Application Support Architecture for a High-Performance, Programmable Secure Coprocessor

Joan Dyer joandy@us.ibm.com
Ron Perez ronpz@us.ibm.com
Sean Smith sean@watson.ibm.com
Mark Lindemann mjl@us.ibm.com

Secure Systems and Smart Cards
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598-0704

Abstract

A "secure system" should be *secure*—but should also be a *system* that achieves some particular functionality. A family of secure systems that our group has been investigating (and building) are high-end *secure coprocessors:* devices that combine a general-purpose computing environment with high-performance cryptography inside a tamper-responding secure boundary. With the appropriate application software, such secure coprocessors can solve security problems that otherwise would be difficult or impossible.

In this paper, we examine a high-end secure coprocessor as a system: the programming environment it must provide to support such on-card applications; the software and hardware architecture we developed and implemented to provide this support; and some of the lessons we learned from this development.

This paper is not just an academic exercise, but a case study of commercial research and development (leading to a released product, the IBM 4758 [4]).

Proceedings, 22nd National Information Systems Security Conference. October 1999.

1. Introduction

A "secure system" needs to be *secure* against some specified attack set—but it also needs to be a *system* that provides some particular functionality.

A family of secure systems that our group has been investigating (and building) are general-purpose *secure coprocessors*: devices that combine a *general-purpose* programming environment with high-performance cryptography, but can resist (and respond to) a wide variety of physical and logical attacks. Such devices can be trusted to carry out their operations despite a hostile environment (which may even include the host computer); with the right application software, such devices can solve security problems that otherwise would be difficult or impossible (see [7, 16, 17] for some examples).

Previous reports on this work have focused exclusively on *security:* security architectecture problems [9] and solutions [11]; physical security [15]; FIPS 140-1 Level 4 validation [10].

In contrast, this paper focuses on the *system* itself. It is easy to speculate about the programming environment and services that such devices should offer to make these on-card applications possible. However, actually *building* such a support architecture leads to a number of challenges and subtleties:

- in specifying the environment,
- in ensuring the underlying hardware can support this environment,
- and in developing and testing the software that provides this environment.

This paper presents our experiences in designing and implementing an application support architecture for a commercial high-end secure coprocessor.

Target The traditional model of a cryptographic module (e.g., [6]), protects cryptographic keys and algorithms within a secure perimeter. In contrast, a general-purpose secure coprocessor moves beyond this traditional model to also protect non-cryptographic data (such as meter balance) and non-cryptographic algorithms. As a fundamental property, such devices must offer fairly complex programmability—for ever-evolving cryptographic algorithms, for more advanced protocols that build on basic cryptography, or even for security-relevant algorithms and applications that have very little to do with cryptography. Hence, such a device must have both a general-purpose CPU (for the software algorithms) while also having cryptographic hardware to avoid tying up the limited resources of this on-board CPU. Our coprocessor is a PCI card, with ample computational power (a 486-class CPU, megabytes of memory) and cryptographic acceleration: modular math, DES, and hardware random number generation; see Figure 1. (More advanced hardware adding 3DES and SHA-1 is in development.)

We wanted this device to be a general-purpose platform that is sufficiently flexible to support the full spectrum of current and projected secure coprocessor applications. Minimally, it needed to support an application that transformed this device into an accelerator for the *Common Cryptographic Architecture (CCA)* API [1]. It also needed to allow any future applications to be potentially validated against FIPS 140-1 (the US standard for secure cryptographic modules).

This plan led to three goals:

- *security:* a non-tampered card should always be able to prove its authenticity and its software configuration;
- *programmability:* different instances of the same basic platform should be customizable by third-party application developers;
- *application support*: the device's computational and security features can be effectively used by these applications.

This paper focuses on how we addressed the third goal, application support.

The security and programmability goals led to the layered software architecture shown in Figure 2. But the application support goal involved crafting an API and software architecture for Layer 2—and ensuring the underlying software layers and hardware could support it. This supervisor-level "helper" layer would offer services that simplify the development process for user-level Layer 3 applications; such services should include:

- a programming environment,
- communication with the outside world,
- secure data storage,
- cryptography,
- and (when appropriate) debugging tools.

