Outbound Authentication for Programmable Secure Coprocessors

***7th European Symposium on Research in Computer Security*, to appear**

Sean W. Smith⋆

Department of Computer Science
Dartmouth College
Hanover, New Hampshire USA
`http://www.cs.dartmouth.edu/~sws/`

Abstract. A programmable secure coprocessor platform can help solve many security problems in distributed computing. However, 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.

This paper offers our experiences in solving this problem for a high-end secure coprocessor product. This work required synthesis of a number of techniques, so that parties with different and dynamic views of trust can draw consistent and complete conclusions about coprocessor applications. These issues may be relevant to the industry's growing interest in rights management for general desktop machines.

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., [15, 22, 23]) 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, such an adversary could 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. [11] For another example, an adversary could subvert *decentralized e-cash* simply by increasing a register. However, secure coprocessors enable

---

solutions: the register lives inside a trusted box, which modifies the value only as part of a transaction with another trusted box. [22] Many other applications—including private information retrieval [3, 17], e-commerce co-servers [12], and mobile agents [24]—can also benefit from the high-assurance neutral environment that secure coprocessors could provide.

As the literature (e.g.,[5, 8]) discusses, achieving this potential requires several factors, including establishing and maintaining physical security, enabling the device to authenticate code-loads and other commands that come from the outside world, and building applications whose design does not negate the security advantages of the underlying platform.

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 in general. A signed statement *about* the configuration also does not suffice. For maximal effectiveness, the platform should enable the *entity itself* to have authenticated key pairs 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 software architecture for a programmable secure coprocessor platform must address the complexities of shipping, upgrades, maintenance, and hostile code, for a generic platform that can be configured and maintained in the hostile field. [16] Our team spent several years working on developing a such a device; other reports [7, 8, 18] present our experiences in bringing such a device into existence as a COTS product, the IBM 4758.

Although our initial security architecture [18] sketched a design for outbound authentication, we did not fully implement it—nor fully grasp the nature of the problem—until the Model 2 device. As is common in product development, we had to concurrently 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.

The Basic Problem. A relying party needs to conclude that a particular key pair really belongs to a particular software entity within a particular untampered coprocessor. Design and production constraints led to a non-trivial set of software entities in a coprocessor at any one time, and in any one coprocessor over time. Relying parties trust some of these entities and not others; furthermore, we needed to accommodate a *multiplicity* of trust sets (different parties have different views), as well as the *dynamic* nature of any one party's trust set over time.

This background sets the stage for the basic problem: how do we generate, certify, change, store, and delete private keys, so that relying parties can draw those conclusions consistent with their trust set, and only those conclusions?

This Paper. This paper is a post-facto report expanding on this research and development experience, which may have relevance both to other secure coprocessor technology (e.g., [21]) as well as to the growing industry interest in remotely authenticating what's executing on a desktop (e.g., [9, 19]).

Section 2 discusses the evolution of the problem in the context of the underlying technology. Section 3 presents the theoretical foundations. Section 4 presents the design. Section 5 suggests some directions for future study.

2 Evolution of Problem

2.1 Underlying Technology

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 battery-backed RAM (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 invokes Layer 1, and so on. (Because our device is an enclosed controlled system, we can avoid the difficulties of secure boot that arise in exposed desktop systems; we know execution starts in Layer 0 ROM in a known state, and higher-level firmware is changed only when the device itself permits it.) In the current family, Layer 2 is intended to be an internal operating system, leading to the constraint that it must execute at maximum CPU privilege; it invokes the single application (Layer 3) but continues to run, depending on its use of the CPU privilege levels to protect itself.

We intended the device to be a generic platform for secure coprocessor applications. The research team insisted on the goal that third parties (different from IBM, and from each other) be able to develop and install code for the OS layer and the application layer. 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. This design decision stemmed from our vision that application code and OS code may come from different entities who may not necessarily trust each other's updates; centralization of loading made it easier to enforce the appropriate security policies.

Layer 1 is updatable, in case we want to upgrade algorithms, fix bugs, or change the public key of the party authorized to update the layer. However, we mirror this

layer—updates are made to the inactive copy, which is then atomically made active for the next boot—so that failures during update will not leave us with a non-functioning code-loading 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 provide any on-line services to cards once they left. This constraint naturally suggested the use of *public-key cryptography* for authentication, both inbound and outbound.

