Int J Inf Secur (2004) 3: 28–41 / Digital Object Identifier (DOI) 10.1007/s10207-004-0033-0

Outbound authentication for programmable secure coprocessors

Sean W. Smith

Department of Computer Science, Dartmouth College, Hanover NH 03755, USA
e-mail: sws@cs.dartmouth.edu

Published online: 19 May 2004 –

Abstract. A programmable secure coprocessor platform can help solve many security problems in distributed computing, particularly if coprocessor applications can participate as full-fledged parties in distributed cryptographic protocols. Thus, 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. 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 sound and complete conclusions about remote coprocessor applications.

Keywords: Secure coprocessors – Authentication – Attestation – Trust

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.

1.1 Secure coprocessing

To address this problem, research (e.g., [20, 28, 30]) has long explored the potential of *secure coprocessors*: hardware devices that, with high assurance, 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 [12]. 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 [5, 28]. Many other applications – including private information retrieval [2, 21], e-commerce coservers [13], and mobile agents [29] – can also benefit from the high-assurance neutral environment that secure coprocessors could provide.

As discussed in the literature (e.g., [8]), achieving this potential requires several factors, including establishing and maintaining physical security, and enabling the device to authenticate code loads and other commands that come from the outside world. Achieving this potential also requires building applications whose design does not negate the security advantages of the underlying platform (e.g., [4]).

However, using secure coprocessors to secure distributed computation *also* requires *outbound authentication* (OA): the ability of coprocessor applications 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 (e.g., 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 to 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.

(In theory, solutions where the entity does not possess its own key pair but makes use of some other service are also possible, but we did not consider them because they do not easily mesh with standard approaches – where the relying party can draw a conclusion based on whether an entity proves knowledge of a private key – and would also overly complicate the API for key usage.)

1.2 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 [22]. Our team spent several years working on developing just such a device; other reports [8, 9, 23] present our experiences in bringing such a device into existence as a COTS product, the IBM 4758.

Although our initial security architecture [23] 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 released in 2000. 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 we satisfied legacy application concerns.

1.3 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 nontrivial 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. Software and hardware structure can introduce further dependencies that might affect the conclusions relying parties would reach if they knew these dependencies; for example, if a coprocessor permitted multiple concurrent applications, some of these were hostile, and the OS (through oversight) permitted hostile applications to subvert other ones, then a relying party might want to know additional details about the software configuration surrounding a particular application instance. 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[1] over time.

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

[1] See Sects. 4.4 and 4.5 for more discussion.

1.4 This paper

This paper is a post-facto report expanding on this research and development experience that may have relevance both to other secure coprocessor technology and to the growing industry interest in remotely authenticating what's executing on a desktop (e.g., [10, 19, 26]).

Section 2 discusses the emergence 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 Problem

2.1 Underlying technology

We start with a brief overview of the 4758, a PCI card that contains a general-purpose computing environment (486 class), cryptographic accelerators, and hardware RNG. The device is *tamper-responding*: with high assurance, on-board circuits detect tamper attempts and destroy the contents of volatile RAM and nonvolatile *battery-backed RAM (BBRAM)* before an adversary can see them. The 4758 also contains nonvolatile FLASH that is not destroyed upon tamper – but we do check integrity and restrict writing ability. The internal software of the 4758 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. Our 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 could always be revived. Thus, we needed to develop a code-loading architecture that accommodated both sets of goals.

For code loading, one might develop a hierarchical design where each layer chooses what to install in the next layer. Instead of this hierarchical approach, we converged

on a centralized design where Layer 1 contains the security configuration software that 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 that 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 nonfunctioning code-loading layer.

2.2 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 or database of card-specific data (secret or otherwise), nor provide any online services to cards once they left the factory. This constraint naturally suggested the use of *public-key cryptography* for authentication, both inbound and outbound. This choice separates cards from relying parties and frees us from having to track any association of particular cards with their ultimate location, users, and applications.

For inbound authentication, we can preinstall a public key (in FLASH) telling the card who it should listen to for its first command. For outbound authentication, the natural approach is to 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 (and because of a design assumption that the manufacturer would be the central trust root for these devices), the last time we can ensure that an external trust point can interact with the card and sign such certificates is at the factory – after that, it's up to the card itself.

2.3 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 does 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 those developers who wanted the device to preserve secrets across updates reserved the right to, post facto, decide that certain versions of code – even their own – were untrusted and to verify whether an untrusted version had been installed during the lifetime of their secrets.