This helper Layer 2 must also isolate Layer 3 from the underlying hardware complexity.

Furthermore, the design of this support architecture must, where possible, also speed development of the support code itself—since this project was part of a product release (the IBM 4758 [4]) driven by very real market deadlines.

Overview of Application Support Architecture To achieve these goals, we refined the basic structure of Figure 2 with the application support architecture of Figure 3. Section 2 presents the secure loading and hardware protection of our configuration control software, which permits safe use of the basic platform for development and debugging. Section 3 presents the *kernel* that provided the foundation for this architecture. Section 4 through Section 8 present the other *managers* that comprise the Layer 2 software. Section 9 presents how these pieces work together to provide a programming environment for applications. Section 10 presents our experiences in integrating these pieces together. Section 11 discusses some ongoing work at evaluating and refining this architecture.

2. Secure Bootstrap

Developing application software for a physically encapsulated, secure device raises a fundamental question: how does the developer get software into the device? Further consideration of this problem leads to more subtleties, particularly when business constraints dictate that

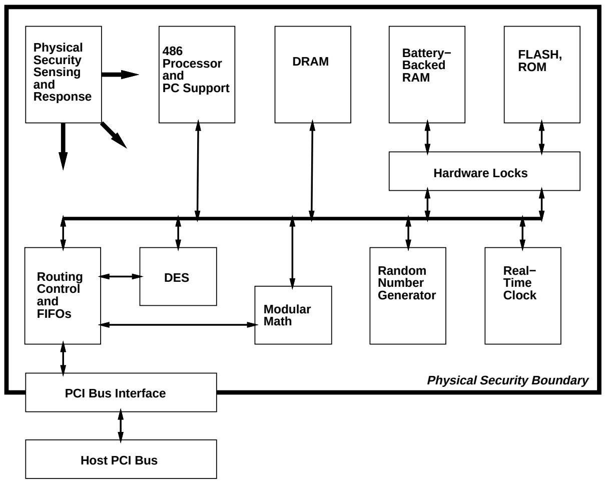

Figure 1 The hardware architecture of our current-generation device.

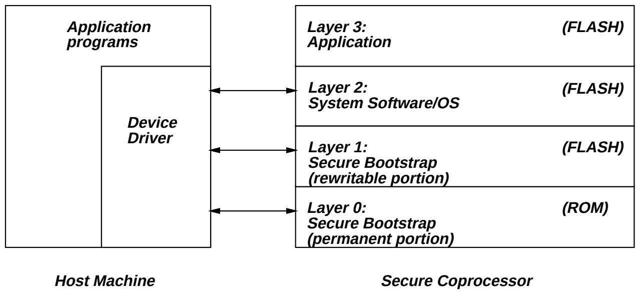
Application programs
Device Driver
Layer 3: Application (FLASH)
Layer 2: System Software/OS (FLASH)
Layer 1: Secure Bootstrap (rewritable portion) (FLASH)
Layer 0: Secure Bootstrap (permanent portion) (ROM)
Host Machine
Secure Coprocessor

Figure 2 The layered software architecture within our secure coprocessor.

