DRAFT

Building a High-Performance, Programmable Secure Coprocessor

Sean W. Smith Steve Weingart
Secure Systems and Smart Cards
IBM T.J. Watson Research Center
P.O Box 704, Yorktown Heights NY 10598 USA
{sean, c1shw}@watson.ibm.com

Revision of October 16, 1998

Abstract

Secure coprocessors enable secure distributed applications by providing safe havens where an application program can execute (and accumulate state), free of observation and interference by an adversary with direct physical access to the device. However, for these coprocessors to be effective, participants in such applications must be able to verify that they are interacting with an authentic program on an authentic, untampered device. Furthermore, secure coprocessors that support *general-purpose* computation and will be manufactured and distributed as *commercial products* must provide these core sanctuary and authentication properties while also meeting many additional challenges, including:

- the applications, operating system, and underlying security management may all come from different, mutually suspicious authorities;
- configuration and maintenance must occur in a hostile environment, while minimizing disruption of operations;
- the device must be able to recover from the vulnerabilities that inevitably emerge in complex software;
- physical security dictates that the device itself can never be opened and examined; and
- ever-evolving cryptographic requirements dictate that hardware accelerators be supported by reloadable on-card software.

This paper summarizes the hardware, software, and cryptographic architecture we developed to address these problems. Furthermore, with our colleagues, we have implemented this solution, into a commercially available product.

Contents

1. Introduction

1.1. Secure Coprocessors

Many current and proposed distributed applications face a fundamental security contradiction:

- computation must occur in remote devices,
- but these devices are vulnerable to physical attack by adversaries who would benefit from subverting this computation.

If an adversary can attack a device by altering or copying its algorithms or stored data, he or she often can subvert an entire application. The mere potential of such attack may suffice to make a new application too risky to consider.

Secure coprocessors—computational devices that can be trusted to execute their software correctly, despite physical attack—address these threats. Distributing such trusted sanctuaries throughout a hostile environment enables secure distributed applications.

Higher-end examples of secure coprocessing technology usually incorporate support for high-performance cryptography (and, indeed, the need to physically protect the secrets used in a cryptographic module motivated *FIPS 140-1*, the U.S. Government standard [11, 14] used for secure coprocessors). For over fifteen years, our team has explored building high-end devices: robust, general-purpose computational environments inside secure tamper-responsive physical packages [15, 22, 23, 24]. This work led to the Abyss, μAbyss, and Citadel prototypes, and contributed to the physical security design for some of earlier IBM cryptographic accelerators.

However, although our efforts have focused on high-end coprocessors, devices that accept much more limited computational power and physical security in exchange for a vast decrease in cost—such as IC chip cards, PCMCIA tokens, and "smart buttons"—might also be considered part of the secure coprocessing family. (As of this writing, no device has been certified to the tamper-response criteria of Level 4 of the FIPS 140-1 standard; even the touted Fortezza achieved only Level 2, tamper evident.)

Even though this technology is closely associated with cryptographic accelerators, much of the exciting potential of the secure coprocessing model arises from the notion of putting *computation* as well as cryptographic secrets inside the secure box. Yee's seminal examination of this model [26] built on our Citadel prototype. Follow-up research by Tygar and Yee [21, 27] and others (e.g., [9, 13, 17]) further explores the potential applications and limits of the secure coprocessing model.

1.2. The Challenge

This research introduced the challenge: how do we make this vision real? Widespread development and practical deployment of secure coprocessing applications requires an infrastructure of secure devices, not just a handful of laboratory prototypes. Recognizing this need, our team has recently completed a several-year research and development project to design, develop, and distribute such a device—both as a research tool and as a commercial product, which reached market August 1997.

This project challenged us with several constraints: [20]

- the device must offer *high-performance* computational and cryptographic resources;
- the device must be easily *programmable* by IBM and non-IBM developers, even in small quantities;
- the device must exist within the manufacturing, distribution, maintenance, and trust confines of a *commercial product* (as opposed to an academic prototype) from a private vendor.

However, the projected lifecycle of a high-end secure coprocessor challenged us with security issues:

- How does a generic commercial device end up in a hostile environment, with the proper software and secrets?
- How do participants in distributed applications distinguish between a properly configured, untampered device, and a maliciously modified one or a clone?

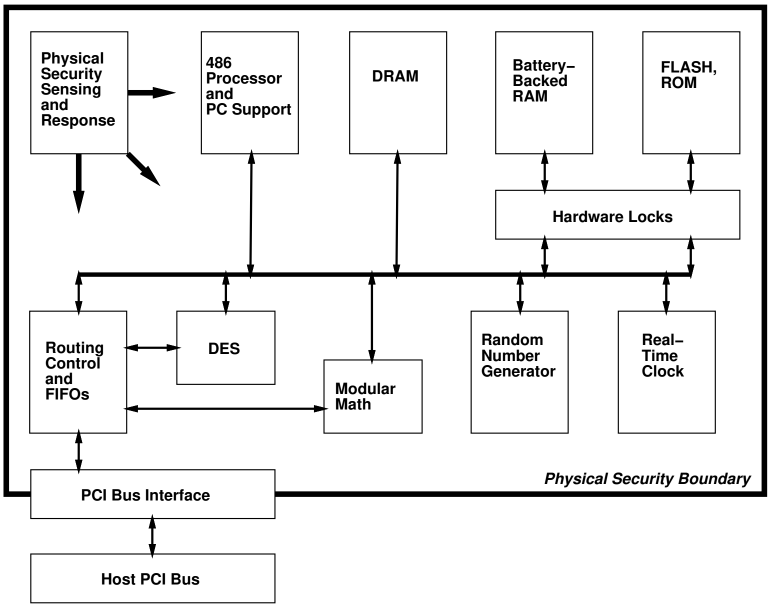

Figure 1 Hardware architecture of our high-end secure coprocessor.

1.3. Overview of our Technology

Hardware Design For our product [12], we answered these questions by building on the design philosophy that evolved in our previous prototypes:

- maximize computational power (e.g., use as big a CPU as is reasonable, and use good cryptographic accelerators[1]);
- support it with ample *dynamic RAM (DRAM)*;
- use a smaller amount of *battery-backed RAM (BBRAM)* as the non-volatile, secure memory; and
- assemble this on a circuit board with technology to actively sense tamper and near-instantly *zeroize* the BBRAM.

Figure 1 sketches this design.

Security Design Active tamper response gives a device a lifecycle shown in Figure 2: tamper destroys the contents of secure memory—in our case, the BBRAM and DRAM. However, one can logically extend the secure storage area beyond the BBRAM devices themselves by storing keys in BBRAM and ciphertext in FLASH,[2] or even *cryptopaging* [26] it onto the host file system.

[1] Our current hardware features a 66 MHz 486 CPU, and accelerators for DES, modular math (hence RSA and DSA), and noise-based random number generation.

[2] FLASH is a non-volatile memory technology similar to EEPROM. FLASH differs significantly from the more familiar RAM model in several ways: FLASH can only be reprogrammed by erasing and then writing an entire *sector* first; the erase and rewrite cycles take significantly longer than RAM; and FLASH imposes a finite lifetime (currently, usually 10^4 or 10^5) on the maximum number of erase/rewrite cycles for any one sector.

Application Design This design leads to a notion of a *high-end* secure coprocessor that is substantially more powerful and secure—albeit larger[3] and more expensive—than the family's weaker members, such as chip cards. (The larger physical package and higher cost permit more extensive protections.) This approach shapes the design for application software:

- protect the critical portion of the application software by having it execute inside the secure coprocessor;
- allow this critical portion to be fairly complex;
- then structure this critical software to exploit the tamper protections: tamper destroys only contents of volatile DRAM and the smaller BBRAM—but not, for example, the contents of FLASH or ROM.

Software Design Making a commercial product support this application design requires giving the device a robust programming environment, and making it easy for developers to use this environment—even if they do not necessarily trust IBM or each other. These goals led to a multi-layer software architecture:

- a foundational *Miniboot* layer manages security and configuration;
- an *operating system* or *control program* layer manages computational, storage, and cryptographic resources; and
- an unprivileged *application* layer that uses these resources to provide services

Figure 3 sketches this architecture.

Currently, Miniboot consists of two components: *Miniboot 0*, residing in ROM, and *Miniboot 1*, which resides, like the OS and the application, in rewritable non-volatile FLASH memory. However, we are also considering adding support for various multi-application scenarios, including the simultaneous existence of two or more potentially malicious applications in the same device, as well as one "master" application that dynamically loads additional applications at run-time.

1.4. This Paper

Building a high-performance, programmable secure coprocessor as a mass-produced product—and not just as a laboratory prototype—requires identifying, articulating, and addressing a host of research issues regarding security and trust. This paper discusses the security architecture we designed and (with our colleagues) implemented.

- Section 2 presents the security goals and commercial constraints we faced.
- Section 3 introduces our approach to solving them.
- Section 4 through Section 8 presents the different interlocking pieces of our solution.
- Section 9 and Section 10 summarize how these pieces work together to satisfy the security goals.

Section 11 presents some thoughts for future directions.

[3] For example, our product is a PCI card, although we see no substantial engineering barriers to repackaging this technology as a PCMCIA card.

DRAFT

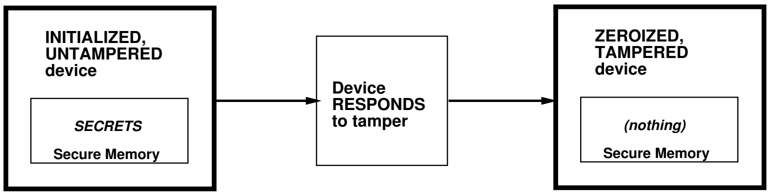

Figure 2 Sample lifecycle of a high-end secure coprocessor with active tamper response.

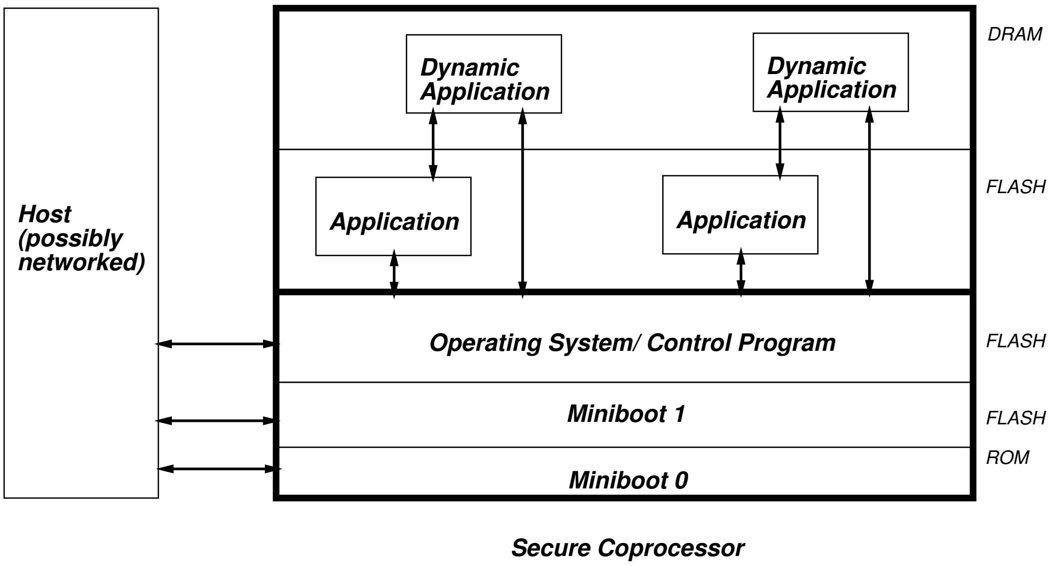

Figure 3 Software architecture for our high-end secure coprocessor. Our current software only supports one application, not dynamically loaded.

2. Requirements

In order to be effective, our solution must simultaneously fulfill two different sets of requirements. The device must provide the core security properties necessary for secure coprocessing applications. But the device must also be a practical, commercial product; this goal gives rise to many additional constraints, which can interact with the security properties in subtle ways.

2.1. Commercial Requirements

Our device must exist as a programmable, general-purpose computer—because the fundamental secure coprocessing model (e.g.,[26, 27]) requires that *computation*, not just cryptography, reside inside the secure box. This notion—and previous experience with commercial security hardware (e.g., [1])—gives rise to many constraints.