Because of the last-touch-at-manufacturing constraint, we (the manufacturer) can last do something cryptographic at the factory.

2.2   User and Developer Scenarios

Discussions about potential relying parties led to additional requirements.

Developers were not necessarily going to trust each other. For example, although an application developer must trust the contents of the lower layers when his application is actually installed, he should be free to require that his secrets be destroyed should a lower layer be updated in a way he did not trust. As a consequence, we allowed each code-load to include a policy specifying the conditions under which that layer's secrets should be preserved across changes to lower layers. Any other scenario destroys secrets.

However, even full-preservation developers reserved the right to, post-facto, decide that certain versions of code—even their own—were untrusted, and be able to verify whether an untrusted version had been installed during their current epoch.

In theory, the OS layer should resist penetration by a malicious application; in practice, operating systems have a bad history here, so we only allow one application above it (and intend the OS layer solely to assist the application developer). Furthermore, we need to allow that some relying parties will believe that the OS in general (or specific version) may indeed be penetrable by malicious applications.

Small developers may be unable to assure the public of the integrity and correctness of their applications (e.g., through code inspection, formal modeling, etc). Where possible, we should maximize the credibility our architecture can endow on such applications.

2.3   On-Card Entities

One of the first things we need to deal with is the notion of what an on-card entity *is*. Let's start with a simple case: suppose the coprocessor had exactly one place to hold software and that it 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, this simple case raises a challenges. If a reload replaces C_1 with C_2, and reloads clear all tamper-protected memory, how does the resulting entity—C_2 on D_1—authenticate itself to a party on the other side of the net? The card itself would have

PSfrag replacements
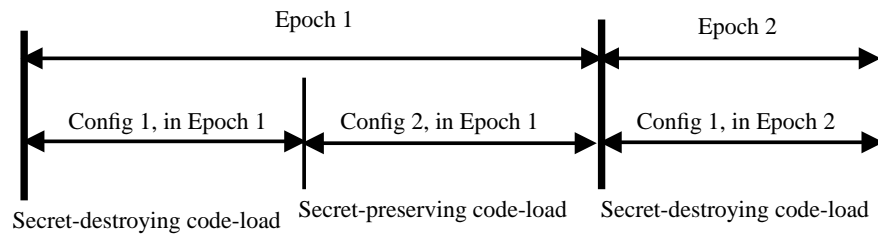
Fig. 1. An *epoch* starts with code-load action that clears a layer's secrets; with an epoch, each secret-preserving code-load starts a new *configuration*.

no secrets left, since the only data storage hidden from physical attack was cleared. Consequently, any authentication secrets would have to come with C_2, and we would start down a path of shared secrets and personalized code-loads. 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?

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

Secret Retention. As noted, developers demanded that we sometimes permit secret retention across reload. With a secret-preserving load; the entity may stay the same, but the code may change. The conflicting concepts that developers had about what exactly happens to their on-card entity when code update occurs lead us to think more closely about entity lifetimes. We introduce some language to formalize that. (Figure 1 sketches these concepts.)

Definition 1 (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.*

An 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.

2.4 Authentication Scenarios

This design left us with on-card software entities made up of several components with differing owners, lifetimes, and state. A natural way to do outbound authentication to give the card a certified key pair, whose private key lives in tamper-protected memory. However, the complexity of the entity structure creates numerous problems.

Application Code. Suppose entity C is the code C_1 residing in the application Layer 3 in a particular device. C 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.

If a relying party P trusts C_1, C_2, and C_3 to be free of flaws, vulnerabilities, and malice, then the natural approach might work. However, if P distrusts some of this code, then problems arise.

- If P does not trust C_1, then how can P distinguish between an entity with the C_2 patch, and an entity with a corrupt C_1 pretending to have the C_2 patch?
- If P does does not trust C_2, then then how can P distinguish between the an entity with the honest C_1, and an entity with the corrupt C_2 pretending to be the honest C_1? (The mere existence of a signed update command compromises all the cards.)
- If P does not trust C_3, then how can P distinguish between the honest C_1 and a malicious C_3 that pretends to be C_1?

