Trusting Trusted Hardware: Towards a Formal Model for Programmable Secure Coprocessors *

Sean W. Smith   Vernon Austel
Secure Systems and Smart Cards
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598-0704
sean@watson.ibm.com   austel@watson.ibm.com

Abstract

Secure coprocessors provide a foundation for many exciting electronic commerce applications, as previous work [20, 21] has demonstrated. As our recent work [6, 13, 14] has explored, building a high-end secure coprocessor that can be easily programmed and deployed by a wide range of third parties can be an important step toward realizing this promise. But this step requires *trusting* trusted hardware—and achieving this trust can be difficult in the face of a problem and solution space that can be surprisingly complex and subtle.

Formal methods provide one means to express, verify, and analyze such solutions (and would be required for such a solution to be certified at FIPS 140-1 Level 4). This paper discusses our current efforts to apply these principles to the architecture of our secure coprocessor. We present formal statements of the security goals our architecture needs to provide; we argue for *correctness* by enumerating the architectural properties from which these goals can be proven; we argue for *conciseness* by showing how eliminating properties causes the goals to fail; but we discuss how simpler versions of the architecture can satisfy weaker security goals.

We view this work as the beginning of developing formal models to address the trust challenges arising from using trusted hardware for electronic commerce.

* *The Third USENIX Workshop on Electronic Commerce Proceedings.* To appear, August 1998.

1 Introduction

Motivation and Context Programmable secure coprocessors can be the foundation for many e-commerce applications, as has been demonstrated in the laboratory (e.g., Tygar and Yee's seminal work [20, 21]). To enable real-world deployment of such applications, our team at IBM has recently completed a multi-year project to design and build such a device, that also meets the requirements of realistic manufacturing and use scenarios [13] (and which reached market [6] in the fall of 1997). Bootstrap software that enables secure configuration and programmability, while requiring neither on-site security officers nor trusted shipping channels, played a key part in meeting these requirements. [14]

However, using trusted hardware as a foundation for real e-commerce applications gives rise to an additional requirement: validating that trust. Many e-commerce efforts cite FIPS 140-1 [11] (even though this U.S. government standard addresses crypto modules, not general-purpose secure coprocessors). Certification to this standard means that the device passed a suite of specific tests [4] administered by an independent laboratory. Level 4 (the most stringent level in the standard) requires that secrets are rendered unavailable in all foreseeable physical attacks—the standard definition of a high-end "secure coprocessor." As of this writing, no device has ever been certified at that level.

Our team plans to meet this trust requirement by submitting our hardware for full certification at FIPS 140-1 Level 4, and (as a research project) carrying out the formal modeling and verification that would be necessary for also certifying the bootstrap software at Level 4. Besides applying formal methods to a sizable, implemented system, this project is challenging because it applies a standard originally crafted for cryptographic modules to programmable secure coprocessors, the flexible platform necessary for e-commerce applications.

This effort involves:

- building a formal model of the configuration of a programmable secure coprocessor
- expressing the security requirements in terms of this model
- translating the bootstrap software and other aspects of system behavior into a set of transformations on this model
- formally verifying that the security properties are preserved by these transformations

This paper presents our initial plan of attack: translating into formal methods the goals and requirements that guided the implementation of our device. We have proceeded to carry out this plan, by building the model, assertions, transformations, and proofs within the ACL2 [7] mechanical theorem prover. This work is nearly complete; follow-up reports will detail our experiences with this modeling, and with the FIPS process.

The Potential of Trusted Hardware The security of electronic commerce applications requires that participating computers and storage devices correctly carry out their tasks. An adversary who modifies or clones these devices can undermine the entire system. However, in many current and proposed electronic commerce scenarios, the adversary may have physical access to machines whose compromise benefits him. These threats include insider attack, attacks on exposed machines, and dishonest users attacking their own machines.

A *secure coprocessor* is a general-purpose computing device with *secure memory* that is protected by physical and/or electrical techniques intended to ensure that all foreseeable physical attack *zeroizes* its contents. Previous work at our laboratory [12, 16, 17, 18] explored the potential of building a coprocessor with a high-end computational environment and cryptographic accelerators, encapsulated in protective membrane and protected by tamper-response circuitry that senses any physical changes to this membrane. Subsequent work at CMU [15, 20] used these prototypes to define and demonstrate the power of the secure coprocessing model—achieving broad protocol security by amplifying the security of a device trusted to keep its secrets despite physical attack. In particular, this model addresses many security problems in electronic commerce applications [21].

Building Trusted Hardware In order to move the secure coprocessing model from the research laboratory into real-world e-commerce applications, our group, over the last two years, has been researching, designing, and implementing a mass-produced high-end secure coprocessor. We needed to accommodate many constraints:

- A device must detect and respond to all foreseeable avenues of tamper.
- We cannot rely on physical inspection to distinguish a tampered device from an untampered device.
- A device must be tamper-protected for life, from the time it leaves the factory—the field is a hostile environment.
- Nearly all software on the device—including most bootstrap, and the operating system—must be remotely replaceable in the field *without security officers, secure couriers, or other trusted physical channels.*
- The different layers of software within any one device must be controllable by different, mutually suspicious authorities—so we need to allow for malicious authorities, as well as *Byzantine failure* of fundamental code (such as the OS or bootstrap).
- Different end-users of the same application (but on different coprocessors) may not necessarily trust each other—and possibly may never have met.
- Nevertheless, an application running on an untampered device must be able to prove, to all concerned, that it's the real thing, doing the right thing.