Development. To begin with, the goal of supporting the widespread development and deployment of applications implies:

- The device must be easily programmable.
- The device must have a general-purpose operating system, with debugging support (when appropriate).
- The device must support a large population of authorities developing and releasing application and OS code, deployed in various combinations on different instantiations of the same basic device.
- The device must support vertical partitioning: an application from one vendor, an OS from another, bootstrap code from a third.
- These vendors may not necessarily trust each other—hence, the architecture should permit no "backdoors."

Manufacturing. The process of manufacturing and distribution must be as simple as possible:

- We need to minimize the number of variations of the device, as manufactured or shipped (since each new variation dramatically increases administrative overhead).
- It must be possible to configure the software on the device after shipment, in what we must regard as a hostile environment.
- We must reduce or eliminate the need to store a large database of records (secret or otherwise) pertaining to individual devices.
- As an international corporation based in the United States, we must abide by U.S. export regulations.

Maintenance. The complexity of the proposed software—and the cost of a high-end device—mean that it must be possible to update the software *already installed* in a device.

- These updates should be safe, easy, and minimize disruption of device operation.
  - When possible, the updates should be performed remotely, in the "hostile" field, without requiring the presence of a trusted security officer.
  - When reasonable, internal application state should persist across updates.
- Particular versions of software may be so defective as to be non-functional or downright malicious. Safe, easy updates must be possible even then.

- Due to its complexity and ever-evolving nature, the code supporting high-end cryptography (including public-key[4], hashing, and randomness) must itself be updatable. But repair should be possible even if this software is non-functional.

2.2. Security Requirements

The primary value of a secure coprocessor is its ability to provide a trusted sanctuary in a hostile environment. This goal leads to two core security requirements:

- The device must really provide a safe haven for application software to execute and accumulate secrets.
- It must be possible to remotely distinguish between a message from a genuine application on an untampered device, and a message from a clever adversary.

We consider these requirements in turn.

2.2.1. Safe Execution

It must be possible for the card, placed in a hostile environment, to distinguish between genuine software updates from the appropriate trusted sources, and attacks from a clever adversary.

The foundation of secure coprocessing applications is that the coprocessor really provides safe haven. For example, suppose that we are implementing decentralized electronic cash by having two secure devices shake hands and then transactionally exchange money (e.g., [27]). Such a cash program may store two critical parameters in BBRAM: the private key of this wallet, and the current balance of this wallet. Minimally, it must be the case that physical attack really destroys the private key. However, it must *also* be the case that the stored balance never change except through appropriate action of the cash program. (For example, the balance should *not* change due to defective memory management or lack of fault-tolerance in updates.)

Formalizing this requirement brings out many subtleties, especially in light of the flexible shipment, loading, and update scenarios required by Section 2.1 above. For example:

- What if an adversary physically modifies the device before the cash program was installed?
- What if an adversary "updates" the cash program with a malicious version?
- What if an adversary updates the operating system underneath the cash program with a malicious version?
- What if the adversary already updated the operating system with a malicious version *before* the cash program was installed?
- What if the adversary replaced the public-key cryptography code with one that provides backdoors?
- What if a sibling application finds and exploits a flaw in the protections provided by the underlying operating system?

After much consideration, we developed *safe execution* criteria that address the *authority* in charge of a particular software layer, and the *execution environment*—the code and hardware—that has accesses to the secrets belonging to that layer.

- **Control of software.** If Authority N has ownership of a particular software layer in a particular device, then only Authority N, or a designated superior, can load code into that layer in that device.
- **Access to secrets.** The secrets belonging to this layer are accessible only by code that Authority N trusts, executing on hardware that Authority N trusts, in the appropriate context.

[4] Our hardware accelerator for RSA and DSA merely does modular arithmetic; hence, additional software support is necessary.

2.2.2. Authenticated Execution

Providing a safe haven for code to run does not do much good, if it is not possible to distinguish this safe haven from an impostor. It must thus be possible to:

- authenticate an *untampered device*;
- authenticate its *software configuration*; and
- do this *remotely*, via computational means.

The first requirement is the most natural. Consider again example of decentralized cash. An adversary who runs this application on an exposed computer but convinces the world it is really running on a secure device has compromised the entire cash system—since he or she can freely counterfeit money by incrementing the stored balance.

The second requirement—authenticating the software configuration—is often overlooked but equally important. In the cash example, running a *maliciously modified* wallet application on a *secure device* also gives an adversary the ability to counterfeit money. For another example, running a Certificate Authority on a physically secure machine without knowing for certain what key generation software is really installed leaves one open to attack [28].

The third requirement—remote verification—is driven by two main concerns. First, in the most general distributed application scenarios, participants may be separated by great physical distance, and have no trusted witnesses at each other's site. Physical inspection is not possible, and even the strongest tamper-evidence technology is not effective without a good audit procedure.

Furthermore, we are reluctant to trust the effectiveness of commercially feasible *tamper-evidence* technology against the dedicated adversaries that might target a high-end device. (Tamper-evidence technology only attempts to ensure that tampering leaves clear visual signs.) We are afraid that a device that is opened, modified and reassembled may *appear* perfect enough to fool even trained analysts.

This potential for perfect reassembly raises the serious possibility of attack during distribution and configuration. In many deployment scenarios, no one will have both the skills and the motivation to detect physical tamper. The user who takes the device out of its shipping carton will probably not have the ability to carry out the forensic *physical* analysis necessary to detect a sophisticated attack with high assurance. Furthermore, the users may *be* the adversary—who probably should not be trusted to report whether or not his or her device shows signs of the physical attack he or she just attempted. Those parties (such as, perhaps, the manufacturer) with both the skills and the motivation to detect tamper may be reluctant to accept the potential liability of a "false negative" tamper evaluation.

For all these reasons, our tamper-protection approach does not rely on tamper-evidence alone—see Section 4.

3. Overview of Our Architecture

In order to meet the requirements of Section 2, our architecture must ensure secure loading and execution of code, while also accommodating the flexibility and trust scenarios dictated by commercial constraints.

3.1. Secrets

Discussions of secure coprocessor technology usually begin with "physical attack zeroizes secrets." Our security architecture must begin by ensuring that tamper actually destroys secrets that actually meant something. We do this with three main techniques:

- **The secrets go away with physical attack.** Section 4 presents our tamper-detection circuitry and protocol techniques. These ensure that physical attack results in the actual zeroization of sensitive memory.
- **The secrets started out secret.** Section 5 presents our factory initialization and regeneration/recertification protocols. These ensure that the secrets, when first established, were neither known nor predictable outside the card, and do not require assumptions of indefinite security of any given keypair.
- **The secrets stayed secret despite software attack.** Section 6 presents our hardware ratchet lock techniques. These techniques ensure that, despite arbitrarily bad compromise of rewritable software, sufficiently many secrets remain to enable recovery of the device.

3.2. Code

Second, we must ensure that code is loaded and updated in a safe way. Discussions of code-downloading usually begin with "just sign the code." However, focusing on code-signing alone neglects several additional subtleties that this security architecture must address. Further complications arise from the commercial requirement that this architecture accommodate a pool of mutually suspicious developers, who produce code that is loaded and updated in the hostile field, with no trusted couriers.

- **Code loads and updates.** We must have techniques that address questions such as:
  - What about updates to the code that checks the signatures for updates?
  - Against whose public key should we check the signature?
  - Does code end up installed in the correct place?
  - What happens when another authority updates a layer on which one's code depends?
- **Code integrity.** For the code loading techniques to be effective, we must also address issues such as:
  - What about the integrity of the code that checks the signature?
  - Can adversarial code rewrite other layers?

Section 7 presents our techniques for code integrity, and Section 8 presents our protocols for code loading. Together, these ensure that the code in a layer is changed and executed only in an environment trusted by the appropriate code authority.

3.3. Achieving the Security Requirements

Our full architecture carefully combines the building blocks described in Section 4 through Section 8 to achieve the required security properties:

- **Code executes in a secure environment.** Section 9 presents how our secrecy management and code integrity techniques interact to achieve the requirements of Section 2.2.1: software loaded onto the card can execute and accumulate state in a continuously trusted environment, despite the risks introduced by dependency on underlying software controlled by a potentially hostile authority.
- **Participants can remotely authenticate real code on a real device.** Section 10 presents how our secrecy management and code integrity techniques interact to achieve the requirement of Section 2.2.2: any third party can distinguish between a message from a particular program in a particular configuration of an untampered device, and a message from a clever adversary.

4. Defending against Physical Threats

The main goal of physical security is to ensure that the hardware can know if it remains in an unmolested state—and if so, that it continues to work in the way it was intended to work. To achieve physical security, we start with our basic computational/crypto device and add additional circuitry and components to detect tampering by *direct physical penetration* or by *unusual operating conditions*. If the circuit a condition that would compromise correct operation, the circuit responds in a manner to prevent theft of secrets or misuse of the secure coprocessor.

4.1. Overview

Traditionally, physical security design has taken several approaches:

- *tamper evidence*, where packaging forces tamper to leave indelible physical changes;
- *tamper resistance*, where the device packaging makes tamper difficult;
- *tamper detection*, where the device actually is aware of tamper; and
- *tamper response*, where the device actively takes countermeasures upon tamper.

We feel that commercially feasible tamper-evidence technology and tamper-resistance technology cannot withstand the dedicated attacks that a high-performance, multi-chip coprocessor might face. Consequently, our design incorporates an interleaving of resistance and detection/response techniques, so that penetrations are sufficiently difficult to trigger device response.

- Section 4.2 will discuss our techniques to detect penetration attacks.
- Section 4.3 will discuss how our device responds once tamper is detected.
- Section 4.4 will discusses the additional steps we take to ensure that tamper response is effective and meaningful—particularly against physical attacks other than penetration.

Historically, work in this area placed the largest effort on physical penetration [8, 22, 23]. Preventing an adversary from penetrating the secure coprocessor and probing the circuit to discover the contained secrets is still the first step in a physical security design. Although some standards have emerged as groundwork and guidelines [14, 24, 25], exact techniques are still evolving.

A significant amount of recent work examines efforts to cause incorrect device operation, and thus allow bypass of security functions [2, 3]. Other recent work capitalizes on small induced failures in cryptographic algorithms to make discovery of keys easier [6, 7]. Consequently, as feasible tampering attacks have become more sophisticated through time and practice, it has become necessary to improve all aspects of a physical security system. Over the years many techniques have been developed, but they all face the same problem: *no provably tamper-proof system exists.* Designs get better and better, but so do the adversary's skill and tools. As a result, physical security is, and will remain, a race between the defender and the attacker. (To date, we have not been able to compromise our own security, which is also under evaluation by an independent laboratory against the FIPS 140-1 Level 4 criteria. [11, 14])

The economic challenge of producing a physically secure but usable system at a reasonable cost is difficult.

4.2. Detecting Penetration

We have taken the approach of making incremental improvements on well-known technology, and layering these techniques. This way, the attacker has to repeat, at each layer, work that has a low probability of success; furthermore, the attacker must work *through* the layers that have already been passed (and may still be active). The basic element is a grid of conductors which is monitored by circuitry that can detect changes in the properties (open, shorts, changes in conductivity) of the conductors. The conductors themselves are non-metallic and closely resemble the material in which they are embedded—making discovery, isolation, and manipulation more difficult. These grids are arranged in several layers and the sensing circuitry can detect accidental connection *between* layers as well as changes in an *individual* layer.

The sensing grids are made of flexible material and are wrapped around and attached to the secure coprocessor package as if it were being gift-wrapped. Connections to and from the secure coprocessor are made via a thin flexible cable which is brought out between the folds in the sensing grids so that no openings are left in the package. (Using a standard connector would leave such openings.)

After we wrap the package, we embed it in a potting material. As mentioned above, this material closely resembles the material of the conductors in the sensing grids. Besides making it harder to find the conductors, this physical and chemical resemblance makes it nearly impossible for an attacker to penetrate the potting without also affecting the conductors. Then we enclose the entire package in a grounded shield to reduce susceptibility to electromagnetic interference, and to reduce detectable electromagnetic emanations.

4.3. Responding to Tamper

The most natural tamper response in a secure coprocessor is to erase secrets that are contained in the unit, usually by erasing (*zeroizing*) an *Static Random Access Memory (SRAM)* that contains the secrets, then erasing the operating memory and ceasing operation. An SRAM can be made persistent with a small battery, and can, under many conditions, be easily erased.

This is what we do in our device: *battery-backed* SRAM (*BBRAM*) exists as storage for secrets. Upon detection of tamper, we zeroize the BBRAM and disable the rest of the device by holding it in reset. The tamper detection/response circuitry is *active at all times* whether the coprocessor is powered or not—the detection/response circuitry runs on the same battery that maintains the BBRAM when the unit is unpowered.