Code-Loading Code. Even more serious problems arise if a corrupted version of the configuration software 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 impersonation.

OS Code. Problems can also arise because the OS code changes. Debugging an application requires an operating system with debug hooks; in final development stages, a reasonable 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, a party P cannot distinguish between the application running securely with the real OS, the application with debug hooks underneath it, and the application with the real OS but with a policy that permits hot-update to the debug version. The private key would be the same in all cases.

Internal Certification. The above scenarios suggest that perhaps a single key pair (for all entities in a card for the lifetime of the card) may not suffice. However, extending to schemes where one on-card entity generates and certifies key pairs for other on-card entities also creates challenges.

For example, suppose Layer 1 generates and certifies key pairs 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 key pair 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—since it could have seen the Layer 1 private key.

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.

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 key pair depends on the *future* behavior of the certifier. In the 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 the application (and its secrets) be destroyed, if the underlying OS undergoes an update the application developer does not trust. What if the untrusted OS has access to a private key used in certifying the original application?

We revisit these issues in Section 4.3.

3   Theory

The construction of the card suggests that we use certified key pairs for outbound authentication. However, as we just sketched, the straightforward approach of just sending the card out with a certified key pair permits trouble.

In this section, we try to formalize the principles that emerged while considering this problem.

A card leaves the factory and undergoes some sequence of code loads and other configuration changes. A relying party interacts with an entity allegedly running inside this card. The card's OA scheme enables this application to wield a private key and to offer a collection of certificates purporting to authenticate its keyholder.

It would be simplest if the party could use a straightforward validation algorithm on this collection. As Maurer [13, 14] formalized, a relying party's validation algorithm needs to consider which entities that party trusts. Our experience showed that parties have a wide variety of trust views. Furthermore, we saw the existence of two spaces: the conclusions that a party *will* draw, given an entity's collection of certificates and the party's trust view; and the conclusions that a party *should* draw, given the history of those keyholders and the party's trust view.

We needed to design a scheme that permits these sets of conclusions to match, for parties with a wide variety of trust views.

3.1   What the Entity Says

Relying party P wants to authenticate interaction with a particular entity E. Many scenarios could exist here; for simplicity, our analysis reduces these to the scenario of E needing to prove to P that $\mathsf{own}(E, K)$: that E has exclusive use of the private element of key pair K.

We need to be able to talk about what happens to a particular coprocessor.

Definition 2 (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.*

In the context of OA for coprocessors that cannot be opened or otherwise examined, and that disappear once they leave the factory, it seemed reasonable to impose the restriction that on-card entities carry their certificates. For simplicity, we also imposed the restriction that they present the same fixed set no matter who asks.

Definition 3. *When entity E wishes to prove it owns K after history H, let* Chain(E, K, H) *denote the set of certificates that it presents.*

3.2 Validation

Will a relying party P believe that E owns K?

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 whose statements about certificates P trusts. Let* root *be the factory CA: the trust root for the card. A legitimate* trust set *is one that contains* root.

Our scheme needs to accommodate *any legitimate trust set*m since discussion with developers (and experiences doing security consulting) suggested that relying parties would have a wide divergence of trust sets, which may change over time.

In the context of OA for coprocessors, it was reasonable to impose the restriction that the external party decides validity based on an entity's chain and the party's own list of trusted entities. (The commercial restriction that we can never count on accessing cards after they leave made revocation infeasible.) We formalize that:

Definition 5 (Trust-set scheme). *A* trust-set *certification scheme is one where the relying party's* Validate *algorithm is deterministic on the variables* Chain(E, K, H) *and* TrustSet(P).

3.3 Dependency

The problem scenarios in Section 2.4 arose because one entity E_1 had an unexpected avenue to use the private key that belonged to another entity E_2. We need language to express these situations, where the integrity of E_2's key actions *depends* on the correct behavior of E_1.