These constraints were driven by the goal of building a mass-produced tool that enables widespread development of e-commerce applications on secure hardware, much as the earlier generation laboratory prototypes enabled Tygar and Yee's development of research proofs-of-concept [21]. Separate reports discuss the broader problem space [13] and the details of our solution [14] for the resulting commercial product [6].

Trusting Trusted Hardware Clearly, a critical component of such a device is effective tamper response: physical attack must zeroize the contents of secure memory, near-instantly and with extremely high probability. Our protections are based on always-active circuitry that *crowbars* memory when it senses changes to a tamper-detecting membrane, which is interleaved with tamper-resistant material to force attacks to affect the membrane. As widely reported (e.g., [1, 2, 19]), designing effective tamper-response techniques and verifying they work can be tricky; [14] discusses the suite of techniques we use in our device, currently under independent evaluation against the FIPS 140-1 Level 4 criteria [4, 11].

Other important components include the hardware and software design that enable fast crypto performance, and the software programming environment that enables applications to easily exploit these abilities.

However, a *security architecture* must connect these pieces in order to ensure that the tamper response actually did any good. Since developing our architecture required simultaneously addressing several complex issues, we find that presenting this material often leads to the same questions:

- "What does a secure coprocessor *do*?"
- "What is your architecture trying to achieve?"
- "Why does it have so many pieces?"
- "Why should I believe it works?"
- "What if I'm solving a simpler problem, and can eliminate property X?"

As part of verification of our commercial design, as well as preparation for future work in secure commerce systems, we are currently developing formal models (based on the formalism which guided the implementation) in which to frame and answer these questions. This paper presents some preliminary results. Section 2 presents English statements of some of the overall security goals for a programmable high-end secure coprocessor. Section 3 refines these statements in more formal terms. Section 4 enumerates the elements of our architecture that makes these properties hold. Section 5 presents some arguments for conciseness of design, by showing how eliminating elements of the design causes these properties to fail. But Section 6 shows how weakening the security properties can lead to simpler designs.

This paper presents a snapshot of one aspect of the broader efforts required to ensure that trusted hardware can be trusted. Section 7 explores some of this broader picture.

2 English Statements of Security Goals

2.1 System Components

As noted earlier, a generic secure coprocessor consists of a CPU, secure memory (which may be both volatile and non-volatile), potentially some additional non-secure non-volatile memory, and often some cryptographic accelerators. (See Figure 1.) All this is packaged in a physical security design intended to render unavailable the contents of secure memory, upon physical attack. Since exploring the effectiveness of physical tamper response lies beyond the scope of this paper, this analysis simply assumes that the physical tamper response works. However, we stress that as part of meeting the trust requirement of trusted hardware, our device's physical security design is undergoing the extensive independent tests required by FIPS 140-1 Level 4.

In our design, the non-volatile secure memory consists of battery-backed static RAM (*BBRAM*). Hardware constraints limited the amount of BBRAM to 8.5 kilobytes; the business goal required that the device carry within it its own software; and our security model did not require that this software be secret[1] from adversaries. Consequently, our design includes a megabyte of FLASH memory[2] as the

[1] Clearly, this design extends to a model where software is secret, by putting in FLASH a two-part program: one part uses secrets in BBRAM to verify and decrypt the second part.

[2] FLASH memory provides rewritable non-volatile storage; but rewriting involves a lengthy process of erasing and mod-

primary code store; this memory is contained with the secure boundary (so attacks on it should first trigger tamper response) but is not itself zeroized upon attack.

The constraints we faced caused us to partition the code-space in FLASH into three layers: a foundational *Miniboot 1* layer, an operating system layer, and an application layer. We refer to a generic layer as *Layer N*; boot-block ROM code is Layer 0, Miniboot 0. See Figure 2.

For each device, each of these layers may potentially be controlled by a different authority. Articulating and accommodating the nature of this control constituted one of the major challenges of moving our secure coprocessor from a laboratory prototype to a real-world device.

- We had to permit the authority to be at a remote site, not where the card is.
- We had to recognize that, potentially, no one at the card's site may be trustworthy.
- We had to allow different authorities to distrust each other.

For a particular device, we refer to the party in charge of the software in Layer N as *Authority N*. Each layer includes code, as well as space for a public key (of the authority over that layer), and some additional parameters.

We partition the BBRAM into a region for each code layer: *Page N* belongs to Layer N. Given the multitude of potential owners and the potential failure properties of FLASH, we also must include some notion of the *status* of a code layer at any particular time: e.g., "unowned," "owned but unreliable," "reliable but unrunnable," "runnable."

2.2 Desired Properties

Besides accommodating the constraints dictated by business and engineering concerns, our device must make it easy for third party programmers to develop and deploy applications in the style of [21]. This led us to articulate some basic goals for our security architecture.

ifying a *sector* that is tens of kilobytes long, and can only be done a relatively small number of times compared to ordinary RAM.

2.2.1 Control of Software

Suppose A has ownership of a particular software layer in particular untampered device. Then only A, or a designated superior, can load code into that layer in that device—even though the card may be in a hostile environment, far away from A and her superiors.

2.2.2 Access to Secrets

The secrets belonging to layer N are accessible only by code that Authority N trusts, executing on an untampered device that Authority N trusts, in the appropriate context.

2.2.3 Authenticated Execution

It must be possible to distinguish remotely between

- a message from a particular layer N with particular software environment on a particular untampered device
- a message from a clever adversary

However, the adversary could have access to other untampered devices, or even to this one, untampered but with a different software environment.

2.2.4 Recoverability

We must be able to recover from arbitrary failure in any rewritable layer—including the operating system and rewritable Miniboot.

