commerce Workshop.* 1996.

[15] B.S. Yee. *A Sanctuary For Mobile Agents.* Computer Science Technical Report CS97-537, UCSD, April 1997. (An earlier version of this paper appeared at the *DARPA Workshop on Foundations for Secure Mobile Code*.)