Definition 6 (Dependency Function). *Let $\mathcal{E}$ be the set of entities. A* dependency function *is a function $\mathcal{D} : \mathcal{E} \longrightarrow 2^{\mathcal{E}}$ such that, for all E_1, E_2:*

– *$E_1 \in \mathcal{D}(E_1)$*
– *if $E_2 \in \mathcal{D}(E_1)$ then $\mathcal{D}(E_2) \subset \mathcal{D}(E_1)$*

When a dependency function depends on the run R, we write $\mathcal{D}_R$.

What dependency function shall we use for analysis? In our specialized hardware, code runs in a single-sandbox controlled environment which (if the physical security works as intended) is free from outside observation or interference. Hence, in our analysis, dependence follows read and write:

Definition 7. *For entities E_1 and E_2 in run R, we write $E_2 \xrightarrow{\text{data}}_R E_1$ when E_1 has read/write access to the secrets of E_2 ($E_2 \xrightarrow{\text{data}}_R E_2$ trivially) and $E_2 \xrightarrow{\text{code}}_R E_1$ when E_1 has write access to the code of E_2. Let $\longrightarrow_R$ be the transitive closure of the union of these two relations. For an entity E in a run R, define $\text{Dep}_R(E)$ to be $\{F : E \longrightarrow_R F\}$.*

In terms of the coprocessor, if C_1 follows B_1 in the post-boot sequence, then we have $C_1 \xrightarrow{\text{data}}_R B_1$ (since B_1 could have manipulated data before passing control). If C_2 is a secret-preserving replacement of C_1, then $C_1 \xrightarrow{\text{data}}_R C_2$ (because C_2 still can touch the secrets C_1 left). If A can reburn the FLASH segment where B lives, then $B \xrightarrow{\text{code}}_R A$ (because A can insert malicious code into B, that would have access to B's private keys).

3.4   Consistency and Completeness

Should the relying party draw the conclusions it actually will? In our analysis, security dependence depends on the run; entity and trust do not. This leads to a potential conundrum. Suppose, in run R, $C \longrightarrow_R B$ and $C \in \mathsf{TrustSet}(P)$, but $B \notin \mathsf{TrustSet}(P)$. Then a relying party P cannot reasonably accept any signed statement from C, because B may have forged it.

To capture this notion, we define *consistency* for OA. The intention of consistency is that if the party concludes that an message came from an entity, then it really did come from that entity—modulo the relying party's trust view. That is, in any $H \prec R$ where P concludes $\mathsf{own}(E, K)$ from $\mathsf{Chain}(E, K, H)$, if the entities in $\mathsf{TrustSet}(P)$ behave themselves, then E really does own K. We formalize this notion:

Definition 8. *An OA scheme is* consistent *for a dependency function $\mathcal{D}$ when, for any entity E, a relying party P with any legitimate trust set, and history and run $H \prec R$:*

$$\mathsf{Validate}(P, \mathsf{Chain}(E, K, H)) \implies \mathcal{D}_R(E) \subseteq \mathsf{TrustSet}(P)$$

One might also ask if the relying party *will* draw the conclusions it actually *should*. We consider this question with the term *completeness*. If in any run where E produces $\mathsf{Chain}(E, K, H)$ and $\mathcal{D}_R(E)$ is trusted by P—so in P's view, no one who had a chance to subvert E would have—then P should conclude that E owns K.

Definition 9. *An OA scheme is* complete *for a dependency function $\mathcal{D}$ when, for any entity E, relying party P with any legitimate trust set, and history and run $H \prec R$:*

$$\mathcal{D}_R(E) \subseteq \mathsf{TrustSet}(P) \implies \mathsf{Validate}(P, \mathsf{Chain}(E, K, H))$$

These definitions equip us to formalize a fundamental observation:

Theorem 1. *Suppose a trust-set OA scheme is both consistent and complete for a given dependency function $\mathcal{D}$. Suppose entity E claims K in histories $H_1 \prec R_1$ and $H_2 \prec R_2$. Then:*

$$\mathcal{D}_{R_1}(E) \neq \mathcal{D}_{R_2}(E) \implies \mathsf{Chain}(E, K, H_1) \neq \mathsf{Chain}(E, K, H_2)$$

Proof. Suppose $\mathcal{D}_{R_1}(E) \neq \mathcal{D}_{R_2}(E)$ but $\mathsf{Chain}(E, K, H_1) = \mathsf{Chain}(E, K, H_2)$. We cannot have both $\mathcal{D}_{R_1}(E) \subseteq \mathcal{D}_{R_2}(E)$ and $\mathcal{D}_{R_2}(E) \subseteq \mathcal{D}_{R_1}(E)$, so, without loss of generality, let us assume $\mathcal{D}_{R_2}(E) \not\subseteq \mathcal{D}_{R_1}(E)$. There thus exists a set S with $\mathcal{D}_{R_1}(E) \subseteq S$ but $\mathcal{D}_{R_2}(E) \not\subseteq S$.