These failures include:

- the layer contents themselves become scrambled
- the program in that layer behaves with arbitrary evil intent

Furthermore, for this recovery to be meaningful, we must be able to confirm that the recovery has taken place.

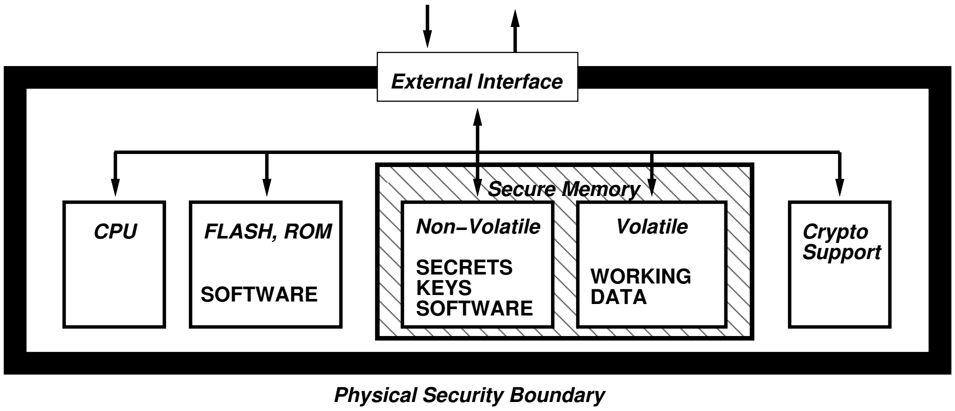

Figure 1: High-level sketch of our secure coprocessor.

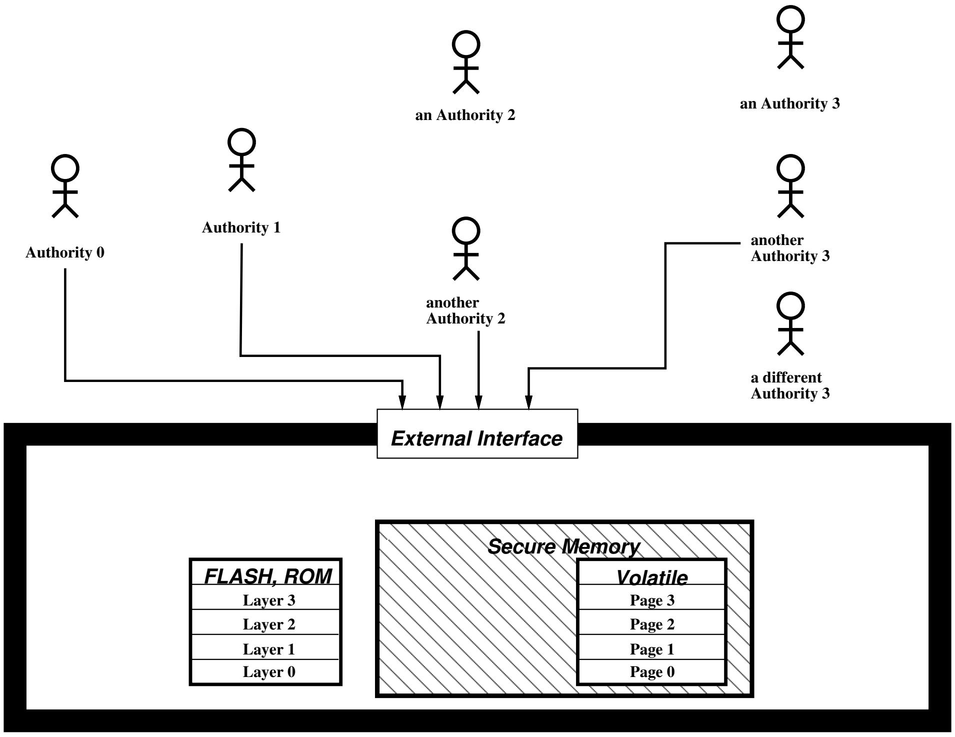

Figure 2: Each code layer has its own subset of BBRAM, and is potentially controlled by a different authority. But different devices may have different authority sets, and the set for any one device may change over time.

2.2.5 Fault Tolerance

Interruption of execution at any point will not leave the device in a condition that violates the above.

2.3 Justification

Although driven by our particular design constraints, these goals might arguably apply to any general-purpose secure coprocessor intended for third party use, and that is rich enough to require code maintenance.

- (Section 2.2.1) A party who deploys an e-commerce application on such a device wants to be sure that others cannot alter his code. Otherwise, the application can be subverted by an adversary "updating" the program to include a backdoor or Trojan horse.
- (Section 2.2.2) The coprocessor must really provide a trusted environment to safely store application secrets. For example, an adversarial application should not be be able to copy or use secret keys, or modify the balance in an e-wallet.
- (Section 2.2.3) It must be possible for a participant in a coprocessor-based e-commerce application to verify that they are interacting with the correct software on an untampered device. Failure of either of these conditions leaves even basic coprocessor applications, such as cryptographic accelerators, open to attack (e.g., [22]).
- (Section 2.2.4) Failures and interruptions should leave us with at least a safe state, if not a recoverable one; otherwise, an adversary can be expected to cause the necessary failures and interruptions. The cost of the device, and the inevitability of errors in complex software, stress recovery from Byzantine action by loaded software.

Although arguable necessary, whether this set of properties is *sufficient* is an interesting avenue for further exploration.

3 Formal Statements of Security Goals

3.1 Model Components

Formalizing these English statements requires introducing some terms.

Layer N, *Authority* N, and *Page* N were already introduced in Section 2.1.