Tamper can happen quickly. In order to erase quickly, we *crowbar* the SRAM by switching its power connection to ground. At the same time, we force all data, address and control lines to a high impedance state, in order to prevent back-powering of the SRAM via those lines. This technique is employed because it is simple, effective, and it does not depend on the CPU being sufficiently operational for sufficiently long to overwrite the contents of the SRAM on tamper.

4.4. Detecting other Physical Attacks

To prevent attacks based on manipulating the operating conditions, including those that would make it difficult to respond to tamper and erase the secrets in SRAM, several additional sensors have been added to the security circuitry to detect and respond to changes in operating conditions.

Attacks on Zeroization. For zeroization to be effective, certain environmental conditions must be met. For example, low temperatures will allow an SRAM to retain its data even with the power connection shorted to ground. To prevent this, a temperature sensor in our device will cause the protection circuit to erase the SRAM if the temperature goes below a preset level.

Ionizing radiation will also cause an SRAM to retain its data, and may disrupt circuit operation. For this reason, our device also detects significant amounts of ionizing radiation and triggers the tamper response if detected.

Storing the same value in a bit in SRAM over long periods can also cause that value to imprint. Our software protocols take this threat into account, by periodically inverting this data.

Other Attacks. An adversary might also compromise security by causing incorrect operation through careful manipulation of various environmental parameters. As a consequence, a device needs to detect and defend against such attacks.

One such environmental parameter is supply voltage, which has to be monitored for several thresholds. For example, at each power-down, the voltage will go from an acceptable level to a low voltage, then to no supply voltage.

But the detection and response circuitry needs to be always active—so at some point, it has to switch over to battery operation. A symmetric transition occurs at power-up.

Whenever the voltage goes below the acceptable operating level of the CPU and its associated circuitry, these components are all held in a reset state until the voltage reaches the operating point. When the voltage reaches the operating point, the circuitry is allowed to run. If the voltage exceeds the specified upper limit for guaranteed correct operation, it is considered a tamper, and the tamper circuitry is activated.

Another method by which correct operation can be compromised is by manipulating the clock signals that go to the coprocessor. To defend against these sorts of problems, we use *phase locked loops* and independently generated internal clocks to prevent clock signals with missing or extra pulses, or ones that are either too fast or slow.

High temperatures can cause improper operation of the device CPU, and even damage it. So, high temperatures cause the device to be held in reset from the operational limit to the storage limit. Detection of temperature above the storage limit is treated as a tamper event.

5. Device Initialization

Section 4 discussed how we erase device secrets upon tamper. One might deduce that a natural consequence would be that "knowledge of secrets" implies "device is real and untampered." But for this conclusion to hold, we need more premises:

- the secrets were secret when they were first established;
- the device was real and untampered when its secrets were established;
- weakening of cryptography does not compromise the secrets;
- operation of the device has not caused the secrets to be exposed.

This section discusses how we provide the first three properties; Section 6 will discuss how we provide the fourth.

5.1. Factory Initialization

As one might naturally suspect, an untampered device authenticates itself as such using cryptographic secrets stored in secure memory. The primary secret is the private half of an RSA or DSA keypair. Section 10 elaborates on the use of this private key. Some symmetric-key secrets are also necessary for some special cases (as Section 5.2.3 and Section 8.3 will discuss).

The device keypair is generated at *device initialization*. To minimize risk of exposure, a device generates its own keypair internally, within the tamper-protected and using seeds produced from the internal hardware random number generator. The device holds its private key in secure BBRAM, but exports its public key. An external *Certification Authority (CA)* adds identifying information about the device and its software configuration, signs a certificate for this device, and returns the certificate to the device.

(The device-specific symmetric keys are also generated internally at initialization—see Section 8.3.)

Clearly, the CA must have some reason to believe that the device in question really is an authentic, untampered device. To address this question—and avoid the risks of undetectable physical modification (Section 4.1)—we initialize the cards in the factory, as the last step of manufacturing.

Although factory initialization removes the risks associated with insecure shipping and storage, it does introduce one substantial drawback: the device must remain within the safe storage temperature range (Section 4.4). But when considering the point of initialization, a manufacturer faces a tradeoff between ease of distribution and security; we have chosen security.

5.2. Field Operations

5.2.1. Regeneration

An initialized device has the ability to *regenerate* its keypair. Regeneration frees a device from depending forever on one keypair, or key length, or even cryptosystem. Performing regeneration *atomically*[5] with other actions, such as reloading the crypto code (Section 8), also proves useful (as Section 10 will discuss). For stronger forward integrity, implementations could combine this technique with expiration dates.

To regenerate its keypair, a device does the following:

- create a new keypair from internal randomness,

[5] An *atomic* change is one that happens *entirely*, or *not at all*—despite failures and interruptions. Atomicity for complex configuration-changing operations is in an important property.

- use the *old* private key to sign a *transition certificate* for the *new* public key, including data such as the reason for the change, and
- atomically complete the change, by deleting the old private key and making the new pair and certificate "official."

The current list of transition certificates, combined with the initial device certificate, certifies the current device private key. Figure 4 illustrates this process.

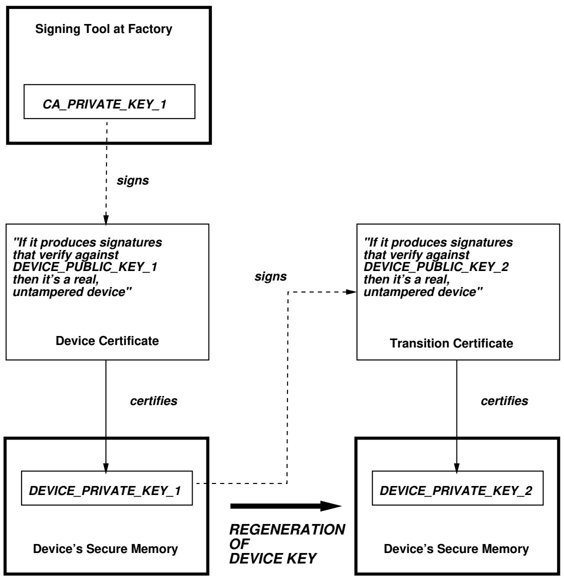

Figure 4 The device may regenerate its internal keypair, and atomically create a transition certificate for the new public key signed with the old private key.

5.2.2. Recertification

The CA for devices can also *recertify* the device, by atomically replacing the old certificate and (possibly empty) chain of transition certificates with a single new certificate. Figure 5 illustrates this process. (Clearly, it would be a good idea for the CA to verify that the claimed public key really is the current public key of an untampered device in the appropriate family.)

This technique can also frees the CA from depending forever on a single keypair, key length, or even cryptosystem. Figure 6 illustrates this variation. Again, for stronger forward integrity, implementations could combine this technique with expiration dates.

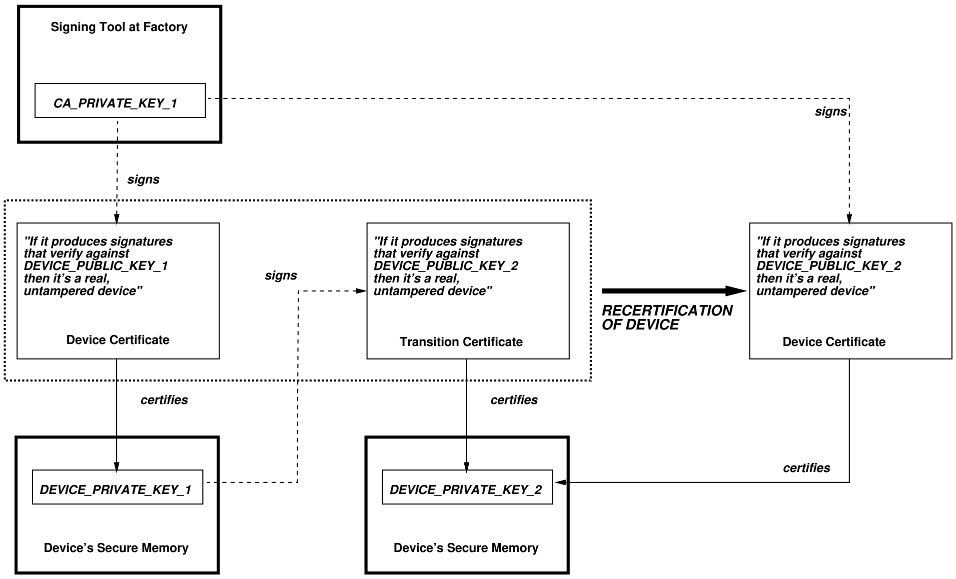

Figure 5 The CA can recertify a device, by replacing its current device certificate and transition certificate sequence with a new device certificate, certifying the latest public key.

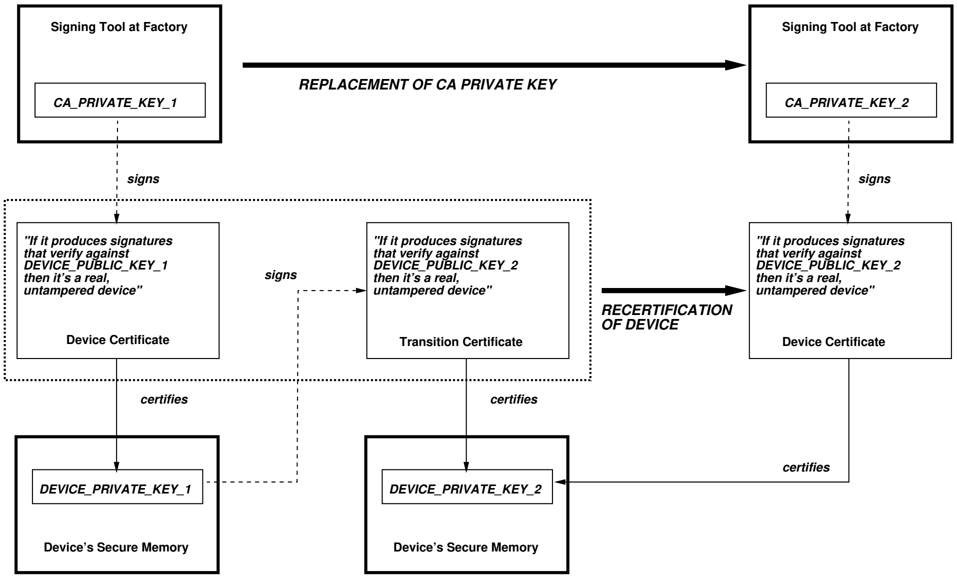

Figure 6 The CA can use device recertification in order to avoid depending forever on the same keypair.

5.2.3. Revival

Scenarios arise where the tamper detection circuitry in a device has zeroized its secrets, but the device is otherwise untampered. As Section 4 discusses, certain environmental changes—such as cold storage or bungled battery removal—trigger tamper response in our design, since otherwise these changes would provide an avenue for undetected tamper. Such scenarios are arguably inevitable in many tamper-response designs—since a device cannot easily wait to see if a tamper attempt is successful before responding.

Satisfying an initial commercial constraint of "save hardware whenever possible" requires a way of *reviving* such a zeroized but otherwise untampered device. However, such a revival procedure introduces a significant vulnerability: how do we distinguish between zeroized but untampered device, and a tampered device? Figure 7 illustrates this problem.

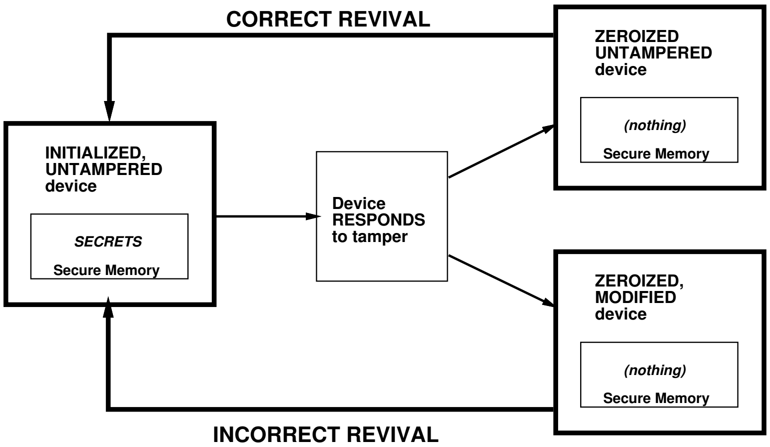

Figure 7 Tamper response zeroizes the secrets in an initialized device, and leaves either an untampered but zeroized device, or a tampered device. A procedure to revive a zeroized device must be able distinguish between the two, or else risk introducing tampered devices back into the pool of allegedly untampered ones.