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. (That is, the OS can support multiple concurrent processes, but we assume these are all in the same trust domain.) Furthermore, we need to allow that some relying parties will believe that the OS in general (or a specific version) may indeed be penetrable by malicious applications.

Small-scale developers (without a large preestablished reputation) 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 applications from such developers.

(We note that this design assumption of "one application space" was driven by the generally poor record of operating systems in this regard and by the lack of a suitable high-assurance candidate at the time of product development. Considering our problem in the framework of a higher assurance operating system, where this restriction may be unnecessary, or a general-purpose desktop, where this restriction may be unacceptable, is an interesting area of future work.)

2.4 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 reinstallation of C_1 inside D_1.

However, this simple case raises 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 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.

This line of thinking leads to questions. 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? (As we discuss later, one can even make a case that an entity

should include entities that were previously installed but are no longer present on the card.)

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.

2.5 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 a code update occurs led us to think more closely about entity lifetimes. We introduce some language – *epoch* and *configuration* – to formalize that.

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, if the code-loading officer included policies that permitted such persistence across such changes.

For example, Layer N may undergo a transition such as a hot update, a complete reinstall, or a surrender.[2] A hot update will begin a new N configuration but will preserve the old N epoch; whether it preserves a K epoch (for $K > N$) depends on the policy the owner of Layer K has established.

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 in layers $1 \ldots N$. 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 only for a Layer N configuration.*

[2] Table 4 in [23] and Appendix F in the *IBM 4758 PCI Cryptographic Coprocessor Custom Software Developer's Toolkit Guide* give more details on the transition sequences.

2.6 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 is 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 reinstall 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? (See Fig. 2.)
- If P does does not trust C_2, then then how can P distinguish between 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 – since the relying party cannot know whether any particular card carried out this update. See Fig. 3.)
- 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? (Essentially, this is isomorphic to Fig. 3.)

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,

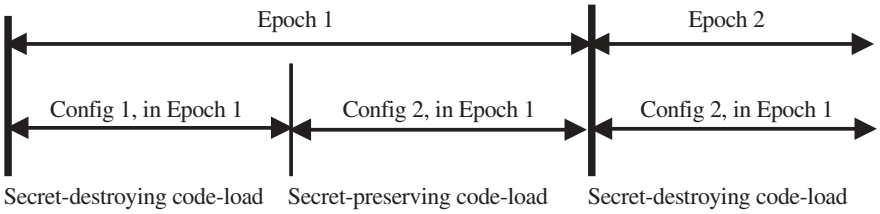

Fig. 1. An *epoch* starts with code-load action that clears a layer's secrets; with an epoch, each secret-preserving code load starts a new *configuration*

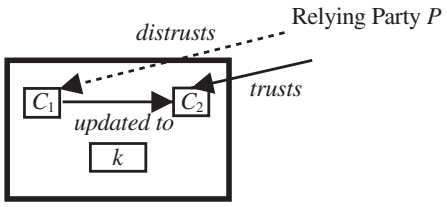

Fig. 2. Replacing untrusted software with trusted software, while retaining the private key, creates trust problems

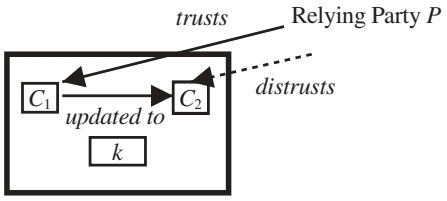

Fig. 3. The potential to replace trusted software with untrusted software, while retaining the private key, also creates trust problems

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 updates to the debug version. The private key would be the same in all cases.

2.7 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. If two different entities, one trusted and one untrusted, had access to the same private key material, then the relying party can no longer draw a reasonable conclusion from use of the private key alone. If we want to enable the relying party to do this, the natural generalization is to have separate keys for separate entities. 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 case of the coprocessor, this observation has 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? (This observation might also have implications for other types of PKI, where a CA/RA both generates as well as certifies user key pairs.)

We revisit all these issues in Sect. 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 [14, 17] 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 that change dynamically. 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. For this interaction to be meaningful, P must already trust E to behave correctly with its

keys (we will elaborate on this point in Sect. 4.3). 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; that (in P's view) no one who might subvert this will do so.

We need to be able to talk about what happens to a particular coprocessor: both a long-term sequence of actions and specific instants along that sequence. So we introduce some notation – *history* and *run* – for these concepts.