The *system configuration* consists of a tuple of the relevant properties, including:

- a vector of conditions for each layer: its status, owner, code contents, and BBRAM page contents
- whether or not the device has been tampered.
- what other hardware failures may have occurred (e.g., does the DRAM scratch area still work?)

We use standard notation $\pi_N C$ to denote the Layer N component of a configuration C.

We organize the possible values of these layer components into a natural partial order. For any A, "owned by A but unreliable" dominates "unowned." For any A and P, "owned by A with reliable and runnable contents P" dominates "owned by A with reliable but unrunnable contents P," which in turn dominates "owned by A but unreliable."

For each BBRAM page, we need some notion of "initial contents."

For a runnable program at layer N, the *software environment of* N in configuration C consists of the programs and status of the components $\pi_K C$ for $0 \leq K \leq N$. We denote this by $ENV_N C$. Our intention here is that the correct and secure execution of the layer N code on an untampered device depends only on the correct and secure execution of the code in its software environment.

Initial State. In our design, devices are *initialized* at the factory and left in a configuration where only Layers 0 and 1 are installed, and each of these has a self-generated random secret in its BBRAM page, with a corresponding certificate stored in the FLASH layer. For Layer 1, this is just an RSA keypair, with public key signed by the Factory Certificate Authority (CA); for Layer 0 (which, as ROM, does not contain public-key code), this consists of a set of DES keys, and a certificate consisting of the encryption and signature of these keys by the privileged Authority 0.

Transitions. The configuration of a device can change due to several potential causes:

- explicit configuration-changing commands, which (in our design) are carried out by the Miniboot layers;
- failures of hardware components, BBRAM and FLASH (discussed further in Section 3.2.5 below)
- tamper events
- ordinary operation of the device, and execution of the code layers.

We model this notion by defining the set of *valid transitions* on configurations, following the above causes. A *valid configuration* is one that can be reached from a valid initial configuration by a sequence of zero or more valid transitions. We frequently discuss a sequence of configurations $C_0 \ldots$ with C_0 being some particular valid configuration and with each C_{i+1} reachable from C_i by a valid transition.

Tamper. With our earlier assumption that the physical tamper response actually works, perhaps the most natural characterization of tamper is "the device doesn't have its secrets any more." However, this characterization of the effect of tamper-response on secrets overlooks some critical issues:

- The device's secrets may have been copied off-card before tamper, due to exploitation of some error in memory management, and be restored later. (For example, the OS might have a bug that permits a hostile application to download the entire contents of secure memory.)
- The secrets belonging to *another* device may be loaded into this one, after tamper.
- The device's secrets may still be present after tamper, perhaps due a malicious code-loading command that tricked the device into thinking that the secret storage area contained the new code to be burned into FLASH.

As far as secrets go, the only thing we can say for certain is that a tamper event transforms the device configuration by destroying the contents of BBRAM. Immediately after the event, the secrets are no longer in that location.

However, the potential for tamper caused further headaches for formal analysis. The possible configuration transformations for a device are governed by its current code layers, and the physical construction of the device. (Indeed, it is the physical construction which causes the tamper response to occur.) Physical tamper can change these properties—and thus arbitrarily change the possible transformations and behaviors after tamper.

Clearly, reasoning about whether a remote party can authenticate an untampered device requires reasoning about tamper. However, our initial work also found it necessary to include "untamperedness" as an assumption for many other properties—since without it, no memory restrictions or other useful behaviors can hold with any certainty. This is the reason we ended up including "memory of being tampered" as an explicit element in the system configuration: to distinguish traces that have ventured into this terrain, from traces that have remained safe.

Authentication. We need to consider scenarios where needs to determine whether a particular message M came from a particular A, or from someone else, and we consider authentication schemes based on some secret possessed by A.

For such secret-based schemes, we define the *necessary trust set* to consist of those parties who, if they abuse or distribute their secrets, can make it impossible for B to make this distinction. For example, in a standard public-key scheme with a single CA, the necessary trust set would include both A, as well as the root CA. (In a scheme with multiple certificate authorities, the trust set would include the intermediate certificate authorities as well.)

3.2 Goals

We now attempt to express the goals of Section 2 more formally. (Recall that Section 3.1 discussed the initial conditions of the device.)

3.2.1 Control of Software

Formal Statement Suppose:

- an untampered device is in valid configuration C
- $N \geq 0$
- $AUTH$ be the set of authorities over $\pi_K C$, for $0 \leq K \leq N$.
- a sequence of valid transitions takes C to some C', where the device is still untampered

If $\pi_k C'$ does not precede or equal $\pi_k C$ in the layer partial order, then:

- at least one transition in the sequence from C to C' was caused by a configuration-changing command
- at least the first such command in this sequence required knowledge of a secret belonging to a member of $AUTH$

English An authority must be able to control his layer, and (under appropriate conditions) a superior authority should be able to repair that layer.

However, we also need to recognize the fact that failures of hardware or "trust invariants" also affect a layer's configuration, even without the action of any of these authorities.

The formal statement above attempts to express that one's layer can be *demoted*, due to failure or other reasons, but any change that otherwise modifies the contents must be traceable to an action—now or later—by some *current* Authority $K \leq N$.

3.2.2 Access to Secrets

Formal Statement Suppose:

- A is the authority over a layer N in some untampered device in valid configuration C_0
- In $\pi_N C_0$, Page N has its initial contents, but no program in $ENV_N C_0$ has yet executed since these contents have been initialized.
- T_A is the set of software environments for N that A trusts.