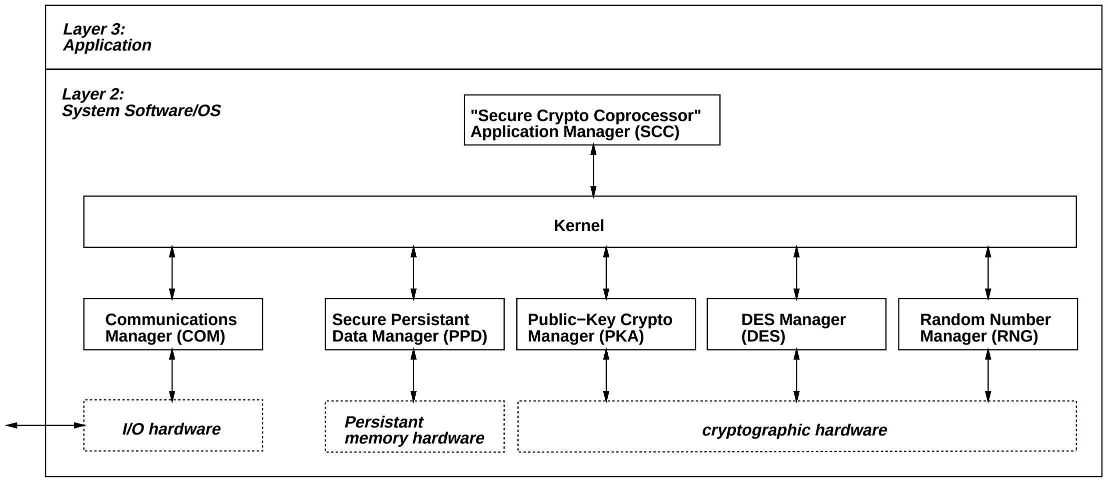
Layer 3: Application
Layer 2: System Software/OS
"Secure Crypto Coprocessor" Application Manager (SCC)
Kernel
Communications Manager (COM)
Secure Persistant Data Manager (PPD)
Public–Key Crypto Manager (PKA)
DES Manager (DES)
Random Number Manager (RNG)
I/O hardware
Persistant memory hardware
cryptographic hardware

Figure 3 The application support architecture within Layer 2.

one type of off-the-shelf device must support a wide variety of development and deployment scenarios, including maintenance of software in the hostile field and authentication of executing software [9].

To address these issues, we developed our *Miniboot* security bootstrap software that resides in ROM and Layer 1 FLASH, and runs at boot-time [11]. Miniboot controls device configuration, and enables any particular device to be configured as "development" without risking contamination of live, production devices. Our approach separates layer ownership from layer contents, and thus allows safe testing of development software with the production-level Layer 2. The flexible loading structure also allows "hot" substitution of different versions of Layer 2 and Layer 3.

For example, the developer might first configure the device for development, and then install a debug Layer 2. He can then iterate loads of Layer 3 (opting to preserve state, if that assists debugging). When the developer is ready to test a near-final application, he can switch from the debug Layer 2 to the real one (and then switch back, if more bugs show up). But all the time, this card is in "development" mode and cannot impersonate a live, production card.

The constraint that Miniboot function correctly without *any* assumptions about the behavior of the code in Layer 2 and Layer 3 led to some early hardware re-design; in particular, the use of proprietary *hardware locks* to ensure integrity of Miniboot code, data, and keys, while still allowing Layer 2 to have full supervisor ("ring 0") access to the 486-class CPU [11].

Development of Miniboot itself raised some challenges. Because it needs to talk with the outside world and use internal memory and cryptographic hardware, Miniboot needs many of the same services as the applications. We addressed this need by equipping Miniboot with simplified versions of the Layer 2 components—sometimes even compiled from the same source files.

3. Kernel

Application development requires a programming environment. The support code providing specialized hardware services, such as cryptography, requires a programming environment, and *also* requires privileged access to the appropriate internal hardware devices.

To address these problems, we designed our Layer 2 around the foundation of a *kernel* that provides the programming environment necessary both for Layer 3 (which runs at the least-privileged level) as well for as the various additional Layer 2 components (which run at the same privilege level as the kernel). We decided to build from a pre-existing kernel, since developing one from scratch would not fit within our implementation timeframe. This kernel should:

- provide separation between *address spaces* for different computational entities;
- provide for multiple threads of execution, even within the same address space;
- provide dynamic-build configurability, to facilitate parallel development of both user-level application code and supervisor-level device drivers and managers;
- provide debugging tools for both user and supervisor code; and
- work with standard tools (compiler, linker, etc.).