Since the scheme is consistent and complete, it must work for any legitimate trust set, including S. Let party P have $S = \mathsf{TrustSet}(P)$. Since this is a trust-set certification scheme and E produces the same chains in both histories, party P must either validate these chains in both scenarios, or reject them in both scenarios. If party P accepts in run R_2, then the scheme cannot be consistent for $\mathcal{D}$, since E depends on an entity that P did not trust. But if party P rejects in run R_1, then the scheme cannot be complete for $\mathcal{D}$, since party P trusts all entities on which E depends.

3.5   Design Implications

We consider the implications of Theorem 1 for specific ways of constructing chains and drawing conclusions, for specific notions of dependency. For example, we can express the standard approach—P makes its conclusion by recursively verifying signatures and applying a basic inference rule—in a Maurer-style calculus [13]. Suppose C is a set of *certificates*: statements of the form $K_1 \mathsf{\ says\ own}(E_2, K_2)$. Suppose S be a set of entities *trusted to speak the truth about certificate ownership*: $\{E_1 : \mathsf{trust}(E_1, \mathsf{own}(E_2, K_2))\}$. A relying party may start by believing $C \cup \{\mathsf{own}(\mathsf{root}, K_{\mathsf{root}})\}$.

We can define $\mathsf{View_{will}}(C, S)$ to be the set of statements derivable from this set by applying the rule

$$\mathsf{own}(E_1, K_1), \quad E_1 \in S, \quad K_1 \mathsf{\ says\ own}(E_2, K_2) \ \vdash \ \mathsf{own}(E_2, K_2)$$

The $\mathsf{Validate}$ algorithm for party P then reduces to the decision of whether $\mathsf{own}(E, K)$ is in this set.

We can also express what a party should conclude about an entity, in terms of the chain the entity presents, and the views that the party has regarding trust and dependency. If $\mathcal{D}$ is a dependency function, we can define $\mathsf{View_{should}}(C, S, \mathcal{D})$ to be the set of statements derivable by applying the alternate rule:

$$\mathsf{own}(E_1, K_1), \quad \mathcal{D}(E_1) \subseteq S, \quad K_1 \mathsf{\ says\ own}(E_2, K_2) \ \vdash \ \mathsf{own}(E_2, K_2)$$

In terms of this calculus, we obtain consistency be ensuring that for any chain and legitimate trust set, and $H \prec R$, the set $\mathsf{View_{will}}(\mathsf{Chain}(E, K, H), S)$ is contained in the set $\mathsf{View_{should}}(\mathsf{Chain}(E, K, H), S, \mathcal{D}_R)$. The relying party should only use a certificate to reach a conclusion when the entire dependency set of the signer is in $\mathsf{TrustSet}(P)$.

4   Design

For simplicity of verification, we would like $\mathsf{Chain}(E, K, H)$ to be a literal chain: a linear sequence of certificates going back to root. To ensure consistency and completeness, we need to make sure that, at each step in the chain, the partial set of certifiers equals the dependency set of that node (for the dependency function we see relying parties using). To achieve this goal, the elements we can manipulate include generation of this chain, as well as how dependency is established in the device.

4.1   Layer Separation

Because of the post-boot execution sequence, code that executes earlier can subvert code that executes later.[1] If B, C are Layer i, Layer $i+1$ respectively, then $C \longrightarrow_R B$ unavoidably.

However, the other direction should be avoidable, and we used hardware to avoid it. 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.

To ensure $B \overset{data}{\not\longrightarrow}_R C$, we reserved a portion of BBRAM for B, and used the ratchet hardware to enforce access control. 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.