Let $C_0, ..., C_i, ...$ be a valid sequence of configurations, and let C_k be the first in this sequence such that one of the following is true:

- The device is tampered in C_k
- Some $\pi_M C_k$, for $M \leq N$, is not runnable
- $ENV_N C_k \notin T_A$

Then:

- The contents of Page N are destroyed or returned to their initial state in C_k.
- For all $0 \leq i < k$, only the programs in $ENV_N C_i$ can directly access page N.

English This formal statement expresses that once an authority's program establishes secrets, the device maintains them only while the supporting environment is trusted by that authority—and even during this period, protects them from lower-privileged layers.

The overall motivation here is that authority A should be able to trust that any future from configuration C_0 is safe, without trusting anything more than $ENV_N C_0$—his environment right now.

In particular, we should permit A to regard any of the following to cause ENV_N to stop being trustworthy:

- Some C_i to C_{i+1} transition might include a change to the code in layer $K < N$, which A does not trust. (Your parent might do something you don't trust.)
- Some $\pi_K C_i$, for $K > N$ and $i \geq 0$, might try to attack layer N. (Your child might be try to usurp your secrets.)

- Some other authority in any C_i, $i \geq 0$, might behave clumsily or maliciously with his or her private key—e.g., to try to change what the device believes is A's public key. (Your parent might try to usurp your authority.)

3.2.3 Authenticated Execution

Formal Statement Suppose Bob receives a message M allegedly from layer N on an untampered device with a particular $ENV_N C_k$, for valid configuration C_k.

Then Bob can determine whether M came from this program in this environment, or from some adversary, with a necessary trust set as small as possible.

In particular, Bob should be able to correctly authenticate M, despite potential adversarial control of any of these:

- new code in any layer in C_j, for $j > k$;
- code in any layer in some alternate configuration C' that follows from some C_j via a sequence of valid transformations, one of which is tamper;
- code in any layer $L > N$ in C_k;
- old code in any layer $L > 1$ in C_j for $j < k$.

English This formal statement basically restates Section 2.2.3, while acknowledging that a clever adversary might load new code into the device, including new bootstrap code, or control code already in a lower-privileged layer in that device, or have controlled code that used to be even in layer N, but has since been replaced.

3.2.4 Recoverability

Formal Statement Suppose:

- an untampered device is in configuration C
- $N > 0$,
- $AUTH$ is the set of authorities for 0 to $N-1$ in C.
- $\pi_0 C$ through $\pi_{N-1} C$ are runnable
- $\pi_N C$ is arbitrary (and may have attempted to cause unauthorized configuration changes through its execution)
- P_N specifies proposed new contents for layer N.

Then there exists a sequence of configuration-changing commands, from the members of $AUTH$, that takes C to C' where

- $\pi_k C = \pi_k C'$, for $0 \leq k < N$
- $\pi_N C = P_N$.
- the device is still untampered in C'
- the members of $AUTH$ can verify that these properties of C' hold

English The formal statement basically says that, if layer N is bad but the higher-privileged layers are good, then the authorities over these layers can repair layer N.

The final condition expresses a necessary but often-overlooked aspect of code-downloading: the ability to repair a device in a hostile environment is often not meaningful, unless one authenticate that the repair actually took place. Did the new code ever get there? (Any secret used to authenticate this fact must not be accessible to the faulty code.)

3.2.5 Fault Tolerance

The above statements asserted properties of untampered devices in a configuration reachable from an initial configuration via a sequence of valid transformations.

To achieve the real-world goal of fault tolerance, it is important that our model be as complete as possible: these transformations must include not just configuration-changing commands and ordinary operation, but also:

- improperly formatted but authentic commands
- early termination of command processing
- as many "hardware failures" (e.g., FLASH storage failures) as possible

Unlike the other desired property, this goal is best expressed as a meta-statement: the other properties hold, even when the model is expanded to cover these conditions.

However, in the process of carrying out the plan outlined in this paper, we discovered that this requirement leads to quite a few subtleties. For example, we did easily extend our initial formal model to express abnormal termination, but we found it inadequate to address improper formatting. (These issues are potentially amenable to formal models, but not with the level of abstraction we chose.) Consequently, we resorted to a separate, systematic code analysis to address improper formatting.

3.3 Independence

The initial statements of Section 2 came almost directly from engineering requirements. We note however that, now rendered more formally, the goals appear to remain independent.

For example, "Control of Software" does not imply "Recoverability." A scheme where software could never change, or where software could only change if the public key of that layer's owner was contained in the FLASH segment, would satisfy the former but not the latter.

Conversely, a scheme where *any* layer owner could change layer N establishes that "Recoverability" does not imply "Control of Software."

This remains an area for further investigation, particularly with the mechanical theorem prover. Exploring whether this set of necessary properties could be expressed in a smaller, more concise form could be an interesting problem.

4 Correctness

In the previous section, we attempted to render statements of the important security properties in more formal terms. In this section, we now try to examine the assumptions and proof strategies that can verify that the architecture meets these goals.

4.1 Architecture Components

Examining why our architecture might achieve its goals first requires articulating some common preliminary subgoals.

4.1.1 Boot Sequence

Reasoning about the run-time behavior of the device requires addressing the issue of who is doing what, when.

Our architecture addresses this problem linking device-reset to a *hardware ratchet*:

- a *ratchet* circuit, independent of the main CPU, tracks a "ratchet value" *ratchet*
- hardware ensures that the *reset* signal forces CPU execution to begin in Miniboot 0 boot-block ROM, and forces *ratchet* to zero.
- software on the CPU can increment the ratchet at any point, but only reset can bring it back to zero.

The intention is that each layer N advances the ratchet to $N+1$ before first passing control or invoking layer $N+1$; the ratchet then controls access control of code to critical resources.