A coprocessor can take action only in the context of the particular history H that it has experienced to that point in time. However, we need to consider both history and run because this run may continue in several different ways beyond that point and the actions in these potential futures may be relevant to the conclusions a relying party draws from an action the coprocessor takes now.

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 $\mathsf{Chain}(E, K, H)$ denote the set of certificates that it presents.*

3.2 What the relying party concludes

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 of which ones it is unsure.

Definition 4. *For a party P, let $\mathsf{TrustSet}(P)$ 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.*

As discussed earlier, this project arose in the context of a specific commercial product effort, which imposed some specific constraints. In particular: we could not construct a database of coprocessors, we could not track where they went, we could neither contact nor audit them, we could not assume that they or their applications or relying parties would have network access back to the factory. These constraints made revocation infeasible. Consequently, for the problem space we faced, it was reasonable to impose the restriction that the external party decided validity based on an entity's chain and the party's own list of trusted entities.

We formalize this notion of "reasonable" validation schemes.

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

We thus needed to design a trust-set certification scheme that accommodates *any legitimate trust set* since discussion with developers (and experiences doing security consulting) suggested that relying parties would have a wide divergence of opinions about which versions of which software they trust.

3.3 Dependency

The problem scenarios in Sect. 2.6 arose because one entity E_2 had an unexpected avenue to use the private key that belonged to another entity E_1. We need language to express these situations, where the integrity of E_1's key actions *depends* on the correct behavior of E_2.

We formalize this concept as a *dependency function*, taking an entity to the set of entities that can subvert its correct operation, with respect to private keys.

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, we have:*

- $E_1 \in \mathcal{D}(E_1)$ *(Idempotency)*
- *if* $E_2 \in \mathcal{D}(E_1)$ *then* $\mathcal{D}(E_2) \subset \mathcal{D}(E_1)$ *(Transitivity).*

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

Different entity architectures give rise to different appropriate dependency functions.

In our specialized hardware, code runs in a single-sandbox controlled environment that (if the physical security works as intended) is free from outside observation or interference. Hence, in our analysis, dependence should follow from the ability of an entity to read or write another entity's secrets or to modify code that can read or write another entity's secrets.

Definition 7. *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.
($E_2 \xrightarrow{data}_R E_2$ trivially.)
We write

$$E_2 \xrightarrow{code}_R E_1$$

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

$$Dep_R(E) = \{F : E \longrightarrow_R F\}.$$

The intuition here is that, for the coprocessor architecture we considered, $Dep_R(E)$ lists all the on-card software entities that could have subverted the correct operation of entity E in run R.

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

3.4 Soundness

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, we have:

- $C \longrightarrow_R B$ 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 *soundness* for OA. The intention of soundness is that if a relying party concludes that a message came from an entity, then it really did come from that entity – modulo the relying party's trust view. The party will not conclude that it *should* trust the entity if such a conclusion is inconsistent with the party's beliefs.

That is, suppose in some history $H \prec R$, P concludes $\mathsf{own}(E, K)$ from $\mathsf{Chain}(E, K, H)$. If the $\mathsf{TrustSet}(P)$ entities behave themselves, then E should really own K. We formalize this notion:

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

$$\mathit{Validate}(P, \mathit{Chain}(E, K, H)) \Longrightarrow \mathcal{D}_R(E) \subseteq \mathit{TrustSet}(P).$$

We restrict our attention to legitimate trust sets because given commercial product constraints (a party could not open and examine a coprocessor without destroying it), it would be difficult for a relying party who did not trust root to draw any useful conclusions.

3.5 Completeness

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 some $\mathsf{Chain}(E, K, H)$ 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 claiming key K, relying party P with any legitimate trust set, and history and run $H \prec R$:*

$$\mathcal{D}_R(E) \subseteq \mathit{TrustSet}(P) \Longrightarrow \mathit{Validate}(P, \mathit{Chain}(E, K, H)).$$

(Note that by our definition of $\mathsf{TrustSet}$, if $\mathcal{D}_R(E) \subseteq \mathsf{TrustSet}(P)$, then P believes that E will act honestly.)

3.6 Achieving both soundness and completeness

These definitions equip us to formalize a fundamental observation. If we're going build a trust-set authentication scheme that is both sound and complete, then the certificate chain for an entity needs to name its full dependency set. Figure 4 sketches why.