After much consideration, we chose CP/Q, a mature OS for industrial embedded systems, because it met all these requirements. CP/Q also had a small footprint, good performance, and was based on message-passing (which was more appropriate than shared-memory for the communication paradigm we foresaw—although CP/Q does not preclude sharing memory for long-lived interactions). Furthermore, we had access not just to its source code, but to the expertise of its development team as well.

This kernel itself provides a rich programming environment (e.g., private address spaces, multiple execution threads) to the Layer 3 application. Furthermore, the kernel also provides these properties to other Layer 2 entities. Thus, this kernel enabled us to partition the additional required services into related groups, and then independently develop *managers* (independent code modules, each in their own address space) to implement each group.

Using a well-tested kernel with well-tested kernel-level tools helped in more than just developing the managers and applications that ran on top of the kernel—it also helped with developing the Miniboot security software that (ordinarily) would run *before* the kernel. Using simulations of the various hardware devices, we ran development versions of Miniboot on top of the kernel—and gradually replaced simulations with direct calls to the "metal."

OS Security Although the CP/Q kernel was neither designed nor tested for resilience against *malicious* application-level code, this drawback is not an issue with its use in this system. The "Orange Book" OS requirements in the FIPS 140-1 validation process only apply if the OS protects validated code from unvalidated code. However, in our architecture, the entry of all code into the device is controlled by Miniboot. We have successfully validated our hardware and Miniboot control software at FIPS 140-1 Level 4 [10]; consequently, in order to validate a device customized with a particular application, the developer of that application only needs to validate his additional software (Layer 3 with Layer 2) on the device. Two of our group's current research projects include investigating "partial" validation of our Layer 2 software (to lower the validation barrier for application developers even further), as well as the issues involved in building a provably secure embedded OS from scratch.

4. Communications

Work done within the card's protected environment is presumably done on behalf of something in the outside world—and the most natural starting point is the (possibly untrusted) host. Thus, an on-card application should have at least one host-based communication partner. We needed to provide a way for these partners to identify and address messages to each other; we also needed to provide an underlying mechanism for fast transport of these communications.

We address these problems with the *COM Manager*, the *SCC*[1] *Manager*, and some special-purpose hardware.

Partners Potential models for card-host interaction can vary greatly in complexity. Who should talk to an on-card application? How many instances of an on-card application can there be? What if an on-card application wants to send an unsolicited message?

To speed our design and implementation process, we start with a very simple model: each on-card entity has a host-side partner that initiates work requests, to which the card-side entity responds. Our Layer 2 software provides these services via a relatively simple API provided by the COM Manager, which works in conjunction with the routing and registration tables maintained by the SCC Manager, as well as the host-side device driver. Within the card, the SCC Manager maintains an *Agent Table* containing entries for each *Agent ID* (externally visible name). The sole way that an on-card application task becomes visible to the host is via an entry in this Agent Table. The SCC Manager creates such entries at the behest of the application task itself, but other supervisor entities also access the Agent Table. (In particular, the COM Manager does a look-up in order to route external work to the appropriate internal agent.)

When an application thread is ready to receive communication from the host, it *signs on* (through the SCC Manager), announcing that it is "open for business" under some specified Agent ID already known by the host-side application. Its host-side partners can then send messages to this agent (although the application and its partners should use additional cryptographic and authentication measures if the communication channel is considered insecure). If appropriate, an application can establish multiple Agent IDs, or use the Agent ID to distinguish between different instances of itself.

At first glance, it might seem that the request-response service model is overly limiting. For example, what about a long-lived fraud detection application that sends out alerts only when it detects some critical situation? In theory, the ability for each internal application to have multiple agent names (for receiving messages from the outside) provides an avenue for application-initiated conversation, while still allowing the simpler device-driver behavior of the request-response model. (We will see how effective this avenue is in practice.)

Fast Data Movement An orthogonal set of issues—especially critical for the design goal of "high-performance cryptography"—is how to move data quickly between the host and the card. One might speculate that device hardware links together an I/O port on each machine, and the machines move data by having one CPU send a byte to its port, and the other CPU pick it up. However, this ties up both CPUs—which, in a multi-tasking environment, prevents either CPU from doing more useful work during the transfer.