We model this by introducing an *execution set* of programs that have had control or have been invoked since reset. Hardware reset forces the execution set to empty; invocation or passing control adds a program to the set.

4.1.2 Hardware Locks

Who does the hardware permit to access critical memory?

For purposes of this paper, we simplify the behavior of the ratchet to restrict access only to FLASH and BBRAM, according to the following policy:

- The protected FLASH segments can only be written when $ratchet \leq 1$.
- Page N in BBRAM can be read or written only when $ratchet \leq N$.

We stress this is enforced by hardware: the CPU is free to issue a forbidden request, but the memory device will never see it.

4.1.3 Key Management

How does the outside world know it's hearing from a real device?

As noted earlier, Miniboot 1 generates a keypair during factory initialization; the private key is retained in Page 1 and the public key is certified by the factory CA. Miniboot 1 can also regenerate its keypair, certifying the new public key with the old private key. In our design, Miniboot 1 generates a keypair for use by Layer 2; the private key is left in Page 2, the public key is certified by Miniboot 1, and the keypair is regenerated each time Layer 2 is altered. Lacking public key code, Miniboot 0 possesses a set of random DES keys used for mutual secret-key authentication with Authority 0.

Section 4.2.3 and Section 5.2 below discuss this in more detail, as does the architecture report [14].

Both the BBRAM and the FLASH locks play a critical role in this solution: the BBRAM locks ensure that the secrets are accessible only by the right layers, and the FLASH locks ensure that the right layers are what we thought they were.

4.1.4 Authentication Actually Works

How does the device know it's hearing from the right party in the outside world?

Our design addresses this problem by having each Authority N ($N \geq 1$) generate an RSA keypair and use the private key to sign commands; Miniboot 1 verifies signatures against the public key currently stored in Layer N. (With no public-key code in Miniboot 0, Authority 0 uses secret-key authentication.)

However, reasoning that "Miniboot accepted authenticated command from Authority N" implies "Authority N signed that command" requires many additional assumptions:

- the intractability assumption underlying cryptography are true;
- keys are really generated randomly;
- the authorized owner of a private key actually keeps it secret;
- Miniboot 0 does not leak its secret keys; and
- Miniboot correctly carries out the cryptographic protocols and algorithms.

4.1.5 Code Actually Works

Throughout this work (such as in the last item in the previous section), we continually need to make assertions about "the code actually works."

For example:

- reasoning about the identity of ENV_N in some configuration C requires assuming that the code in previous configurations correctly followed the policy of updating code layers;
- reasoning about the behavior of ENV_N in some configuration requires assuming that ENV_{N-1} correctly evaluated and responded to hardware failures and other status changes affecting Layer N, and actually passes control to Layer N when appropriate;
- asserting that only ENV_N can access Page N requires not just the BBRAM locks, but also the assumption that each Layer $K < N$ correctly incremented the ratchet (as well as the assumption that ENV_N is and does what we thought)

Proving these assertions will require careful mutual induction: e.g., establishing that code layers change only through the action of Miniboot requires first establishing that Miniboot hasn't changed in an unauthorized way.

Given the fact that Miniboot 1 itself can change, this means that most trust assertions about the architecture will follow the schema "believing X is true for the system, from now on, requires believing that Miniboot does Y right now."

4.2 Assumptions for Goals

4.2.1 Control of Software

Establishing that the architecture achieves the "Control of Software" goal requires establishing, as noted above, that only Miniboot, executing as Miniboot can change the FLASH code layers; that authentication of commands works correctly; and then tracing through the possible transformations to show that:

- demotion of status can occur via failure response;
- any other transition requires an authenticated command; and
- the authority issuing this command must have been in the authority set at C.

4.2.2 Access to Secrets

Establishing that the architecture achieves the "Access to Secrets" goal requires the lemma, noted in Section 4.1.5 above, that only $ENV_N C$ can access Page N in an untampered device, and then showing that all configuration transitions preserve the invariant:

> If C_i differs from C_{i-1} for at least one of the following reasons:
>
> - the device is tampered
> - $ENV_N C_i$ stopped being fully runnable
> - $ENV_N C_i \notin T_A$
>
> then C_i also differs in that the contents of Page N are cleared or returned to initial state.

(Departure from T_A is, from Layer N's perspective, essentially equivalent to tamper.)

Enforcing this invariant requires developing some reasonable way that the device can determine whether or not a new configuration is still a member of T_A. We adopted some simple trust parameter schemes to characterize the detectable T_A, then further reduce this detectable set by requiring that

- the device itself must be able to confirm that a new $ENV_N \in T_A$
- by *directly* verifying its signature against a reliable public key currently in Layer N
- *before* the change occurs

The fact that this invariant is enforced by Miniboot code that is itself part of ENV_N, and subject to untrusted change, complicates this implementation. If the C_k transition involved the loading of an untrusted Miniboot, we ensure that trusted Miniboot code currently part of $ENV_N C_{k-1}$ ensures that Page N secrets are erased before the transition succeeds. The fact that, if A does not trust a new Miniboot in C_k, it cannot trust Miniboot to correctly carry out authentication in C_k and the future, leads to some additional protocol considerations.

4.2.3 Authenticated Execution

Basically, our scheme (Section 4.1.3) binds a public key to certain software environment, and then confines the private key to that environment.

Suppose Bob is trying to authenticate a message from layer N, in C_k. For simplicity, let suppose $0 \leq N \leq 2$. (The argument extends to the 3-layer schemes discussed in [14].)

The hierarchical nature of code layers, coupled with the fact that each layer $N > 0$ is replaceable, gives a two-dimensional history of code changes. We can extract this history from the configuration sequence $C_0, ...C_i...$ undergone by an untampered device: each $\pi_N C_i$ depends on $\pi_{N-1} C_i$, and is succeeded by $\pi_N C_{i+1}$.