Theorem 1. *Suppose a trust-set OA scheme is both sound 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:*

$$\mathcal{D}_{R_1}(E) \neq \mathcal{D}_{R_2}(E) \Longrightarrow$$
$$\mathit{Chain}(E, K, H_1) \neq \mathit{Chain}(E, K, H_2).$$

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 sound 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

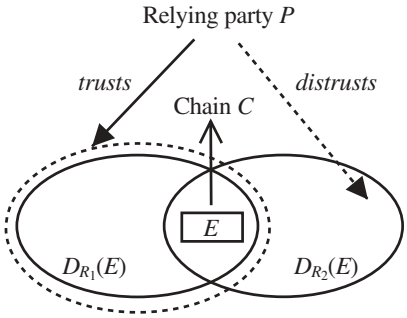

Fig. 4. If E produces the same chain but may or may not depend on things that P does not trust, then P must accept a chain it should reject, or reject a chain it should accept

scenarios or reject them in both scenarios. If party P accepts in run R_2, then the scheme cannot be sound 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.

3.7 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 – the relying party P makes its conclusion by recursively verifying signatures and applying a basic inference rule – in a Maurer-style calculus [14]. Suppose C is a set of *certificates*: statements of the form K_1 says $\mathsf{own}(E_2, K_2)$. Let S be the set of entities that P *trusts to speak the truth about assertions of key ownership*. That is:

$$S = \mathsf{TrustSet}(P).$$

We can then start reasoning about $\mathsf{View}_{\mathsf{will}}(C, S)$: the set of certificates statements and key ownership conclusions that this party will conclude are true, given the set of entities the party trusts.

A relying party may start by believing

$$C \cup \{\mathsf{own}(\mathsf{root}, K_{\mathsf{root}})\}.$$

So we initialize $\mathsf{View}_{\mathsf{will}}(C, S)$ to that set of statements. We then keep adding statements derivable from this set by applying the rule

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

Informally, if the party trusts an entity and believes that entity owns a key, then it believes certificates that key signs. The 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 $\mathsf{View}_{\mathsf{should}}(C, S, \mathcal{D})$ to be the set of statements derivable by applying the alternate rule:

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

Informally, if the party trusts an entity, and the entity (in this run) is in a configuration environment that the party trusts, then the party should believe certificates signed with that entity's key.

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

By construction of the inference rules, we can see that containment holds the other way.

$$\mathsf{View}_{\mathsf{should}}(\mathsf{Chain}(E, K, H), S, \mathcal{D}_R) \subseteq \mathsf{View}_{\mathsf{will}}(\mathsf{Chain}(E, K, H), S).$$

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 soundness and completeness, we need to make sure that, at each step in the chain, we maintain the invariant that 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. In particular, we follow two guidelines:

- Use the software and hardware architecture to *eliminate* any unnecessary dependence, and then
- Ensure that the dependency set that remains *participates* in certification.

4.1 Layer separation

Because of the postboot execution sequence, code that executes earlier can subvert code that executes later.[3] 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 execution 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

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

for epoch and configuration lifetimes, for Layer 2 and Layer 3. The initial boot-time cleanup code Layer 1 (already in the dependency set) zeroizes the appropriate regions on the appropriate transition. That is, if this boot sequence does not preserve the Layer K epoch, the BBRAM region for the Layer K epoch is zeroized; if this boot sequence does not preserve the Layer K configuration, the BBRAM region for the Layer K configuration is zeroized.

(For transitions to a new Layer 1, the cleanup is enforced by the *old* Layer 1 and the permanent Layer 0 – to avoid incurring 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 is unavoidable that $A_2 \xrightarrow{code}_R A_1$ (because A_1 could have cheated and not installed the correct code). However, to avoid $A_1 \xrightarrow{data}_R A_2$, 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. Figure 5 illustrates this process.

This technique – which we implemented and shipped with the Model 1 devices in 1997 – differs from the concept of *forward security*[4] 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.*

That is, the device leaves the factory with a key pair owned by Layer 1 and a certificate[5] signed by the factory root that names the factory root and that binds the device's public key to that device (specified by model number, serial number, etc.) with that version of Layer 1 code (identity of the owner of this layer, name they gave to this code, revision number they gave to this code, SHA-1 hash of this code, when the current Layer 1 epoch started, when the current Layer 1 configuration started, etc.). Each such update of Layer 1 then adds a transition certificate, signed by the old version, that names both the old and new version of the Layer 1 code as well as the fact that a transition took place from the old to the new.