To address these problems, our hardware includes special *first-in, first-out (FIFO) queues*, controlled by the COM Manager software. (FIFOs integrated better into our hardware and had smaller impact on the host than other

[1] "Secure Crypto Coprocessor": the environment offered to the on-card application.

approaches, such as dual-addressable memory.) At the request of other software, the COM Manager configures these queues to provide pipelines for bulk communications:

- between the host and card (e.g., Miniboot command and application exchanges);
- between two points internal to the card (e.g., perhaps from RAM through DES and back); and
- between two points external to the card (e.g., bulk DES from host RAM through the card).

Since the host is usually running a multi-tasking operating system, the host device driver should also use non-CPU-intensive DMA hardware to transfer data.

5. Secure Persistent Storage

Applications (and, possibly, supervisor code) need storage that persists over hardware reboots, power cycles, and subsequent invocations of that application. Depending on the data, the calling software may require *integrity* (the stored data will not change due to error or malice), *secrecy* (the stored data has not been revealed to an unauthorized party, including someone who physically attacks the device), and/or *atomicity* (the data changes as an atomic unit, despite interruption or failure; no inconsistent, intermediate states are visible).

Our hardware includes two underlying storage components: *FLASH* and *battery-backed RAM (BBRAM)*. However, using these components is complex.

FLASH provides large amounts of non-volatile, non-zeroizable storage:

- The minimum erasable unit in FLASH is a *sector*. The sizes of the sectors vary from 4KB to 64KB, depending on where they reside in the physical FLASH chip.
- Each sector can only be erased a finite number of times before the FLASH chip fails.
- Bits in FLASH can be cleared to zero by a special writing process, but can only be set to one by erasing the entire sector.
- FLASH can be read like ordinary memory, but writing FLASH requires first writing a special sequence of commands to the device. Erasing FLASH is even more complex and time-consuming.
- The contents of FLASH are available to any attacker who pries open the card.

BBRAM provides small amounts of non-volatile, zeroizable storage:

- Unlike FLASH, BBRAM data can be randomly accessed and changed; however, this access must occur over a several-step I/O process to the BBRAM chip.
- BBRAM is zeroized by tamper-response.
- Bits in BBRAM that store the same value for too long can *imprint* that value, remaining visible despite zeroization.

Some types of data storage require using both devices. For example, since design constraints permit megabytes of FLASH but only a few KB of BBRAM, storing large amounts of *secret* data requires using BBRAM to store a session key that decrypts the ciphertext stored in FLASH.

Solution The *protected program data (PPD)*[2] *Manager* provides a simple API for secure persistent storage, while masking the complexity of the underlying storage components. It treats FLASH sectors as a circular buffer, in order to spread the erasure cycles evenly over the memory. Transparent to the caller, the PPD Manager provides atomicity for FLASH writes and (at the invoker's request) will encrypt stored data using keys safely stored in BBRAM. (This use of DES makes PPD a "manager who uses other managers"—increasing its complexity.) To avoid BBRAM imprinting, we periodically invert the contents of BBRAM, transparently to the application. (We also ensure that this inversion is itself atomic.)

Like the FIFOs, both BBRAM and FLASH have the property that these are singular devices. Consequently, concurrent access from different code modules would be successful only if these modules use semaphores or some other software technique to ensure consistent, serializable access. To simplify development, we instead

[2] We deliberately avoided using the FIPS 140-1 term *security relevant data items (SRDI)* in order to minimize confusion. Depending on one's FIPS validation strategy, not all SRDI may be PPD, and not all PPD may be SRDI.

force all software access to BBRAM or FLASH to go through an API supplied by the PPD manager.

(Miniboot also must have secure, persistent storage. However, as Section 2 notes, Miniboot has its own storage regions, and uses hardware locks to block access by anyone—including the PPD Manager—that executes later.)

6. Public Key Cryptography

Our underlying hardware for public-key cryptography is a modular math engine. Our *PKA Manager* must use this underlying hardware to provide RSA and DSA services to Layer 3 code (and, potentially, to other Layer 2 managers). However, this manager should keep hardware-specific details transparent to the calling code, including contention from these multiple algorithms and services for the same basic engine.