To keep configuration entities from needlessly depending on the epoch entities, in our Model 2 device, we subdivided the higher BBRAM to get four regions, one each for epoch and configuration lifetimes, for Layer 2 and Layer 3. The initial clean-up code in Layer 1 (already in the dependency set) zeroizes the appropriate regions on the appropriate transition. (For transitions to a new Layer 1, the clean-up is enforced by the *old* Layer 1 and the permanent Layer 0—to avoid incurring unnecessary dependency on the new code.)

4.2   The Code-Loading Code

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. But this centralization creates some issues for OA. Suppose the code-loading Layer 1 entity A_1 is reloaded with A_2. Business constraints dictated that A_1 do the reloading, because the ROM code had no public-key support. It's unavoidable that $A_2 \overset{code}{\longrightarrow}_R A_1$ (because A_1 could have cheated, and not installed the correct code). However, 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 key pair for its successor A_2; A_1 uses its current key pair to sign a *transition certificate* attesting to this change of versions and key pairs; and A_1 destroys its current private key.

This technique—which we implemented and shipped with the Model 1 devices in 1997—differs from the standard concept of *forward security* [1] in that we change keys with each new version of software, and ensure that *the name of the new version is spoken by the old version.* As a consequence, a single malicious version cannot hide its presence in the trust chain; for a coalition of malicious versions, the trust chain will name at least one. (Section 5 considers this further.)

---

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

B_1

Appeared in 7th European Symposium on Research in Computer Security, 2002.

B_2

12   DRAFT
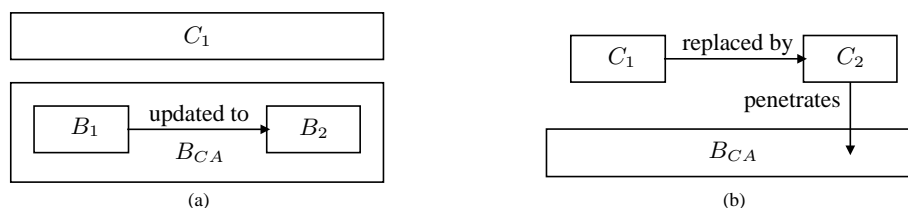
updated to

Fig. 2. Having the OS certify key pairs for the application creates interesting lifetime issues. In (a), if the OS is updated while preserving its key pair, then the application depends on both versions; in (b), if the OS outlives the application but is potentially penetrable, then the application may depend on future applications.

4.3   The OA Manager

Since we do not know a priori what device applications will be doing, we felt that application key pairs needed to be created and used at the application's discretion. Within our software architecture, Layer 2 should do this work—since it's easier to provide these services at run-time instead of reboot, and the Layer 1 protected memory is locked away before Layer 2 and Layer 3 run.

This *OA Manager* component in Layer 2 will wield a key pair generated and certified by Layer 1, and will then generate and certify key pairs at the request of Layer 3. These certificates indicate that said key pair belongs to an application, and also include a field chosen by the application. (A fuller treatment of our trust calculus would thus distinguish between owning and trusting a key pair for certification purposes, and owning and trusting a key pair for the application-specified purpose—the last link.)

To keep the chain linear, we decided to have Layer 1 generate and destroy the OA Manager key pair (e.g., instead of adding a second horizontal path between successive versions of the OA Manager key pairs). The question then arises of when the OA Manager key pair should be created and destroyed.

We discuss some false starts. If the OA Manager outlived the Layer 2 configuration, then our certification scheme *cannot be both consistent and complete.* For a counterexample (see Figure 2, (a)) suppose that application C_1 is a configuration-entity on top of OS B_1; that OS B_1 changes code to OS B_2 but both are part of the same entity B_{CA}; and that party P trusts C_1, B_1 but not B_2. For the scheme to be complete, P should accept certificate chains from C_1—but that means accepting a chain from B_{CA}, and $B_{CA} \longrightarrow_R B_2 \notin \mathsf{TrustSet}(P)$.

This counterexample fails because the application entity has a CA whose dependency set is larger than the application's. Limiting the CA to the current Layer 2 configuration eliminates this issue, but still fails to address penetration risk. 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.* (See Figure 2, (b).)