How do we perform this authentication?

As discussed earlier, we cannot rely on physical evidence to determine whether a given card is untampered—since we fear that a dedicated, well-funded adversary could modify a device (e.g., by changing the internal FLASH components) and then re-assemble it sufficiently well that it passes direct physical inspection. Indeed, the need for factory-initialization was driven by this concern:

> *We can only rely on secrets in tamper-protected secure memory to distinguish a real device from a tampered device.*

Indeed, the problem is basically unsolvable—how can we distinguish an untampered but zeroized card from a tampered reconstruction, when, by definition, every aspect of the untampered card is visible to a dedicated adversary?

To accommodate both the commercial and security constraints, our architecture compromises:

- **Revival is Possible.** We provide a way for a trusted authority to revive an allegedly untampered but zeroized card, based on authentication via non-volatile, non-zeroizable "secrets" stored inside a particular device component.

Clearly, this technique is risky, since a dedicated adversary can obtain a device's revival secrets via destructive analysis of the device, and then build a fake device that can spoof the revival authority.

- **Revival is Safe.** To accommodate this risk, we force revival to atomically destroy all secrets within a device, and to leave it without a certified private key. A trusted CA must then re-initialize the device, before the device can "prove" itself genuine. This initialization requires the creation of a new device certificate, which provides the CA with an avenue to explicitly indicate the card has been revived (e.g., "if it produces signatures that verify against Device Public Key N, then it is allegedly a real, untampered device that has undergone revival—so beware").

  Thus, we prevent a device that has undergone this risky procedure from impersonating an untampered device that has never been zeroized and revived.

Furthermore, given the difficulty of effectively authenticating an untampered but zeroized card, and the potential risks of a mistake, the support team for the commercial product has decided not to support this option in practice.

5.3. Trusting the Manufacturer

A discussion of untamperedness leads to the question: why should the user trust the manufacturer of the device? Considering this question gives rise to three sets of issues.

- **Contents.** Does the black box really contain the advertised circuits and firmware? The paranoid user can verify this probabilistically by physically opening and examining a number of devices. (The necessary design criteria and object code listings could be made available to customers under special contract.)
- **CA Private Key.** Does the factory CA ever certify bogus devices? Such abuse is a risk with any public-key hierarchy. But, the paranoid user can always establish their own key hierarchy, and then design applications that accept as genuine only those devices with a secondary certificate from this alternate authority.
- **Initialization.** Was the device actually initialized in the advertised manner? Given the control a manufacturer might have, it is hard to see how we can conclusively establish that the initialization secrets in a card are indeed relics of the execution of the correct code. However, the cut-and-examine approach above can convince a paranoid user that the key creation and management software in an already initialized device is genuine. This assurance, coupled with the regeneration technique of Section 5.2.1 above, provides a solution for the paranoid user: causing their device to regenerate after shipment gives it a *new* private key that must have been produced in the advertised safe fashion.

6. Defending against Software Threats

6.1. Motivation

Section 4 discussed how we ensure that the core secrets are zeroized upon physical attack, and Section 5 discussed how we ensure that they were secret to begin with. However, these techniques still leave an exposure: did the device secrets remain secret throughout operation?

For example, suppose a few months after release, some penetration specialists discover a hole in the OS that allows untrusted user code to execute with full supervisor privilege. Our code loading protocol (Section 8) allows us to ship out a patch, and a device installing this patch can sign a receipt with its private key.

One might suspect verifying this signature would imply the hole has been patched in that device. Unfortunately, this conclusion would be wrong: a hole that allows untrusted code full privileges would also grant it access to the private key—that is, *without additional hardware countermeasures.* This section discusses the countermeasures we use.

6.2. Software Threat Model

This risk is particularly dire in light of the commercial constraints of multiple layers of complex software, from multiple authorities, remotely installed and updated in hostile environments. History shows that complex systems are, quite often, permeable. Consequently, we address this risk by assuming that *all rewritable software* in the device may behave *arbitrarily badly*.

Drawing our defense boundary here frees us from the quagmire[6] of having low-level miniboot code evaluate incoming code for safety. It also accommodates the wishes of system software designers who want full access to "Ring 0" in the underlying Intel x86 CPU architecture.

Declaring this assumption often raises objections from systems programmers. We pro-actively raise some counterarguments. First, although all code loaded into the device is somehow "controlled," we need to accommodate the pessimistic view that "controlled software" means, at best, good intentions. Second, although an OS might provide two levels of privilege, history[7] is full of examples where low-level programs usurp higher-level privileges. Finally, as implementers ourselves, we need to acknowledge the very real possibility of error and accommodate mistakes as well as malice.

6.3. Hardware Access Locks

In order to limit the abilities of rogue but privileged software, we use *hardware locks*: independent circuitry that restricts the activities of code executing on the main CPU. We chose to use a simple hardware approach for several reasons, including:

- We cannot rely on the device operating system, since we do not know what it will be—and a corrupt or faulty OS might be what we need to defend against.
- We cannot rely on the protection rings of the device CPU, because the rewritable OS and Miniboot layers require maximal CPU privilege.

Figure 1 shows how the hardware locks fit into the overall design: the locks are independent devices that can interact with the main CPU, but control access to the FLASH and to BBRAM.

[6] With the advent of Java, preventing hostile downloaded code from damaging a system has (again) become a popular topic. Our architecture responds to this challenge by allowing "applets" to do whatever they want—except they can neither access critical authentication secrets, nor alter critical code (which includes the code that can access these secrets). Furthermore, these restrictions are enforced by hardware, independent of the OS and CPU.

[7] For examples, consult the on-line archives of the Computer Emergency Response Team at Carnegie Mellon University.

However, this approach raises a problem. Critical memory needs protection from bad code. How can our *simple* hardware distinguish between good code and bad code?

We considered and discarded two options:

- *False Start:* Good code could write a *password* to the lock. Although this approach simplifies the necessary circuitry, we had doubts about effectively hiding the passwords from rogue software.
- *False Start:* The lock determines when good code is executing by monitoring the address bus during instruction fetches.

  This approach greatly complicates the circuitry. We felt that correct implementation would be difficult, given the complexities of instruction fetching in modern CPUs, and the subtleties involved in detecting not just the address of an instruction, but the context in which it is executed. For example, it is not sufficient merely to recognize that a sequence of instructions came from the address range for privileged code; the locks would have to further distinguish between

  – these instructions, executing as privileged code;
  – these instructions, executing as a subroutine; called by unprivileged code;
  – these instructions, executing as privileged code, but with a sabotaged interrupt table.

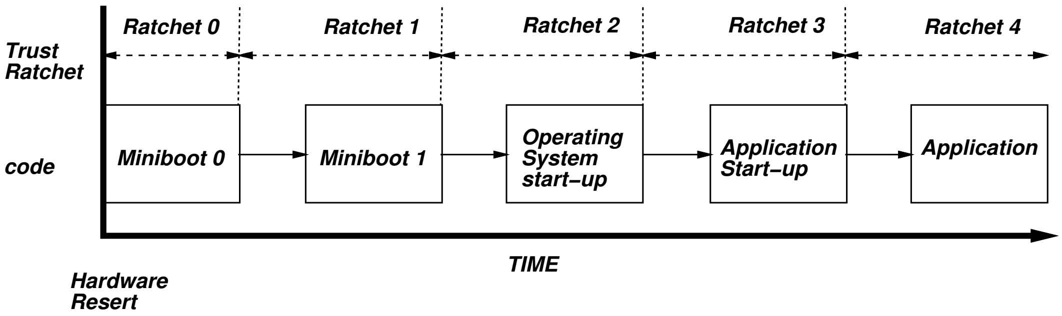

Figure 8 Hardware reset forces the CPU to begin executing Miniboot 0 out of ROM; execution then proceeds through a non-repeating sequence of phases, determined by code and context. Hardware reset also forces the trust ratchet to zero; each block of code advance the ratchet before passing control to the next block in the sequence. However, no code block can decrement the ratchet.

Solution: Time-based Ratchet. We finally developed a lock approach based on the observation that *reset* (a hardware signal that causes all device circuitry return to a known state) forces the device CPU to begin execution from a fixed address in ROM: known, trusted, permanent code. As execution proceeds, it passes through a non-repeating sequence of code blocks with different levels of trust, permanence, and privilege requirements. Figure 8 illustrates this sequence:

- Reset starts Miniboot 0, from ROM;
- Miniboot 0 passes control to Miniboot 1, and never executes again.
- Miniboot 1 passes control to the OS, and never executes again.
- The OS may perform some start-up code.
- While retaining supervisor control, the OS may then execute application code.

- The application (executing under control of the OS) may itself do some start-up work, then (potentially) incur dependence on less trusted code or input.

Our lock design models this sequence with a *trust ratchet*, currently represented as a nonnegative integer. A small microcontroller stores the the ratchet value in a register. Upon hardware reset, the microcontroller resets the ratchet to 0; through interaction with the device CPU, the microcontroller can advance the ratchet—but *can never turn it back*. As each block finishes its execution, it advances the ratchet to the next appropriate value. (Our implementation also enforces a maximum ratchet value, and ensures that ratchet cannot be advanced beyond this value.) Figure 8 also illustrates how this trust ratchet models the execution sequence.

The microcontroller then grants or refuses memory accesses, depending on the current ratchet value.

Decreasing Trust. The effectiveness of this trust ratchet critically depends on two facts:

- The code blocks can be organized into a hierarchy of decreasing privilege levels (e.g., like classical work in protection rings [16] or lattice models of information flow [5, 10]).
- In our software architecture, these privilege levels strictly decrease *in real time!*

This time sequencing, coupled with the independence of the lock hardware from the CPU and the fact that the hardware design (and its physical encapsulation) forces any reset of the locks to also reset the CPU, give the ratchet its power:

- The only way to get the most-privileged level ("Ratchet 0") is to force a hardware reset of the entire system, and begin executing Miniboot 0 from a hardwired address in ROM, in a known state.
- The only way to get a non-maximal privilege level ("Ratchet N," for $N > 0$) is to be passed control by code executing at a an earlier, higher-privileged ratchet level.
- Neither rogue software (nor any other software) can turn the ratchet back to an earlier, higher-privileged level—short of resetting the entire system.

The only avenue for rogue software at Ratchet N to steal the privileges of ratchet $K < N$ would be to somehow alter the software that executes at rachet K or earlier. (However, as Section 7.2 will show, we use the ratchet to prevent these attacks as well.)

Generalizations. Although this discussion used a simple total order on ratchet values, nothing prevents using a partial order. Indeed, as Section 7.2 discusses, our initial implementation of the microcontroller firmware does just that, in order to allow for some avenues for future expansion.

6.4. Privacy and Integrity of Secrets

The hardware locks enable us to address the challenge of Section 6.1: how do we keep rogue software from stealing or modifying critical authentication secrets? We do this by establishing *protected pages*: regions[8] of battery-backed RAM which are locked once the ratchet advances beyond a certain level. The hardware locks can then permit or deny write access to each of these pages—rogue code might still issue a read or write to that address, but the memory device itself will never see it.

Table 1 illustrates the access policy we chose: each Ratchet level R (for $0 \leq R \leq 3$) has its own *protected page*, with the property that Page P can only be read or written in ratchet level $R \leq P$.

We use *lockable BBRAM (LBBRAM)* to refer to the portion of BBRAM consisting of the protected pages. (As with all BBRAM in the device, these regions preserve their contents across periods of no power, but zeroize their contents

[8] The term "page" here refers solely to a particular region of BBRAM—and *not* to special components of any particular CPU memory architecture.

upon tamper.) Currently, these pages are used for outgoing authentication (Section 10); Page 0 also holds some secrets used for ROM-based loading (Section 8).

We partition the remainder of BBRAM into two regions: one belonging to the OS exclusively, and one belonging to the application. Within this non-lockable BBRAM, we expect the OS to protect its own data from the application's.