Accommodating the goal of speedy development of correct application support code yields additional challenges. As noted earlier, Miniboot needs similar PKA services. Furthermore, host-side test tools—and card-side Miniboot and Layer 2 software *during development*—need these services *without* having access to a modular math engine.

Solution We addressed these problems with the PKA architecture shown in Figure 4: a core library can provide various high-level and low-level interfaces, selectable at compile-time.

Low-Level Code Our PKA Manager contains several sets of low-level routines. Since the hardware accelerator only does modular math, our PKA code needed a large-integer math library to handle all the operations (i.e. fast adding, multiplying, etc., of large integers in native format) not directly supported by the hardware. Key generation additionally requires software support for generating large prime numbers—the most time consuming operation in the key generation process. (We used the ANSI X9.31 standard specifying strong primes.) Key generation also requires randomness—making the PKA Manager another "manager who calls other managers."

High-Level Algorithms Our PKA Manager supports the RSA and DSA cryptosystems. For RSA, we support both encryption and signatures, and support (via caller-selected options) the ANSI X9.31 standard (which was still in draft form during our implementation). Given the complexities of U.S. export policy (and the fact that, in many export scenarios, the maximum key length depends on the context of use), we decided to enforce no export limits in our Layer 2 API; instead, we leave this responsibility to the Layer 3 application.

Simply choosing "the RSA cryptosystem" still leaves decisions on how to extend data (for signatures, the hash; for encryption, the session key) to the full modulus length. For signatures, we implement two variations on the *ISO 9796* scheme; for encryption, we implement the *Optimal Asymmetric Encryption Padding (OAEP)* scheme.

Although many commercial customers use RSA exclusively, we also support DSA because FIPS 140-1 applications and many government customers are restricted to digital signature systems approved by the US government. (The FIPS 186-1 standard, allowing for ANSI X9.31 *rDSA*, was not approved until after our implementation deadline.)

Because of the time-consuming nature of generating key-pairs, and the fact that only one modular math engine exists in our device, we allow other PKA operations to complete during a key generation operation that may have been requested earlier. We check for such operations at different points during the key generation process (i.e., before starting software/CPU intensive operations).

We are currently exploring several other techniques to increase the throughput of various cryptographic operations.

Transparency A modular math engine has some very specific hardware properties: its maximum modulus size, and whether the duration of its operations leaks information about the operands (making it susceptible to timing attacks [5]). In our current hardware, the engine has a limit of 1024 bits and leaks timing information. In order to make these limits transparent, the supporting software accommodates larger modulus sizes (and reduces them to calls to the 1024-bit engine), and provides support for *blinding* as a defense against timing attacks.

However, as we port this software to prototype hardware with an advanced engine that is subject to neither of these limits, some new issues arise. Many current applications use a de-facto limit of 1024 bits anyway, due to the increased performance hit associated with moving beyond

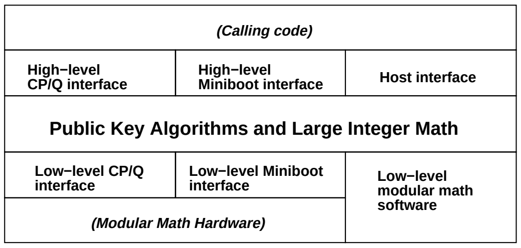
(Calling code)
High–level CP/Q interface
High–level Miniboot interface
Host interface
Public Key Algorithms and Large Integer Math
Low–level CP/Q interface
Low–level Miniboot interface
Low–level modular math software
(Modular Math Hardware)

Figure 4 The structure of our public-key cryptography software.

the engine limit—and also sometimes due to hardcoded limits in legacy host-side code. Furthermore, due to the complexity and performance hit of software blinding, the calling software needs to know whether it is necessary or not. (Some users of application software have already expressed concern that the application does not use blinding anymore on the new prototype.) In hindsight, what really needs to be transparent is "protection against timing attacks" rather than the details of a particular technique which the calling software must explicitly invoke.