Our final design avoided these problems by having the Layer 2 OA Manager live *exactly* as long as the Layer 3 configuration. Using the protected BBRAM regions, we ensure that upon any change to the Layer 3 configuration, Layer 1 destroys the old OA Manager private key, generates a new key pair, and certifies it to belong to the new OA

Appeared in 7th European Symposium on Research in Computer Security, 2002.

DRAFT   13

Manager for the new Layer 3 configuration. This approach ensures that the trust chain names the dependency set for Layer 3 configurations—even if dependency is extended to include penetration of the OS/application barrier.

Since we need to accommodate the notion of both *epoch* and *configuration* lifetimes (as well as to allow parties who choose the former to change their minds), we identify entities both by their current epoch, as well as the current configuration within that epoch. When it requests a key pair, the Layer 3 application can specify which lifetime it desires; the certificate includes this information. Private keys for a configuration lifetime are kept in the Layer 3 configuration region in BBRAM; private keys for an epoch lifetime are kept in the epoch region.

Note that a Layer 3 epoch certificate (say, for epoch E) still names the configuration (say, C_1) in which it began existence. If, in some later configuration C_k within that same epoch, the relying party decides that it wants to examine the individual configurations to determine the whether an untrusted version was present, it can do that by examining the trust chain for C_k and the sequence of OA Manager certificates from C_1 to C_k. An untrusted Layer 1 will be revealed in the Layer 1 part of the chain; otherwise, the sequence of OA Manager certificates will have correct information, revealing the presence of any untrusted Layer 2 or Layer 3 version.

4.4   Summary

As noted earlier, the trust chain for the current Layer 1 version starts with the certificate the factory root signed for the first version of Layer 1 in the card, followed by the sequence of transition certificates for each subsequent version of Layer 1 installed. The trust chain for the OA Manager appends the OA Manager certificate, signed by the version of Layer 1 active when that Layer 3 configuration began, and providing full identification for the current Layer 2 and Layer 3 configurations and epochs. The trust chain for a Layer 3 key pair appends the certificate from the OA Manager who created it.

Our design thus constitutes a trust-set scheme that is consistent and complete for the dependency function we felt was appropriate.

4.5   Implementation

Full support for OA shipped with all Model 2 family devices and the CP/Q++ embedded operating system. (The announced Linux port [10] still has the Layer 1 OA hooks; extending Linux to handle that is an area of future work.)

Implementation required some additional design decisions. To accommodate small developers (Section 2.2), we decided to have the OA Manager retain all Layer 3 private keys and wield them on the application's behalf; consequently, a party who trusts the penetration-resistance of a particular Layer 2 can thus trust that the key was at least used within that application on an untampered device. Another design decision resulted from the insistence of an experienced application architect that users and developers will not pay attention to details of certificate paths; to mitigate this risk, we do not provide a "verify this chain" service—applications must explicitly walk the chain—and we gave different families of cards different factory roots.

A few aspects of the implementation also proved surprising. One aspect was the fact that the design required two APIs: one between Layer 1 and Layer 2, and another between Layer 2 and the application. Another aspect was finding places to store keys. We extended the limited area in BBRAM by storing a MAC key and a TDES encryption key in each protected region, and storing 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. Another interesting aspect was the multiplicity of keys and identities added when extending the Layer 1 transition engine to perform the appropriate generations and certifications. For example, if Layer 1 decides to accept a new Layer 1 load, we now also need to generate a new OA Manager key pair, and certify it *with the new Layer 1 key pair* as additional elements of this atomic change. Our code thus 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.

As has been noted elsewhere [8], we regret the design decisions to use our own certificate format, and the fact that the device has no form of secure time (e.g., Layer 3 can always change the clock). Naming the configuration and epoch entities was challenging— particularly since the initial architecture was designed in terms of parameters such as code version and owner, and a precise notion of "entity" only emerged later.

5 Conclusions

One might characterize the entire OA architecture process "tracing each dependency, and securing it." Our experience here, 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, such as Web servers [12] and rights management boxes [11], that can by authenticated by anyone in the Internet, and participate in PKI-based protocols.

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 a one-application Layer 3. It would be interesting to explore OA in these realms.