As a consequence, a single malicious version cannot hide its presence in the trust chain; for a coalition of malicious versions (and the rest honest), the trust chain will name at least one malicious entity. (Section 5 considers forward-secure signatures further.)

To summarize: we eliminate dependency on future loads by destroying the old private key; but the past loads (on which a given version depends) participate in the chain for that version.

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 runtime 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. This approach follows our guidelines: the ratchet locks (Sect. 4.1) ensure that Layer 1 cannot depend on the OA Manager; the OA Manager depends on Layer 1, but Layer 1 creates and is named in its chain.

When requesting a key pair, the application specifies whether it should live as long as that Layer 3 epoch or as that Layer 3 configuration. The OA Manager will indicate this in the certificate; in conspiracy with Layer 1, the manager will also enforce this lifetime, by using our special BBRAM regions to see that the private key is zeroized when the lifetime ends.

These certificates also indicate that said key pair belongs to an application, and they include a field chosen by the application. (A straightforward extension of our

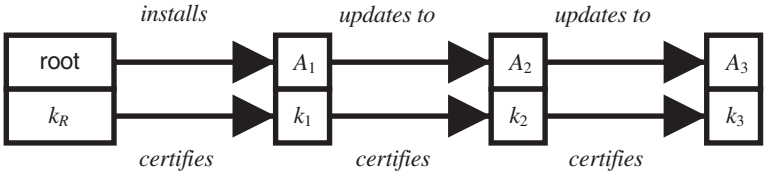

Fig. 5. When the code-loading layer updates itself, it generates and certifies a new key pair for its successor

[4] Ross Anderson's invited lecture at the ACM Conference on Computer and Communications Security in 1997, later documented as "Two Remarks on Public Key Cryptology," `http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/forwardsecure.pdf`, is popularly regarded as the seminal citation on forward-secure signatures; similar terms can be found in earlier work such as [11].

[5] For more information on naming details and examples of certificate formats, see pp. 3–95 to 3–116 in *IBM 4758 PCI Cryptographic Coprocessor Custom Software Interface Reference Version 2: 4758-002 and 4758-023.*

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.)

How long should the OA Manager key pair live? 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.

As discussed in Sect. 2.7, the interaction of certifier and certified lifetimes causes trouble.

If the OA Manager outlived the Layer 2 configuration, then our certification scheme *cannot be both sound and complete.* Figure 6 shows an example. Suppose Layer 2 is updated from B_1 to B_2 while preserving the OA key pair k_B. Application C_1 depends on the new version B_2, but its chain only names B_1. We violate Theorem 1; the scheme cannot be sound and complete.

If the OA Manager outlives the Layer 3 epoch, then we also have trouble. Figure 7 shows an example. Suppose application C_1 is replaced by application C_2 but the OA Manager retains the same key. If a relying party worries that C_2 may penetrate the OS, then C_1 may incur a dependency on C_2 – even though the C_1 chain does not name C_2.

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 Manager for the new Layer 3 configuration. If the new configuration was due to a Layer 1 reload, then the old Layer 1 signs a transition certificate which signs the new OA Manager key pair. 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. Figure 8 sketches this structure.

(As noted earlier, the private halves of any Layer 3 configuration key pairs will also be destroyed; if this configuration change does not preserve the Layer 3 epoch, those private keys are destroyed as well.)

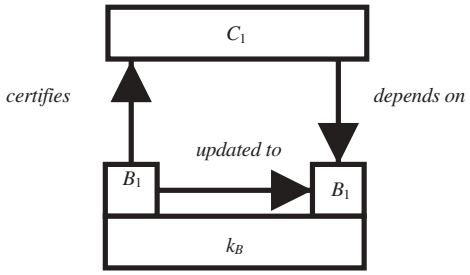

Fig. 6. If the certifier outlives its own code change, then the application can incur a dependency not named in its chain

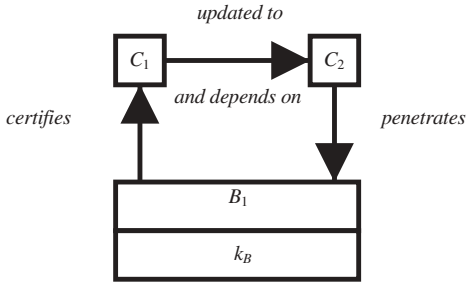

Fig. 7. If the certifier outlives the application, then the old application can incur a dependency not named in its chain

4.4 Naming

We already discussed the naming formats for the initial device certificate and the transition certificates. The OA Manager certificate names the Layer 1 certificate that signed it and this particular device and names the software entities (again, via identity of the owner of this layer, name they gave to this code, revision number they gave to this code, SHA-1 hash of this code, when the current layer epoch started, when the current layer configuration started, etc.) in both Layer 2 and Layer 3. An application certificate names the OA Manager certificate that signed it (which thus names the current Layer 3 epoch and the software present in this Layer 3 configuration), whether this certificate lives for an epoch or just a configuration, and the arbitrary data field given by the application.