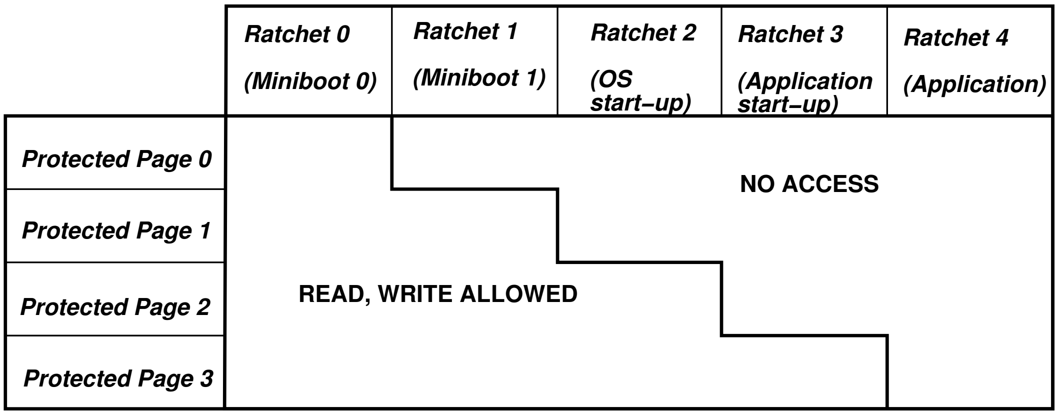

Table 1 Hardware locks protect the privacy and integrity of critical secrets.

7. Code Integrity

The previous sections presented how our architecture ensures that secrets remain accessible only to allegedly trusted code, executing on an untampered device. To be effective, our architecture must integrate these defenses with techniques to ensure that this executing code really is trusted.

This section presents how we address the problem of code integrity:

- Section 7.1 and Section 7.2 describe how we defend against code from being formally modified, except through the official code loading procedure.
- Section 7.3 and Section 7.4 describes how we defend against modifications due to other types of failures.
- Section 7.5 summarizes how we knit these techniques together to ensure the device securely boots.

Note that although our long-term vision of the software architecture (Figure 3) includes simultaneously resident sibling applications and dynamically-loaded applications, this section confines itself to our current implementation, of one application, resident in FLASH.

7.1. Loading and Cryptography

We confine the tasks of deciding and carrying out alteration of code layers to Miniboot. Although previous work considered a hierarchical approach to loading, our commercial requirements (multiple-layer software, controlled by mutually suspicious authorities, updated in the hostile field, while sometimes preserving state) led to trust scenarios that were simplified by centralizing trust management.

Miniboot 1 (in rewritable FLASH) contains code to support public-key cryptography and hashing, and carries out the primary code installation and update tasks—which include updating itself.

Miniboot 0 (in boot-block ROM) contains primitive code to perform DES using the DES-support hardware, and uses secret-key authentication [19] to perform the emergency operations necessary to repair a device whose Miniboot 1 does not function.

(Section 8 will discuss the protocols Miniboot uses.)

7.2. Protection against Malice

As experience in vulnerability analysis often reveals, practice often deviates from policy. Without additional countermeasures, the policy of "Miniboot is in charge of installing and updating all code layers" does not necessarily imply that "the contents of code layers are always changed in accordance with the design of Miniboot, as installed." For example:

- *Without sufficient countermeasures*, malicious code might itself rewrite code layers.
- *Without sufficient countermeasures*, malicious code might rewrite the Miniboot 1 code layer, and cause Miniboot to incorrectly "maintain" other layers.

To ensure that practice meets policy, we use the trust ratchet (Section 6) to guard rewriting of the code layers in rewritable FLASH. We group sets of FLASH sectors into *protected segments*, one[9] for each rewritable layer of code. The hardware locks can then permit or deny write access to each of these segments—rogue code might still issue a write to that address, but the memory device itself will never see it.

[9] Again, the term "segment" is used here solely to denote to these sets of FLASH sectors—and *not* to special components of a CPU memory architecture.

Table 2 illustrates the write policy we chose for protected FLASH. We could have limited Ratchet 0 write-access to Segment 1 alone (since in practice, Miniboot 0 only writes Miniboot 1). However, it makes little security sense to withhold privileges from earlier, higher-trust ratchet levels—since the earlier-level code could always usurp these privileges by advancing the ratchet without passing control.

As a consequence of applying hardware locks to FLASH, malicious code cannot rewrite code layers unless it modifies Miniboot 1. But this is not possible—in order to modify Miniboot 1, an adversary has to either alter ROM, or already have altered Miniboot 1. (Note these safeguards apply only in the realm of attacks that do not result in zeroizing the device. An attacker could bypass all these defenses by opening the device and replacing the FLASH components—but we assume that the defenses of Section 4 would ensure that such an attack would trigger tamper detection and response.)

In order to permit changing to a hierarchical approach without changing the hardware design, the currently implemented lock firmware permits Ratchet 1 to advance instead to a Ratchet $2'$, that acts like Ratchet 2, but permits rewriting of Segment 3. Essentially, our trust ratchet, as implemented, is already ranging over a non-total partial order.

<table>
<tr><th></th><th>Ratchet 0
(Miniboot 0)</th><th>Ratchet 1
(Miniboot 1)</th><th>Ratchet 2
(OS start–up)</th><th>Ratchet 3
(Application start–up)</th><th>Ratchet 4
(Application)</th></tr>
<tr><td>Protected Segment 1
(Miniboot 1)</td><td colspan="2" rowspan="3">READ, WRITE ALLOWED</td><td colspan="3" rowspan="3">READ ALLOWED,
WRITE PROHIBITED</td></tr>
<tr><td>Protected Segment 2
(Operating System/ Control Program)</td></tr>
<tr><td>Protected Segment 3
(Application)</td></tr>
</table>

Table 2 The hardware locks protect the integrity of critical FLASH segments.

7.3. Protection against Reburn Failure

In our current hardware implementation, multiple FLASH sectors make up one protected segment. Nevertheless, we erase and rewrite each segment as a whole, in order to simplify data structures and to accommodate future hardware with larger sectors.

This decision leaves us open to a significant risk: a failure or power-down might occur during the non-zero time interval between the time Miniboot starts erasing a code layer to be rewritten, and the time that the rewrite successfully completes. This risk gets even more interesting, in light of the fact that rewrite of a code layer may also involve changes to other state variables and LBBRAM fields.

When crafting the design and implementation, we followed the rule that the system must remain in a safe state no matter what interruptions occur during operations. This principle is especially relevant to the process of erasing and reburning software resident in FLASH.

- Since Miniboot 1 carries out loading and contains the public-key crypto support, we allocate two regions for it in FLASH Segment 1, so that the old copy exists and is usable up until the new copy has been successfully installed. This approach permits using Miniboot 1 for public-key-based recovery from failures during Miniboot 1 updates.
- When reburning the OS or an application, we temporarily demote its state, so that on the next reset after a failed reburn, Miniboot recognizes that the FLASH layer is now unreliable, and cleans up appropriately.

For more complex transitions, we extend this approach: all changes atomically succeed together, or fail either back to the original state, or to a safe intermediate failure state.

7.4. Protection against Storage Errors

Hardware locks on FLASH protect the code layers from being rewritten maliciously. However, bits in FLASH devices (even in boot block ROM) can change without being formally rewritten—due to the effects of random hardware errors in these bits themselves.

To protect against spurious errors, we include a 64-bit DES-based MAC with each code layer. Miniboot 0 checks itself before proceeding; Miniboot 0 checks Miniboot 1 before passing control; Miniboot 1 checks the remaining segments. The use of a 64-bit MAC from CBC-DES was chosen purely for engineering reasons: it gave a better chance at detecting errors over datasets the size of the protected segments than a single 32-bit CRC, and was easier to implement (even in ROM, given the presence of DES hardware) than more complex CRC schemes.

We reiterate that we *do not* rely solely on single-DES to protect code integrity. Rather, our use of DES as a checksum is solely to protect against random storage errors in a write-protected FLASH segment. An adversary might exhaustively find other executables that also match the DES MAC of the correct code; but in order to do anything with these executables, the adversary must get write-access to that FLASH segment—in which case, the adversary also has write-access to the checksum, so his exhaustive search was unnecessary.

7.5. Secure Bootstrapping

To ensure secure bootstrapping, we use several techniques together:

- The hardware locks on FLASH keep rogue code from altering Miniboot or other code layers.
- The loading protocols (Section 8) keep Miniboot from burning adversary code into FLASH.
- The checksums keep the device from executing code that has randomly changed.

If an adversary can cause (e.g., through radiation) extensive, deliberate changes to a FLASH layer so that it still satisfies the checksum it stores, then he can defeat these countermeasures. However, we believe that the physical defenses of Section 4 would keep such an attack from being successful:

- The physical shielding in the device would make it nearly impossible to produce such carefully focused radiation.
- Radiation sufficiently strong to alter bits should also trigger tamper response.

Consequently, securely bootstrapping a custom-designed, tamper-protected device is easier than the general problem of securely bootstrapping a general-purpose, exposed machine (e.g., [4, 9, 26]).

Execution Sequence Our boot sequence follows from a common-sense assembly of our basic techniques. Hardware reset forces execution to begin in Miniboot 0 in ROM. Miniboot 0 begins with *Power-on Self Test 0 (POST0)*, which evaluates the hardware required for the rest of Miniboot 0 to execute. Miniboot 0 verifies the MACs for itself and Miniboot 1. If an external party presents an alleged command for Miniboot 0 (e.g., to repair Miniboot 1 (Section 8), Miniboot 0 will evaluate and respond to the request, then halt. Otherwise Miniboot 0 advances the trust ratchet to 1, and (if Layer 1 is reliable) jumps to Miniboot 1.

Except for some minor, non-secret device-driver parameters, no DRAM state is saved across the Miniboot 0 to Miniboot 1 transition. (In either Miniboot, any error or stateful change causes it to halt, in order to simplify analysis. Interrupts are disabled.)

Miniboot 1 begins with *POST1*, which evaluates the remainder of the hardware. Miniboot 1 also verifies MACs for Layers 2 and 3. If an external party presents an alleged command for Miniboot 1 (e.g., to reload Layer 2), Miniboot 1 will evaluate and respond to the request, then halt. Otherwise Miniboot 1 advances the trust ratchet to 2, and (if Layer 2 is reliable) jumps to the Layer 2, the OS.

The OS then proceeds with its bootstrap. If the OS needs to protect data from an application that may find holes in the OS, the OS can advance the trust ratchet to 3 before invoking Layer 3 code. Similarly, the application can advance the ratchet further, if it needs to protect its private data.

8. Code Loading

8.1. Overview

One of the last remaining pieces of our architecture is the secure installation and update of trusted code.

In order to accommodate our overall goal of enabling widespread development and deployment of secure coprocessor applications, we need to consider the practical aspects of this process. We review the principal constraints:

- **Shipped empty.** In order to minimize variations of the hardware and to accommodate U.S. export regulations, it was decided that all devices would leave the factory with only the minimal software configuration[10] (Miniboot only). The manufacturer does not know at ship time (and may perhaps never know later) where a particular device is going, and what OS and application software will be installed on it.
- **Impersonal broadcast.** To simplify the process of distributing code, the code-loading protocol should permit the process to be one-round (from authority to device), be impersonal (the authority does not *need* to customize the load for each device), and have the ability to be carried out on a public network.
- **Updatable.** As discussed in Section 2.1, we need to be able to update code already installed in devices.
- **Minimal disruption.** An emphatic customer requirement was that, whenever reasonable and desired, application state be preserved across updates.
- **Recoverable.** We need to be able to recover an untampered device from failures in its rewritable software—which may include malicious or accidental bugs in the code, as well as failures in the FLASH storage of the code, or interruption of an update.
- **Loss of Cryptography.** The complexity of public-key cryptography and hashing code forced it to reside in a rewritable FLASH layer—so the recoverability constraint also implies secure recoverability without these abilities.
- **Mutually Suspicious, Independent Authorities.** In any particular device, the software layers may be controlled by different authorities who may not trust each other, and may have different opinions and strategies for software update.
- **Hostile environments.** We can make no assumptions about the user machine itself, or the existence of trusted couriers or trusted security officers.

To address these constraints, we developed and followed some guidelines:

- We make sure that Miniboot keeps its integrity, and that only Miniboot can change the other layers.
- We ensure that the appropriate authorities can obtain and retain control over their layers—despite changes to underlying, higher-trust layers.
- We use public-key cryptography whenever possible.

Section 8.2 below outlines who can be in charge of installing and changing code. Section 8.3 discusses how a device can authenticate them. Section 8.4 discusses how an "empty" card in the hostile field can learn who is in charge of its code layers. Section 8.5 and Section 8.6 discuss how the appropriate authorities can authorize code installations and updates. Section 8.7 summarizes software configuration management for devices. Section 8.8 illustrates the development process with a simple example.

[10] Our design and implementation actually accommodates any level of pre-shipment configuration, should this decision change.