These two dimensions create two dimensions for spoofing: in one axis, some $\pi_j C_k$ might have sent the message, for $j \neq N$; in the other, some $\pi_N C_j$ might have sent the message, for $ENV_N C_j \neq ENV_N C_k$.

If $N = 0$, the preliminary subgoal that only ENV_N sees Page N gives the result—with Bob's necessary trust set consisting of Miniboot 0, $\pi_1 C_0$ (since Miniboot 1 at the factory participates in initialization) and the Authority 0/Factory CA.

If $N = 1$, the preliminary subgoal coupled with the regeneration policy, gives the result—with Bob's necessary trust set consisting of Miniboot 0, the Factory CA, and each version of Miniboot 1 from $\pi_1 C_0$ up to $\pi_1 C_k$. (The fact that any code-changing action of Authority 0 requires repeating factory certification removes Authority 0 from this set).

If $N = 2$, the preliminary subgoal along with the layer 2 key policy gives the result—with Bob's necessary trust set consisting of the $N = 1$ set, along with $\pi_2 C_k$.

This necessary trust set for $N = 1, 2$ is arguably minimal:

- a secret-based scheme forces the factory CA, and $\pi_1 C_0$ (the on-card code that participates in initialization) to be in the set;
- $\pi_N C_k$ must be in the set
- the on-card components in the set must be "connected": some certification path must exist from the initializer $\pi_1 C_0$ to $\pi_N C_k$
- from hierarchical dependence, the set must be "bottom-closed": if $\pi_j C_i$ is in the set, then so must be $\pi_{j-1} C_i$.

4.2.4 Recoverability

Establishing Recoverability follows from the subgoals that only ENV_{N-1} sees Page $N-1$, that only Miniboot changes layers, that permanent Miniboot 0 can correctly replace Miniboot 1 when appropriate, and that the boot sequence always gives Miniboot a chance to authenticate commands. Confirming a successful change follows from the Authenticated Execution property, for Miniboot.

4.2.5 Fault Tolerance

Establishing that the architecture meets the Fault Tolerance goal follows from careful code design. Code that already works tests for (and responds to) hardware failures. Configuration transitions need to be structured so that, despite interruptions and other failures, the device is left in a clean, predictable state.

However, various constraints forced our design to depart from standard *atomicity*—a change fails completely or succeeds completely—in a few subtle ways.

First, hardware constraints permit us to have redundant FLASH areas only for Layer 1—so reburns of Layer 2 or 3 force the device first to erase the entire Layer, then reburn it. We handle such destructive transitions by implementing a sequence of two transitions, each of which is atomic. The intermediate failure state is taken to a safe state by the transformation performed by clean-up at the next boot.

Second, some failures can leave the device in a fairly odd state, that requires clean-up by execution of code. We handle these situations by having Miniboot 0 in ROM enforce these clean-up rules, and using the boot sequence subgoal to argue that this clean-up happens before anyone else has a chance to perceive the troubled state. This complicates the formal analysis: atomicity of configuration change does not happen to device configurations *as they exist in real time*, but device configurations *as they can be perceived*.

Our design permits the code authorities to specify what family of untampered devices (and with what software environments) should accept a particular code-loading command. These target features provide the hooks for authorities to enforce the serializability and compatibility rules they find important.

5 Conciseness

We argue for conciseness of design by considering two aspects: code loading, and authenticated execution.

5.1 Code Loading

Our code-loading involves the "Control of Software," "Access to Secrets" and "Recoverability" goals.

The correctness of our scheme follows from a number of items, include the ratchet locks on FLASH, the ratchet locks on BBRAM, and the trust parameters for what happens to Page N when something in ENV_N is changed.

If our scheme omitted FLASH locks, then we could no longer be sure what layer code is carrying out code changes, and what's in the contents of the code layer that is supposed to be evaluating and carrying out the changes.

If our scheme omitted BBRAM locks, then we would not be able to authenticate whether a change has actually occurred—so, for example, we could never recover from a memory-manager vulnerability in a deployed operating system.

If our scheme forced all code changes to erase *all* BBRAM, then we would lose the ability to authenticate an untampered device while also performing updates remotely in a hostile field—since the device, after the code change, would not be able to prove that it was the same untampered device that existed before the change, one would have to rely on the testimony of the code-loader.

If we did not treat untrusted changes to Miniboot 1 differently from the other layers, then we would force authorities to be in the inconsistent situation of relying on code they no longer trust to correctly evaluate statements about what they do or do not trust.

5.2 Authenticated Execution

Our outgoing authentication scheme (Section 4.1.3) forces recipients of a message allegedly from some untampered $\pi_N C_k$ to trust $ENV_N C_k$, $\pi_i C_1$ (for $0 \leq i \leq k$), and the Factory CA. But, as noted earlier, this trust set is arguably minimal: and if any party in this set published or abused its secret keys, then, with this standard PKI approach to authentication the recipient could *never* ascertain whether or not the message came from $\pi_N C_k$.

If the device Page 1 keypair was not regenerated as an atomic part of Miniboot 1 reloads, then it would be possible for new code in $\pi_1 C_{k+1}$ to forge this message. Regeneration protects against attack by future, evil versions of Miniboot 1.

If we did not have a separate keypair for Layer 2, then we'd either have the inefficiency of forcing Layer 2 to reboot the device in order to get a signature (and have Miniboot 1 carefully format what it signs on behalf of Layer 2), or leave the private key outside of Page 1 and have the vulnerability of $\pi_2 C_k$ forging messages from $\pi_1 C_k$.