Similarly, the analysis and design presented in this paper assumes that an authority makes a statement about an entity at the time a key pair 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.

Alternate Platforms Since our work, the *Trusted Computing Platform Alliance* [19] has published ideas on how remote parties might gain assurance about the software

configuration of remote desktop machines; Microsoft has considered similar ideas [9], and others are entering this space.

It would be interesting to explore the interaction of our OA work with this increasingly timely topic, as well as the longer history of work in securely booting desktops [2, 6, 20].

Alternate Cryptography We developed our transition certificate scheme for Layer 1 to ensure that not all corrupt entities could hide their presence in a chain. Entities aside, this scheme is essentially a basic forward-secure signature scheme (e.g., [4], Sec. 3.3). It would be interesting how the broader space of forward-secure signature schemes might be used in these settings.

Alternate Dependency Our dependency function—entity E_1 can subvert E_2 when it can read or write secrets, or write code, at any time—emerged for the special case of our device. A more careful incorporation of time would be interesting, as would an examination of the other avenues of manipulation in more complex settings (e.g., the opportunities for covert channels in common desktop operating systems, or if the coprocessor *cryptopages* to the host file system).

Formalizing Trust Sets One linchpin of our design was the divergence and dynamic nature of what relying parties tend to trust. (Consequently, our analysis assumed that parties may have "any legitimate trust set.") It would be interesting to measure and formally characterize the trust behavior that occurs (or should occur) with real-world relying parties and software entities.

Formalizing Penetration Recovery Much complicated reasoning arose from scenarios such as "what if, in six months, trusted software component X turns out be flawed?" Further exploration of the design and implementation of authentication schemes that explicitly handle such scenarios would be interesting.

References

1. R. Anderson. Invited lecture, *ACM CCS*, 1997. The definitive written record appears to be *Two Remarks on Public Key Cryptology,* `http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/forwardsecure.pdf`

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. D. Asonov and J. Freytag. "Almost Optimal Private Information Retrieval." *Privacy Enhancing Technology 2002*. Springer-Verlag LNCS.

4. M. Bellare and S. Miner. "A Forward-Secure Digital Signature Scheme." Extended abstract appears in *Crypto 99*, Springer-Verlag LNCS. Full version at `http://www-cse.ucsd.edu/~mihir/papers/fsig.html`

5. M. Bond, R. Anderson. "API-Level Attacks on Embedded Systems." *IEEE Computer.* 34:67-75. October 2001.

6. L. Clark and L.J. Hoffmann. "BITS: A Smartcard Protected Operating System." *Communications of the ACM.* 37: 66-70. 1994.

7. 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.

8. J. Dyer, M. Lindemann, R. Perez, R. Sailer, S.W. Smith, L. van Doorn, S. Weingart. "Building the IBM 4758 Secure Coprocessor." *IEEE Computer*, 34:57-66. October 2001.

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

10. "IBM Research Demonstrates Linux Running on Secure Cryptographic Coprocessor." Press release, August 28, 2001.

11. A. Iliev, S.W. Smith. "Prototyping an Armored Data Vault: Rights Management for Big Brother's Computer." *Privacy Enhancing Technology 2002*. Springer-Verlag LNCS.

12. S. Jiang, S. W. Smith, K. Minami. "Securing Web Servers against Insider Attack." *ACSA/ACM Annual Computer Security Applications Conference.* December 2001.

13. R. Kohlas and U. Maurer. "Reasoning About Public-Key Certification: On Bindings Between Entities and Public Keys." *Journal on Selected Areas in Communications.* 18: 551-560. 2000. (A preliminary version appeared in *Financial Cryptography 99*, Springer-Verlag.)

14. U. Maurer. "Modelling a Public-Key Infrastructure." *ESORICS 1996*. Springer-Verlag LNCS.

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

16. 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.

17. S.W. Smith, D. Safford. "Practical Server Privacy Using Secure Coprocessors." *IBM Systems Journal (special issue on End-to-End Security)* 40: 683-695. 2001.

18. 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.

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

20. 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.

21. N. van Someren. "Access Control for Secure Execution." *RSA Conference 2001.*

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

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

24. 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.*)