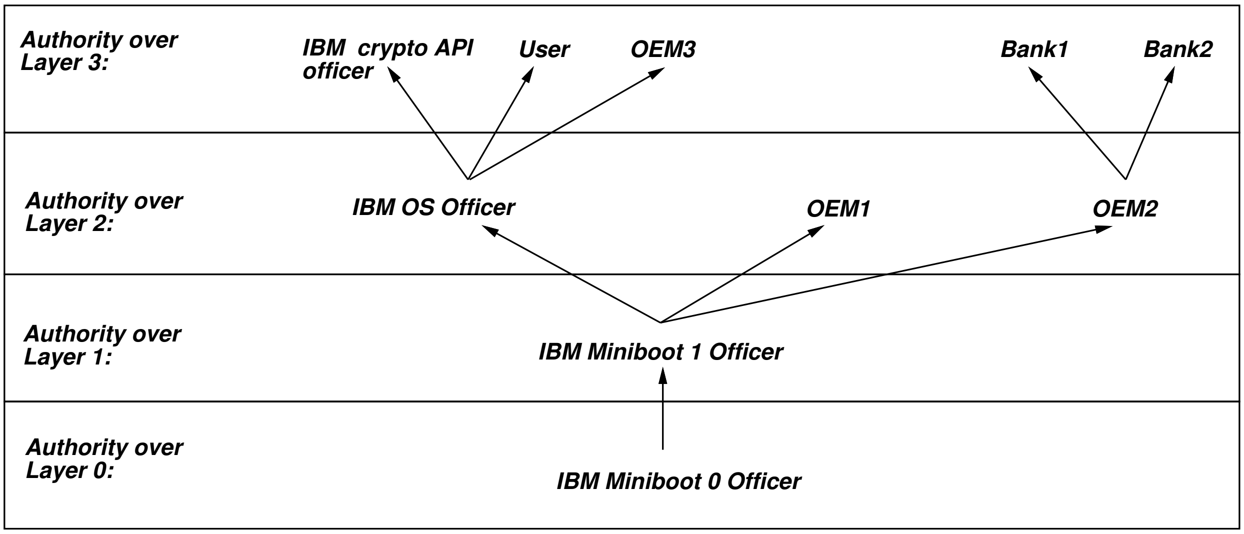

Figure 9 Authorities over software segments are organized into a tree.

8.2. Authorities

As Figure 9 illustrates, we organize *software authorities*—parties who might authorize the loading of new software—into a tree. The root is the sole owner of Miniboot; the next generation are the authorities of different operating systems; the next are the authorities over the various applications that run on top of these operating systems. We stress that these parties are external entities, and apply to the entire family of devices, not just one.

Hierarchy in software architecture implies dependence of *software*. The correctness and security of the application layer depends on the correctness and security of the operating system, which in turn depends on Miniboot 1, which in turn depends on Miniboot 0. (This relation was implied by the decreasing privileges of the trust ratchet.)

Similarly, hierarchy in the authority tree implies dominance: the authority over Miniboot dominates all operating system authorities; the authority over a particular operating system dominates the authorities over all applications for that operating system.

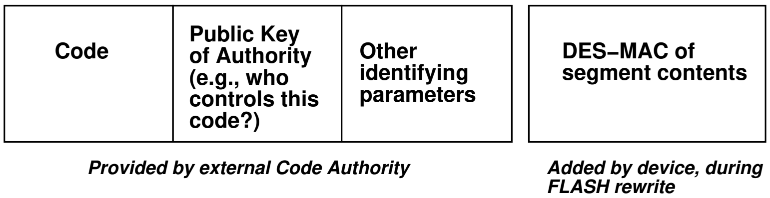

Figure 10 Sketch of the contents of code layer.

8.3. Authenticating the Authorities

Public-Key Authentication. Wherever possible, a device uses a public-key signature to authenticate a message allegedly from one of its code authorities. The public key against which this message is verified is stored in the FLASH segment for that code layer, along with the code and other parameters (see Figure 10).

Using public-key signatures makes it possible to accommodate the "impersonal broadcast" constraint. Storing an authority's public key along with the code, in the FLASH layer owned by that authority, enables the authority to change its keypair over time, at its own discretion. (Adding expiration dates and revocation lists would provide greater forward integrity.)

However, effectively verifying such a signature requires two things:

- the code layer is already loaded and and still has integrity (so the device actually knows the public key to use); and
- Miniboot 1 still functions (so the device knows what to do with this pubic key).

These facts create the need for two styles of loading:

- *ordinary* loading, when these conditions both hold; and
- *emergency* loading, when at least one fails.

Secret-Key Authentication. The lack of public-key cryptography forces the device to use a secret-key handshake to authenticate communications from the Miniboot 0 authority. The shared secrets are stored in Protected Page 0, in LBBRAM. Such a scheme requires that the authority share these secrets. Our scheme [19] reconciles this need with the no-databases requirement by having the device itself store a signed, encrypted message from the authority to itself. During factory initialization, the device itself generates the secrets and encrypts this message; the authority signs the message and returns it to the device for safekeeping. During authentication, the device returns the message to the authority.

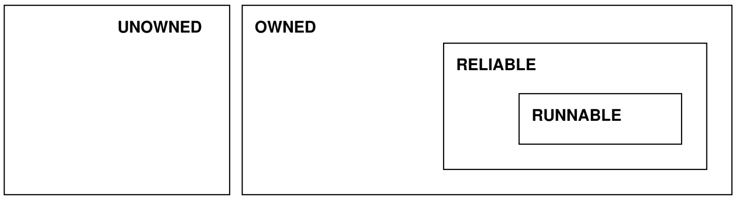

Figure 11 State space of the OS and application code layers.

8.4. Ownership

Clearly, our architecture has to accommodate the fact that each rewritable code layer may have contents that are either reliable or unreliable. However, in order to provided the necessary configuration flexibility, the OS and application layers each have additional parameters, reflecting which external authority is in charge of them.

Our architecture addresses this need by giving each of these layers the state space sketched in Figure 11:

- The code layer may be *owned* or *unowned.*
- The contents of an owned code layer may be *reliable.* However, some owned layers—and all unowned ones—are *unreliable.*
- A reliable code layer may actually be *runnable.* However, some reliable layers—and all unreliable ones— may be *unrunnable.*

This code state is stored in EEPROM fields in the hardware lock, write-protected beyond Ratchet 1.

For $0 < N < 3$, the authority over Layer N in a device can issue a Miniboot command giving an *unowned* Layer $N + 1$ to a particular authority. For $2 \leq N \leq 3$, the authority over Layer N can issue a command surrendering ownership—but the device can evaluate this command only if Layer N is currently reliable. (Otherwise, the device does not know the necessary public key.)

8.5. Ordinary Loading

General Scheme. Code Layer N, for $1 \leq N \leq 3$, is rewritable. Under ordinary circumstances, the authority over layer N can update the code in that layer by issuing an update command signed by that authority's private key. This command includes the new code, a new public key for that authority (which could be the same as the old one, per that authority's key policy), and target information to identify the devices for which this command is valid. The device (using Miniboot 1) then verifies this signature *directly* against the public key currently stored in that layer.

Figure 12 sketches this structure.

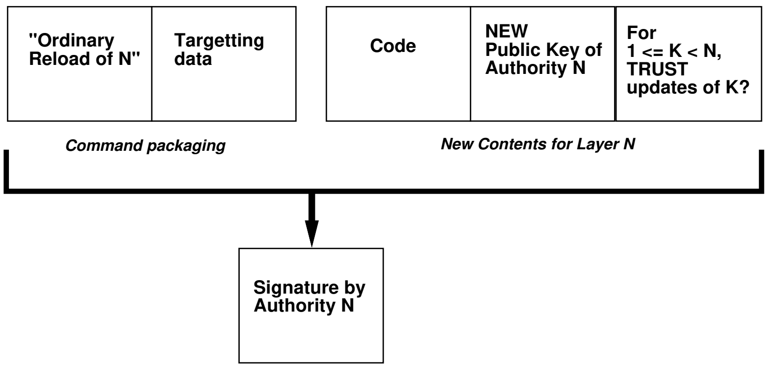

Figure 12 An ordinary load command for Layer N consists of the new code, new public key, and trust parameters, signed by the authority over that layer; this signature is evaluated against the public key currently stored in that layer.

Target. The target data included with all command signatures allows an authority to ensure that their command applies only in an appropriate trusted environment. An untampered device will accept the signature as valid only if the device is a member of this set. (The authority can verify that the load "took" via a signed receipt from Miniboot—see Section 10.)

For example, suppose an application developer determines that version 2 of a particular OS has a serious security vulnerability. Target data permits this developer to ensure that their application is loadable only on devices with version 3 or greater of that operating system.

Underlying Updates. The OS has complete control over the application, and complete access to its secrets; Miniboot has complete control over both the OS and the application. This control creates the potential for serious backdoors. For example, can the OS authority trust that the Miniboot authority will always ship updates that are both secure and compatible? Can the application authority trust that the OS authority uses appropriate safeguards and policy to protect the private key he or she uses to authorize software upgrades?

To address these risks, we permit Authority N to include, when loading its code, *trust parameters* expressing how it feels about future changes to each rewritable layer $K < N$. For now, these parameters have three values: *always* trust, *never* trust, or trust only if the update command for K is *countersigned* by N.

As a consequence, an ordinary load of Layer N can be accompanied by, for $N < M \leq 3$, a *countersignature* from Authority M, expressing compatibility. Figure 13 sketches this structure.

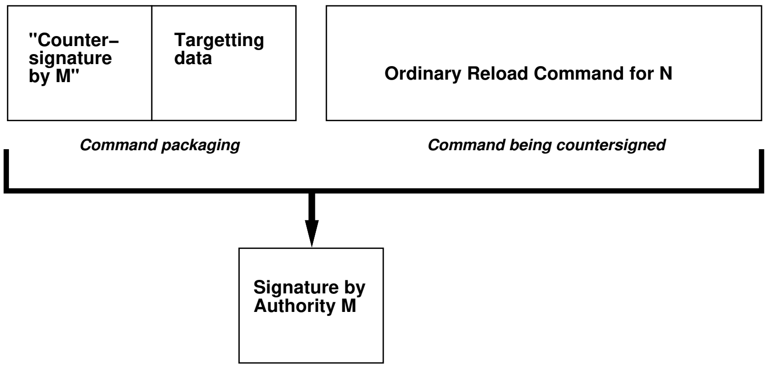

Figure 13 An ordinary load command for Layer N can include an optional countersignature by the authority over a dependent Layer M. This countersignature is evaluated against the public key currently stored in layer M.

Update Policy Trust parameters and countersignatures help us balance the requirements to support hot updates, against the risks of dominant authorities replacing underlying code.

An ordinary reload of Layer N, if successful, preserves the current secrets of Layer N, and leaves Layer N runnable.

For $N < M \leq 3$, an ordinary reload of Layer N, if successful, preserves the current secrets of Layer M if and only if Layer M had been reliable, and either:

- its trust parameter for N was *always*, or
- its trust parameter for N was *countersigned*, and a valid countersignature from M was included.

Otherwise, the secrets of M are atomically destroyed with the update.

An ordinary load of a layer always preserves that layer's secrets, because presumably an authority can trust their own private key.

8.6. Emergency Loading

As Section 8.4 observes, evaluating Authority N's signature on a command to update Layer N requires that Layer N have reliable contents. Many scenarios arise where Layer N will not be reliable—including the initial load of the OS and application in newly shipped cards, and repair of these layers after an interruption during reburn.

Consequently, we require an *emergency* method to load code into a layer without using the contents of that layer. As Figure 14 shows, an emergency load command for Layer N must be authenticated by Layer $N-1$. (As discussed below, our architecture includes countermeasures to eliminate the potential backdoors this indirection introduces.)

OS, Application Layers. To emergency load the OS or Application layers, the authority signs a command similar to the ordinary load, but the authority underneath them signs a statement attesting to the public key. Figure 15 illustrates this. The device evaluates the signature on this emergency certificate against the public key in the underlying segment, then evaluates the main signature against the public key in the certificate.

This two-step process facilitates software distribution: the emergency authority can sign such a certificate once, when the next-level authority first joins the tree. This process also isolates the code and activities of the next-level authority from the underlying authority.