If we did not regenerate the Layer 2 key with each ENV_2 change, then we'd permit $\pi_2 C_{k+1}$ and $\pi_2 C_{k-1}$ to forge messages from $\pi_2 C_k$. (For one example, suppose Authority 2 releases a new operating system to fix a known hole in the old one. Even though the fixed version should retain other operational secrets, not forcing a change of the keypair permits the buggy version to impersonate the fixed version. Conversely, if Authority 2 mistakenly introduces a hole with a new version, not forcing a change of the keypair permits the new version to impersonate the old.)

If we did not have the BBRAM locks, then $\pi_j C_k$ could forge messages from $\pi_N C_k$ for general $j > N$. If we did not have the FLASH locks, then any $\pi_j C_i$ for $0 \leq i \leq k$ might be forging this message.

6 Simplifications

We developed our security architecture to support a secure coprocessor with a specific set of constraints. However, the flexibility and security goals for our "high-end" product forced us (for the most part) into a difficult situation. (Our one "easy way out" was the freedom *not* to spontaneously load arbitrary, mutually suspicious peer applications.)

To put it succinctly, our design is an arguably minimal solution to an arguably pessimal problem. Simplifying the problem can certainly simplify the solution. For example:

- Forcing any configuration change to ENV_N to kill Page N—except for the outgoing authentication keys—would simplify enforcement of Access to Secrets.
- Only allowing for one code authority—or for a family of mutually trusting authorities—would simplify authentication and trust.
- If we assume the OS will never have any memory access vulnerabilities, then we could eliminate FLASH locks and BBRAM locks: memory management would be enforced by code we trusted.
- If Miniboot, and possibly the OS, were never to be changed, then the key management and secret-access schemes can be greatly simplified.

We anticipate that many of these simplifications may arise if this architecture were mapped to a smaller device, such as a next-generation smart card.

7 Related and Future Work

Much previous and current work (e.g., [3, 7, 9]) explores the use of formal methods as tools to examine the basic question of: *does the design work?* Many recent efforts (e.g., [5, 8, 10]) apply automated tools specifically to electronic commerce protocols.

As noted earlier, this paper reports our initial strategy for formal verification of our existing implementation of an e-commerce tool. Having refined the design goals into statements about a formal model, the next step is to express and verify these properties with a mechanical verification tool—and to, in accordance with FIPS 140-1 Level 4, rigorously document how the model and transformation system correspond to the actual device and its code. This work is underway, as is independent verification of the physical security of our device.

However, we have found that convincing users that our trusted hardware can indeed be trusted also requires addressing additional issues. For example:

- *Does the implementation match the design?* History clearly shows that "secure" software is fraught with unintended vulnerabilities. In our work, we address this with several techniques:
  - evaluation by independent laboratories (e.g., as part of FIPS certification)
  - continual consultation and evaluation by in-house penetration specialists
  - following general principles of careful coding design such as clearing the stack, and safely tolerating input that is deranged (e.g., negative offsets) even if authenticated
- *Does the product, when purchased, match the implementation?* Our policy of "tamper-protection for life" protects the device when it leaves the factory; we have procedures in place to address potential attack before that point.

The existence of flexible, powerful and trusted secure coprocessors can help secure computation in untrusted environments. This research is one part of our group's broader efforts to achieve this security by making these tools available.

Acknowledgments

The authors gratefully acknowledge the contributions of entire Watson development team, including 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, Steve Weingart, 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, Matthias Kaiserswerth, Mike Matyas, and the referees for their helpful advice, and Bill Arnold, Liam Comerford, Doug Tygar, Steve White, and Bennet Yee for their inspirational pioneering work.

Availability

Hardware Our secure coprocessor exists as the IBM 4758, a commercially available PCI card. (Additional research prototypes exist in in PCMCIA format.)

Software Toolkits exist for independent parties to develop, experiment with, and deploy their own applications on this platform. In addition, application software available from IBM transforms the box into a cryptographic accelerator.

Data More information—including development manuals—is available on the Web:

`www.ibm.com/security/cryptocards/`

References

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

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

[3] E. M. Clarke and J. M. Wing. "Formal Methods: State of the Art and Future Directions." *ACM Computing Surveys.* 28: 626-643. December 1996.

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

[5] N. Heintze, J. D. Tygar, J. M. Wing, H. C. Wong. "Model Checking Electronic Commerce Protocols." *The Second USENIX Workshop on Electronic Commerce.* November 1996.

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

[7] M. Kaufmann and J. S. Moore. "An Industrial Strength Theorem Prover for a Logic Based on Common Lisp." *IEEE Transactions on Software Engineering.* 23, No. 4. April 1997.

[8] D. Kindred and J.M. Wing. "Fast, Automatic Checking of Security Protocols." *The Second USENIX Workshop on Electronic Commerce.* November 1996.

[9] C. Meadows. "Language Generation and Verification in the NRL Protocol Analyzer." *Proceedings of the 9th Computer Security Foundations Workshop.* 1996.

[10] C. Meadows and P. Syverson. "A Formal Specification of Requirements for Payment Transactions in the SET Protocol." *Proceedings of the Second International Conference on Financial Cryptography.* Springer-Verlag LNCS. To appear, 1998.

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

[12] 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.

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

[14] S. W. Smith, S. H. Weingart. *Building a High-Performance, Programmable Secure Coprocessor.* Resarch Report RC21102. IBM T.J. Watson Research Center. February 1998. (A preliminary version is available as Resarch Report RC21045.)

[15] 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.

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

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

[18] 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.

[19] 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.

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

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

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