Trusting an epoch entity requires, by definition, gambling that future secret-preserving code changes will be trustworthy. To make this more reasonable, we include code owner information (so that the relying party can know whose judgment they are trusting). To accommodate parties who chose to trust epochs to later change their minds, note that we also ensure that a Layer 3 epoch certificate (say, for epoch E) still names the Layer 3 configuration (say, C_1) in which it began existence. If, in some later Layer 3 configuration C_k within that same epoch, the relying party decides that it wants to examine the individual configurations to determine 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.

In a sense, a relying party exercising this "right of retroactive paranoia" begins with a trust set that treats configuration within an epoch in the same equivalence class, but then relaxes this assumption.

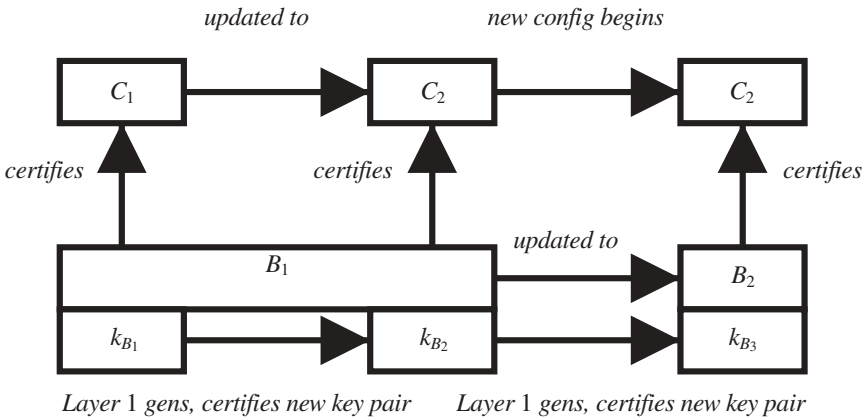

Fig. 8. We insure that chains name dependency by having Layer 1 generate a new OA Manager key pair with each change to code the application depends on

4.5 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 that created it.

Our design thus constitutes a trust-set scheme that is sound and complete for the dependency function we felt was appropriate, for any legitimate trust set. A certificate for an OA Manager key pair names exactly those configuration entities (including Layer 3, in case one does not trust the OS protections) that correct use of the Manager's private key depends on. A certificate for a configuration-length application key pair names exactly those configuration entities it depends on.

A certificate for an epoch-length application key pair names exactly those epoch entities it depends on; should the relying party later decide to not trust a particular Layer 3 configuration, a method exists, as sketched above, to shift to a configuration entity and determine if the untrusted configuration was present.

4.6 Implementation

Full support for OA shipped with all Model 2 family devices and the CP/Q++ embedded operating system. 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.

Implementation required some additional design decisions. To accommodate small developers (Sect. 2.3), 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. We also gave different families of cards different factory roots, to encourage relying parties to make a conscious decision about the root they choose.

A few aspects of the implementation proved challenging. 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 we decide 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 as "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 [13] and rights management boxes [12], that can be authenticated by anyone on the Internet, and can participate in PKI-based protocols.

We briefly 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 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. For example, supporting multiple applications would seem to require accepting at least one of the following scenarios:

- Forcing the relying party to trust OS protections,
- Awkwardly regenerating key pairs with each application change,
- Placing "potential peer applications" in groups that form minimum granularity for relying party trust sets.

It would also be interesting to explore OA in the context of software entities that may migrate between devices or be backed up and restored.

Alternate platforms. In our work (undertaken primarily from 1996 to 1999), we focused on how to authenticate software that lived in the most remote place possible: on the other side of the Internet, inside a box that (in theory and, one hopes, in practice) irrevocably destroys state if opened up.