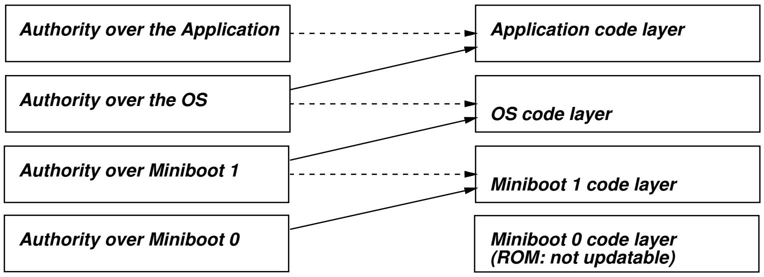

Figure 14 Ordinary loading of code into a layer is directly authenticated by the authority over that layer (dashed arrows); emergency loading is directly authenticated by the authority underlying that layer (solid arrows).

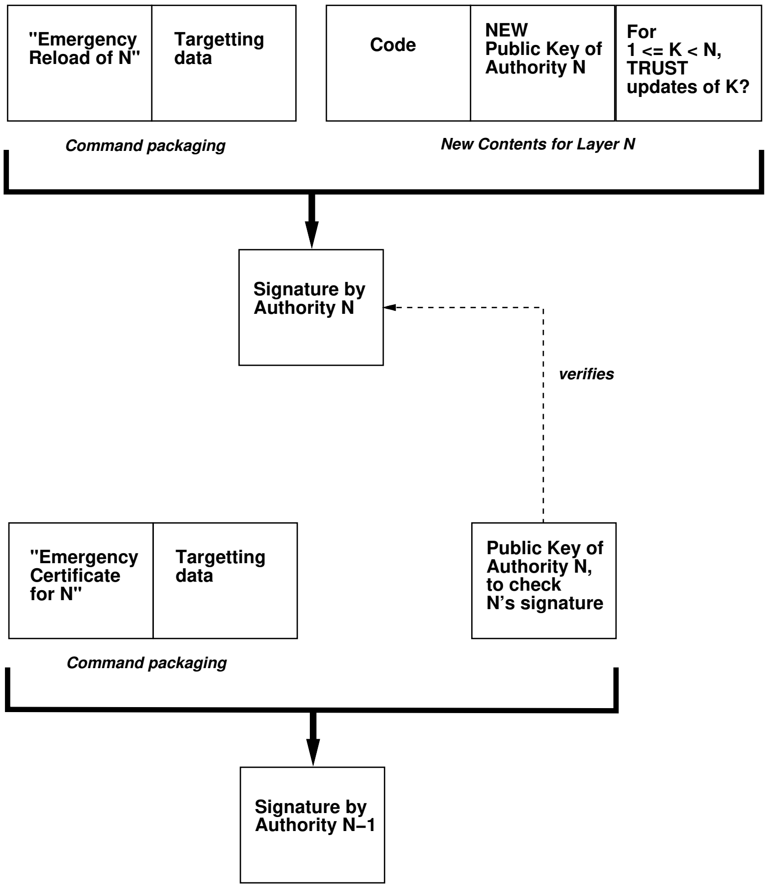

Figure 15 An emergency load command (for $N = 2, 3$) consists of the new code, new public key, and trust parameters, signed by the authority over that layer; and an emergency certificate signed by the authority over the underlying layer. The main signature is evaluated against the public key in the certificate; the certificate signature is evaluated against the public key stored in the underlying layer.

Risks of Siblings. Burning a segment without using the contents of that segment introduces a problem: keeping an emergency load of one authority's software from overwriting installed software from a sibling authority. We address this risk by giving each authority an *ownerID*, assigned by the $N-1$ authority when establishing ownership for N (Section 8.4), and stored outside the code layer. The public-key certificate later used in the emergency load of N specifies the particular *ownerID*, which the device checks.

Emergency Reloading of Miniboot. Even though we mirror Miniboot 1, recoverability still required that we have a way of burning it without using it, in order to recover from emergencies when the Miniboot 1 code layer does not function. Since we must use ROM only (and *not* Miniboot 1), we cannot use public-key cryptography, but instead use mutual authentication between the device and the Miniboot 0 authority, based on device-specific secret keys—see Section 8.3.

Backdoors. Emergency loading introduces the potential for backdoors, since reloading Layer N does not *require* the participation of the authority over that segment. For example, an OS authority could, by malice or error, put *anyone's* public key in the emergency certificate for a particular application authority.

Closing the Backdoors. Since the device cannot really be sure that an emergency load for Layer N really came from the genuine Authority N, Miniboot enforces two precautions:

- It erases the current Layer N secrets but leaves the segment runnable from this clean start (since the alleged owner trusts it).
- It erases all secrets belonging to later layers, and leaves them unrunnable (since their owners cannot directly express trust of this new load—see Section 9).

These actions take place atomically, as part of a successful emergency load.

8.7. Summary

This architecture establishes individual commands for Authority N to:

- establish owner of Layer $N+1$
- attest to the public key of Layer $N+1$
- install and update code in Layer N
- express opinions about the trustworthiness of future changes to Layer $K < N$.

Except for emergency repairs to Miniboot 1, all these commands are authenticated via public-key signatures, can occur over a public network, and can be restricted to particular devices in particular configurations.

Depending on how an authority chooses to control its keypairs and target its commands, these commands can be assembled into sequences that meet the criteria of Section 2.1. A separate report [20] explores some of the scenarios this flexibility enables.

8.8. Example Code Development Scenario

We illustrate how this architecture supports flexible code development with a simple example.

Suppose Alice is in charge of Miniboot 1, and Bob wants to become a Layer 2 owner, in order to develop and release Layer 2 software on some cards.

Bob generates his keypair, and gives a copy of his public key to Alice. Alice then does three things for Bob:

- She assigns Bob a 2-byte ownerID value that distinguishes him among all the other children of Alice. (Recall Figure 9.)
- She signs an "Establish Owner 2" command for Bob.
- She signs an "Emergency Signature" for an "Emergency Burn 2" saying that Owner 2 Bob has that public key. (Recall Figure 15.)

Bob then goes away, writes his code, prepares the remainder of his "Emergency Burn 2" command, and attaches the signature from Alice.

Now, suppose customer Carol wants to load Bob's program into Layer 2 on her card. She first buys a virgin device (which has an *unowned* Layer 2, but has Miniboot 1 and Alice's public key in Layer 1). Carol gets from Bob his "Establish Owner 2" and "Emergency Burn 2" command, and plays them into her virgin card via Miniboot 1. It verifies Alice's signatures and accepts them. Layer 2 in Carol's card is now owned by Bob, and contains Bob's Program and Bob's Public key.

If Bob wants to update his code and/or keypair, he simply prepares an "Ordinary Burn 2" command, and transmits it to Carol's card. Carol's card checks his signature on the update against the public key it has already stored for him.

Note that Bob never exposes to Alice his private key, his code, his pattern of updates, or the identity of his customers. Furthermore, if Bonnie is *another* Layer 2 developer, she shares no secrets with Bob, and updates for Bonnie's software will not be accepted by cards owned by Bob's.

The architecture also support other variations in the installation/development process; for example, maybe Bob buys the cards himself, configures them, *then* ships them to Carol.

(The case for Layer 3 developers is similar.)

9. Securing the Execution

This section *summarizes* how our architecture build on the above techniques to satisfy the security requirements of Section 2.2.1. (Although a formal proof is beyond the scope of this paper, we have completed a side-project to formally specify these properties, formally model the behavior of the system, and mechanically verify that the system preserves these properties [18].)

9.1. Control of Software

Loading software in code Layer N in a particular device requires the cooperation of at least one current authority, over some $0 \leq K \leq N$.

- From the code integrity protections of Section 8, the only way to change the software is through Miniboot.
- From the authentication requirements for software loading and installation (which Table 3 summarizes), any path to changing Layer N in the future requires an authenticated command from some $K \leq N$ now.
- From the hardware locks protecting Page 0 (and the intractability assumptions underlying cryptography), the only way to produce this command is to access the private key store of that authority.

<table>
<tr><th>Miniboot Command</th><th></th><th></th><th>Authentication Required</th></tr>
<tr><td>Establish Owner</td><td>of layer N</td><td></td><td>Authority N−1</td></tr>
<tr><td>Surrender Owner</td><td>of layer N</td><td></td><td>Authority N</td></tr>
<tr><td rowspan="2">Emergency Load</td><td>of layer N</td><td></td><td>Authority N−1</td></tr>
<tr><td>of layer K < N</td><td></td><td>Authority K−1</td></tr>
<tr><td rowspan="3">Ordinary Load</td><td>of layer N</td><td></td><td>Authority N</td></tr>
<tr><td rowspan="2">of layer K < N</td><td>Trusted by Auth N</td><td rowspan="2">Authority K (trust from Authority N)</td></tr>
<tr><td>Untrusted by Auth N</td></tr>
</table>

Table 3 Summary of authentication requirements for Miniboot commands affecting Layer N.

9.2. Access to Secrets

9.2.1. Policy

The multiple levels of software in the device are hierarchically dependent: the correct execution of the application depends on the correct execution of the operating system, which in turn depends on the correct execution of Miniboot. However, when considered along the fact that these levels of software might be independently configured and updated by authorities who may not necessarily trust each other, this dependence gives rise to many risks.

We addressed these risks by formulating and enforcing a policy for secure execution:

> *A program can run and accumulate state only while the device can continuously maintain a trusted execution environment for that program.*

The *execution environment* includes both underlying untampered device, as well as the code in this and underlying layers. The *secrets* of a code layer are the contents of its portion of BBRAM.

The authority responsible for a layer must do the *trusting* of that layer's environment—but the device itself has to verify that trust. To simplify implementation, we decided that changes to a layer's environment must be verified as trusted before the change takes effect, and that the device must be able to verify the expression of trust *directly* against that authority's public key.

9.2.2. Correctness

Induction establishes that our architecture meets the policy. Let us consider Layer N; the inductive assumption is the device can directly verify that Authority N trusts the execution environment for Layer N.

Initial State. A successful emergency load of layer N leaves N in a runnable state, with cleared secrets. This load establishes a relationship between the device and a particular Authority N. The device can *subsequently* directly authenticate commands from this authority, since it now knows the public key.

This load can only succeed if the execution environment is deemed trustworthy, as expressed by the target information in Authority N's signature.

Run-time. During ordinary execution, secure bootstrapping (Section 7) and the hardware locks on LBBRAM (Section 6) ensure that only code currently in the execution environment can directly access Layer N's secrets—and by inductive assumption, Authority N trusts this software not to compromise these secrets.

Changes. The execution environment for Layer N can change due to reloads, to tamper, and to other failure scenarios. Our architecture preserves the Layer N secrets if and only if the change preserves the trust invariant. Table 4 summarizes how these changes affect the state of Layer N; Table 5 summarize how the new state of Layer N affects the secrets of Layer N.

A runnable Layer N stops being runnable if the change in execution environment causes the inductive assumption to fail—unless this change was an emergency load of Layer N, in which case the Layer N secrets are cleared back to an initial state.

- Layer N becomes *unowned* if the environment changes in way that makes it impossible for Authority N to express trust again: the device is tampered, or if Layer 1 (the public key code) becomes untrusted, or if Layer $N-1$ becomes unowned (so the *ownerID* is no longer uniquely defined).
- Layer N also becomes *unowned* if Authority N has explicitly surrendered ownership.
- Layer N becomes *unreliable* if its integrity fails. (Authority N can still express trust, but only indirectly, with the assistance of Authority $N-1$.)
- Otherwise, Layer N stops being *runnable* if an untrusted change occurred.

Layer N stays runnable only for three changes:

- An emergency load of Layer N.
- An ordinary reload of Layer N.
- An ordinary reload of Layer $K < N$, for which Authority N directly expressed trust by either signing an "always trust K" trust parameter at last load of Layer N, or by signing an "trust K if countersigned" at last load of N, and signing a countersignature now.

Only the latter two changes preserve the trust invariant—and, as Table 5 shows, only these preserve the Layer N secrets.

Implementation. Code that is *already part of the trusted environment* carries out the erasure of secrets and other state changes. In particular, the combined efforts of Miniboot 0 (permanently in ROM) and the Miniboot 1 *currently* in Layer 1 (hence already trusted) take care of the clean-up required by an authority that does not trust a new Miniboot 1—despite failures during the load process.

Action				*Transformation of Layer N state*
RELIABLE Layer N fails checksum				NOT RELIABLE
Layer 2<N is OWNED but NOT RUNNABLE				NOT RUNNABLE
Layer 2<N is UNOWNED				UNOWNED
Layer 1 is NOT RELIABLE				UNOWNED
Device is ZEROIZED				UNOWNED
Establish Owner	of layer N			OWNED
Surrender Owner	of layer N			UNOWNED
Emergency Load	of layer N			RUNNABLE
Emergency Load	of layer K < N		K = 2	NOT RUNNABLE
Emergency Load	of layer K < N		K = 1	UNOWNED
Ordinary Load	of layer N			RUNNABLE
Ordinary Load	of layer K < N	Trusted by Auth N		no change
Ordinary Load	of layer K < N	Untrusted by Auth N	K = 2	NOT RUNNABLE
Ordinary Load	of layer K < N	Untrusted by Auth N	K = 1	UNOWNED

Table 4 Summary of how the state of Layer N changes with changes to its execution environment.