Since our work began, industrial efforts have begun examining a similar problem: how to authenticate software living on an ordinary desktop on the other side of the Internet. These new "trusted computing platforms" and their applications in DRM (and implications for the desktop environment) comprise a volatile and sometimes controversial topic. As of this writing, the main efforts consist of the *Trusted Computing Platform Alliance (TCPA)* [19, 26] initiative [which has now changed its name to the *Trusted Computing Group* (TCG)], Microsoft's *Next Generation Secure Computing Base* (NG-SCB) [10], and Intel's *LaGrande* project [24]. The TCPA/TCG approach puts a smart-card-like *Trusted Platform Module (TPM)* on the motherboard; this TPM then releases credentials based on whether hashes measuring system-relevant parameters, such as the OS code, are correct. The NGSCB/LaGrande approaches appear to extend this into OS and CPU structure (although the full details are not yet public).

Academia has also been looking at how to augment system security using small amounts of physical security. The *AEGIS* [25], *CERIUM* [6], and *XOM* [15] projects all take various approaches to augmenting a CPU to protect processes; McGregor and Lee [18] explore augmenting a CPU to protect key material. In our own lab, we have been exploring [16] using commercially available TCPA/TCG hardware to attempt to build a "virtual" 4758.

The emergence of these commercial platforms has refocused attention on issues of how to authenticate a software entity on a remote platform (termed *attestation*), and how to bind data to that entity. It would be interesting to explore the interaction of our OA work with these ideas as well as the longer history of work in securely booting desktops [1, 7, 27].

Alternate PKI. The analysis and design presented in this paper made some implicit assumptions about what entities may certify, and about how relying parties draw conclusions. For example:

- Must a relying party accept the root? What if we had different roots?
- Must on-card entities that generate and certify key pairs limit themselves to certifying just the key pair? If the entity included some way of verifying "trustworthiness" of new code loads – apart from the fact that the right party authorized the load – then the relying party might be able increase its trust set over time, possibly changing our results.
- Must certification happen only when a key pair is created? Long-lived entities with the potential for runtime corruption suggest ongoing integrity-checking techniques.

It would be interesting to examine OA in light of such techniques.

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., [3, Sect. 3.3]). It would be interesting to investigate 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).

Trust sets in the wild. 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.

Acknowledgements. 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 design notes into manuals and customer training material. The author is also grateful for the comments and advice of the Internet2 PKI Labs on new research issues here. Finally, gratitude is due the anonymous referees, who have made this a stronger and clearer paper.

Preliminary versions of this paper appeared as Dartmouth College Technical Report TR2001-401 and in the proceedings of *ESORICS 2002*.

This paper reports work the author did while a research staff member at IBM Watson. Subsequent work was supported in part by the Mellon Foundation, AT&T/Internet2, the U.S. Department of Justice (contract 2000-DT-CX-K001), and NSF. The views and conclusions do not necessarily reflect the sponsors.

References

1. Arbaugh W, Farber D, Smith J (1997) A secure and reliable bootstrap architecture. In: Proceedings of the IEEE symposium on security and privacy. IEEE Press, New York, pp 65–71
2. Asonov D, Freytag J (2003) Almost optimal private information retrieval. In: Dingledine R, Syverson P (eds) Privacy enhancing technologies. Lecture notes in computer science, vol 2482. Springer, Berlin Heidelberg New York, pp 209–223
3. Bellare M, Miner S (1999) A forward-secure digital signature scheme. In: Wiener M (ed) Proceedings of Advances in Cryptology – Crypto 99. Lecture notes in computer science, vol 1666. Springer, Berlin Heidelberg New York, pp 431–448
4. Bond M, Anderson R (2001) API-level attacks on embedded systems. IEEE Comput 34:64–75
5. Chaum D, Pedersen T (1993) Wallet databases with observers. In: Brickell E (ed) Proceedings of Advances in Cryptology – Crypto '92. Lecture notes in computer science, vol 740. Springer, Berlin Heidelberg New York, pp 89–105
6. Chen B, Morris R (2003) Certifying program execution with secure processors. In: Proceedings of the 9th conference on hot topics in operating systems (HOTOS-IX). USENIX
7. Clark P, Hoffmann L (1994) Bits: A smartcard protected operating system. Commun ACM 37:66–70
8. Dyer J, Lindemann M, Perez R, Sailer R, Smith SW, van Doorn L, Weingart S (2001) Building the IBM 4758 secure coprocessor. IEEE Comput 34(October):57–66
9. Dyer J, Perez R, Smith SW, Lindemann M (1999) Application support architecture for a high-performance, programmable secure coprocessor. In: Proceedings of the 22nd National Information Systems Security conference
10. England P, Lampson B, Manferdelli J, Peinado M, Willman B (2003) A trusted open platform. IEEE Comput 36:55–62
11. Günther, CG (1990) An identity-based key-exchange protocol. In: Quiswater J-J, Vandewalle J (eds) Proceedings of Advances in Cryptology – Eurocrypt '89. Lecture notes in computer science, vol 434. Springer, Berlin Heidelberg New York, pp 29–37
12. Iliev A, Smith SW (2003) Prototyping an armored data vault: rights management for big brother's computer. In: Dingledine R, Syverson P (eds) Privacy enhancing technologies. Lecture notes in computer science, vol 2482. Springer, Berlin Heidelberg New York, pp 144–159
13. Jiang S, Smith SW, Minami K (2001) Securing web servers against insider attack. In: Proceedings of the 17th annual computer security applications conference. IEEE Press, New York, pp 265–276
14. Kohlas R, Maurer U (2000) Reasoning about public-key certification: on bindings between entities and public keys. J Select Areas Commun 18:551–560
15. Lie D, Thekkath C, Mitchell M, Lincoln P, Boneh D, Mitchell J, Horowitz M (2000) Architectural Support for Copy and Tamper Resistant Software. In: Proceedings of the 9th international conference on architectural support for programming languages and operating systems. ACM Press, New York, pp 168–177
16. Marchesini J, Smith SW, Wild O, MacDonald R (2003) Experimenting with TCPA/TCG hardware, or: How I learned to stop worrying and love the bear. Computer Science Technical Report TR2003-476, Dartmouth College
17. Maurer U (1996) Modelling a public-key infrastructure. In: Bertino E, Kurth H, Martella G, Montolivo E (eds) Proceedings of Computer Security – ESORICS 96. Lecture notes in computer science, vol 1146. Springer, Berlin Heidelberg New York
18. McGregor P, Lee R (2003) Virtual secure co-processing on general-purpose processors, Computer Engineering Technical Report CE-L2002-003, Princeton
19. Pearson S (ed) (2003) Trusted computing platforms: TCPA technology in context. Prentice-Hall, Upper Saddle River, NJ
20. Smith SW (1996) Secure coprocessing applications and research issues. Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory
21. Smith SW, Safford D (2001) Practical server privacy using secure coprocessors. IBM Sys J 40:683–695
22. Smith SW, Palmer E, Weingart S (1998) Using a high-performance, programmable secure coprocessor. In: Hirschfeld R (ed) Financial cryptography. Lecture notes in computer science, vol 1465. Springer, Berlin Heidelberg New York, pp 73–89
23. Smith SW, Weingart S (1999) Building a high-performance, programmable secure coprocessor. Comput Netw 31:831–860
24. Stam N (2003), Inside Intel's secretive 'LaGrande' project. `http://www.extremetech.com/`
25. Suh G, Clarke D, Gassend B, van Dijk M, Devadas S (2003) AEGIS: architecture for tamper-evident and tamper-resistant processing. In: Proceedings of the 17th international conference on supercomputing. ACM Press, New York, pp 160–171

26. Trusted Computing Platform Alliance (2001) TCPA design philosophies and concepts, version 1.0
27. Tygar JD, Yee BS (1993) Dyad: a system for using physically secure coprocessors. In: Proceedings of the joint Harvard-MIT workshop on technological strategies for the protection of intellectual property in the network multimedia environment
28. Yee BS (1994) Using Secure Coprocessors. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1994. Also available as Computer Science Technical Report CMU-CS-94-149
29. Yee BS (1997) A sanctuary for mobile agents. Computer Science Technical Report CS97-537, UCSD
30. Yee BS, Tygar JD (1995) Secure coprocessors in electronic commerce applications. In: Proceedings of the 1st USENIX Electronic Commerce workshop. USENIX, pp 155–170

S.W. Smith is currently on the faculty of the Department of Computer Science at Dartmouth College. His current research focuses on how to build trustable systems in the real world. He previously worked as a scientist at IBM T.J. Watson Research Center doing secure coprocessor design, implementation, and validation and at Los Alamos National Laboratory doing security designs and analyses for a wide range of public-sector clients. Dr. Smith was educated at Princeton (B.A., mathematics) and CMU (M.S., Ph.D., computer science) and is a member of ACM, USENIX, the IEEE Computer Society, Phi Beta Kappa, and Sigma Xi.