Action		*Transformation of Layer N secrets*
Layer N is NOT RUNNABLE		ZEROIZED
Layer N is RUNNABLE	Emergency Load of Layer N	Cleared to Initial State
Layer N is RUNNABLE	Otherwise	PRESERVED

Table 5 Summary of how changes to the state of Layer N changes its secrets.

10. Authenticating the Execution

10.1. The Problem

The final piece of our security strategy involves the requirement of Section 2.2.2: how to authenticate computation allegedly occurring on an untampered device with a particular software configuration. (Section 8.3 explained how the device can authenticate the external world; this section explains how the external world can authenticate the device.)

It must be possible for a remote participant to distinguish between a message from the real thing, and a message from a clever adversary. This authentication is clearly required for distributed applications using coprocessors. As noted earlier, the e-wallet example of Yee [26] only works if it's the real wallet on a real device. But this authentication is also required even for more pedestrian coprocessor applications, such as physically secure high-end cryptographic modules. For example, a sloppy definition of "secure" software update on crypto modules may require only that the appropriate authority be able to update the code in an untampered device. If a security officer has two devices, one genuine and one evilly modified, but can never distinguish between them, then it does not matter if the genuine one can be genuinely updated. This problem gets even worse if updates all occur remotely, on devices deployed in hostile environments.

10.2. Risks

Perhaps the most natural solution to authentication is to sign messages with the device private key that is established in initialization (Section 5) and erased upon tamper. However, this approach, on its own, does not address the threats introduced by the multi-level, updated, software structure. For example:

- **Application Threats.** What prevents one application from signing messages claiming to be from a different application, or from the operating system or Miniboot? What prevents an application from requesting sufficiently many "legitimate" signatures to enable cryptanalysis? What if an Internet-connected application has been compromised by a remote adversary?
- **OS Threats.** If use of the device private key is to be available to applications in real-time, then (given the infeasibility of address-based hardware access control) protection of the key depends entirely on the operating system. What if the operating system has holes? We are back to the scenario of Section 6.1.
- **Miniboot Threats.** An often-overlooked aspect of security in real distributed systems is the integrity of the cryptographic code itself. How can one distinguish between a good and corrupted version of Miniboot 1? Not only could a corrupt version misuse the device private key—it can also lie about who it is.

This last item is instance of the more general *versioning* problem. As the software configuration supporting a particular segment changes over time, its trustworthiness in the eyes of a remote participant may change. If one does not consider the old version of the OS or the new version of an application to be trustworthy, then one must be able to verify that one is not talking to them. The authentication scheme must accommodate these scenarios.

10.3. Our Solution

These risks suggest the need for decoupling between software levels, and between software versions. Our architecture carries out this strategy (although currently, we have only implemented the bottom level, for Layer 1).

As Section 5 explained, we build an internal key hierarchy, starting with the keypair certified for Miniboot 1 in a device at device initialization. This private key is stored in Page 1 in LBBRAM—so it is visible only to Miniboot 1. Our architecture has Miniboot 1 *regenerate* its keypair as an atomic part of each ordinary reload of Miniboot 1. The transition certificate includes identification of the versions of Miniboot involved. (As Section 8 discusses, each emergency reload of Miniboot 1 erases its private key—the authority who just carried out mutual secret-key authentication must then re-initialize the device.)

Similarly, as an atomic part of loading any higher Layer N (for $N > 1$), our architecture has the underlying Layer $N - 1$ generate a new keypair for Layer N, and then certify the new public key and deletes the old private key. This certification includes identification of the version of the code. Although Miniboot could handle the keys for everyone, our current plan is for Miniboot to certify the outgoing keypair for the operating system, and for our operating system to certify the keypair for the application—because this scheme more easily accommodates customer requirements for application options. The OS private key will be stored in Page 2 in LBBRAM.

Our approach thus uses two factors:

- Certification *binds* a keypair to the layers and versions of code that could have had access to the private key.
- The loading protocol along with the hardware-protected memory structure *confines* the private key to exactly those versions.

This approach provides recoverability from compromise. Code deemed untrustworthy cannot spoof without the assistance of code deemed trustworthy. An untampered device with a trusted Miniboot 1 can always authenticate and repair itself with public-key techniques; an untampered device with trusted ROM can always authenticate itself and repair Miniboot 1 with secret-key techniques.

This approach also arguably minimizes necessary trust. For example, in Figure 16, if Program F is going to believe in the authenticity of the mystery message, then it arguably must trust everything inside the dotted line—because if any of those items leaked secrets, then the message could not be authenticated anyway. But our scheme does not force Program F to trust anything outside the dotted line (except the integrity of the original CA).

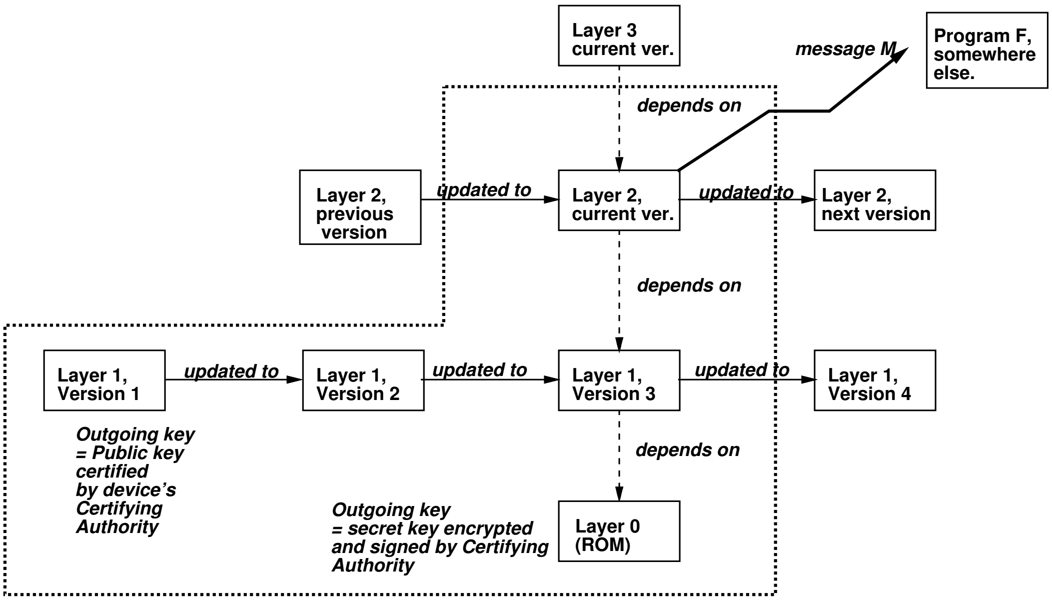

Figure 16 Our outgoing authentication strategy requires that, in order to authenticate message M, Program F trust only what's inside the dotted line—which it would have to trust anyway.

11. Conclusions and Future Work

We plan immediate work into extending the device. The reloadability of Miniboot 1 and the operating system allows exploration of upgrading the cryptographic algorithms (e.g., perhaps to elliptic curves, as well as certificate blacklists and expiration) as well as additional trust parameters for policy enforcement. Hardware work also remains. In the short run, we plan to finish addressing the engineering challenges in moving this technology into PCMCIA format.

However, the main avenue for future work is to develop applications for this technology, and to enable others to develop applications for it. We view this project not as an end-result, but rather as a tool, to finally make possible widespread development and deployment of secure coprocessor solutions.

Acknowledgments

The authors gratefully acknowledge the contributions of entire Watson development team, including Vernon Austel, Dave Baukus, Suresh Chari, Joan Dyer, Gideon Eisenstadter, Bob Gezelter, Juan Gonzalez, Jeff Kravitz, Mark Lindemann, Joe McArthur, Dennis Nagel, Elaine Palmer, Ron Perez, Pankaj Rohatgi, David Toll, and Bennet Yee; the IBM Global Security Analysis Lab at Watson, and the IBM development teams in Vimercate, Charlotte, and Poughkeepsie.

We also wish to thank Ran Canetti, Michel Hack, and Mike Matyas for their helpful advice, and Bill Arnold, Liam Comerford, Doug Tygar, Steve White, and Bennet Yee for their inspirational pioneering work, and the anonymous referees for their helpful comments.

DRAFT

References

[1] D. G. Abraham, G. M. Dolan, G. P. Double, J. V. Stevens. “Transaction Security Systems.” *IBM Systems Journal.* 30:206-229. 1991.

[2] R. Anderson, M. Kuhn. “Tamper Resistance—A Cautionary Note.” *The Second USENIX Workshop on Electronic Commerce.* November 1996.

[3] R. Anderson, M. Kuhn. *Low Cost Attacks on Tamper Resistant Devices.* Preprint. 1997.

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

[5] D. E. Bell and L. J. LaPadula. *Secure Computer Systems: Mathematical Foundations and Model.* Technical Report M74-244, MITRE Corporation. May 1973.

[6] E. Biham, A. Shamir. *Differential Fault Analysis: A New Cryptanalytic Attack on Secret Key Cryptosystems.* Preprint, 1997.

[7] D. Boneh, R. A. DeMillo, R. J. Lipton. *On the Importance of Checking Computations.* Preprint, 1996.

[8] D. Chaum. “Design Concepts for Tamper Responding Systems.” *CRYPTO 83.*

[9] P. C. Clark and L. J. Hoffmann. “BITS: A Smartcard Protected Operating System.” *Communications of the ACM.* 37: 66-70. November 1994.

[10] D. E. Denning. “A Lattice Model of Secure Information Flow.” *Communications of the ACM.* 19: 236-243. May 1976.

[11] W. Havener, R. Medlock, R. Mitchell, R. Walcott. *Derived Test Requirements for FIPS PUB 140-1.* National Institute of Standards and Technology. March 1995.

[12] *IBM PCI Cryptographic Coprocessor.* Product Brochure G325-1118. August 1997.

[13] M. F. Jones and B. Schneier. “Securing the World Wide Web: Smart Tokens and their Implementation.” *Fourth International World Wide Web Conference.* December 1995.

[14] National Institute of Standards and Technology. *Security Requirements for Cryptographic Modules.* Federal Information Processing Standards Publication 140-1, 1994.

[15] E. R. Palmer. *An Introduction to Citadel—A Secure Crypto Coprocessor for Workstations.* Computer Science Research Report RC 18373, IBM T. J. Watson Research Center. September 1992.

[16] M. D. Schroeder and J. H. Saltzer. “A Hardware Architecture for Implementing Protection Rings.” *Comminications of the ACM.* 15” 157-170. March 1972.

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

[18] S. W. Smith, V. Austel. “Trusting Trusted Hardware: Towards a Formal Model for Programmable Secure Coprocessors.” *The Third USENIX Workshop on Electronic Commerce.* September 1998.

[19] S. W. Smith, S. M. Matyas. *Authentication for Secure Devices with Limited Cryptography.* IBM T. J. Watson Research Center. Design notes, August 1997.

[20] 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, to appear, 1998.

[21] 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.

[22] S. H. Weingart. “Physical Security for the μABYSS System.” *IEEE Computer Society Conference on Security and Privacy.* 1987.

[23] S. R. White, L. D. Comerford. “ABYSS: A Trusted Architecture for Software Protection.” *IEEE Computer Society Conference on Security and Privacy.* 1987.

[24] S. R. White, S. H. Weingart, W. C. Arnold and E. R. Palmer. *Introduction to the Citadel Architecture: Security in Physically Exposed Environments.* Technical Report RC 16672, Distributed Systems Security Group. IBM T. J. Watson Research Center. March 1991.

[25] S. H. Weingart, S. R. White, W. C. Arnold, and G. P. Double. “An Evaluation System for the Physical Security of Computing Systems.” *Sixth Annual Computer Security Applications Conference.* 1990.

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

[27] B. S. Yee, J. D. Tygar. “Secure Coprocessors in Electronic Commerce Applications.” *The First USENIX Workshop on Electronic Commerce.* July 1995.

[28] A. Young and M. Yung. “The Dark Side of Black-Box Cryptography— or—should we trust Capstone?” *CRYPTO 1996.* LNCS 1109.