7. DES

Our underlying hardware for DES is a high-speed, proprietary DES engine. However, although a design goal was to provide fast bulk DES, approaching the cryptographic performance theoretically possible from the underlying engine requires addressing a number of issues.

For bulk DES, the primary design issue was data transport. We began with the special FIFO hardware discussed in Section 4; the DES Manager can access the engine a byte-at-a-time via *programmed I/O (PIO)* or via DMA through the FIFOs. The existence of two methods is transparent for DES operations where both the source and destination are inside the card—the manager optimizes performance by using either PIO or DMA, depending on a size threshold established at build time. (This threshold accommodates the trade-off between using CPU cycles for PIO, versus using them for setup and interrupt handling during DMA.)

However, when the source and/or destination resides on the host, then the choice of method cannot be transparent to the calling code, since someone on the host-side needs to prepare to ship or receive the data. In these situations, the calling software must indicate the transport method to be used. To optimize performance, the caller should also account for trade-offs—for example, it might be quicker to route DES operations on small data packets into the card for internal PIO.

When the FIFOs are routed through the DES engine, the data will be fed to and extracted from the DES engine as fast as the chip can handle it. However, we found empirically that maximizing bulk-DES speed is also highly dependent on host-side issues beyond the control of coprocessor hardware and software. These issues include page-alignment of source or destination data within the host memory, as well as other host software competing for host bus resources. Sometimes, apparently identical host systems would yield significantly different DES speeds, due to such issues.

API Issues A number of subtleties emerge in providing an API to a bulk-DES engine. Since bulk data may not necessarily be a complete number of DES blocks, our API provides *pre-padding* and *post-padding* options (so the calling software can specify "use this bulk data, but adjust it with these specific bytes"). The need to break a DES operation across multiple calls to the DES Manager (perhaps because not all the data was available for DMA in one shot) also requires that the API include *termination vectors*: the eight bytes at the end of a stream operation that should be used as the initial vector of a subsequent stream operation, for these operations to be composed as a single operation. We provide software for CDMF weakening of 56-bit keys to 40-bit, although (as with RSA) our Layer 2 API enforces no export limits on what it does on behalf of Layer 3. We are currently porting our software to prototype hardware that uses an advanced DES engine with native 3DES support—which, among other things, underscores a need for user education regarding the modes of 3DES: the existence of over 200 chaining variations [2] and the trade-offs between inner-CBC, outer-CBC, and the chaining modes permitted in the 3DES standard.

8. Random Number Generation

Fast availability of random numbers is critical to the performance of many algorithms, in cryptography and security, as well as other areas. Our device hardware includes a thermal noise source that generates a serial stream of random bits, collected in a 16-bit shift register. However, our *RNG Manager* must bridge the gap between this time-sensitive hardware and the various Layer 2 and Layer 3 software modules that need random numbers.

Solution The RNG Manager has two primary tasks: gathering bits from the hardware, and providing them to calling software. To optimize performance, we divided the RNG software into two threads. One thread runs at a high priority (because of the critical nature of providing random numbers) and performs three simple tasks:

- handling the interrupts that signal a full collector register;
- gathering these 16-bit values into a set of eight fresh random bytes; and
- passing these eight bytes to the upper component.

The other thread runs at lower priority (to avoid needlessly preempting other software tasks) and handles incoming requests for random numbers (from other managers and applications), and any optional processing specified by these requests (e.g. specific parity, check for weak DES keys, etc.).

PRNG Our software design did not initially address *pseudo-random number generation (PRNG)*, since we assumed that hardware-generated random numbers would be universally regarded as a better source, and since our hardware RNG passed the full suite of statistical and continuous tests for FIPS 140-1. However, these assumptions proved to be incorrect. Some standards for key generation within particular cryptosystems specify particular PRNG algorithms; the FIPS 140-1 standard for secure hardware requires an approved PRNG between hardware randomness and *any* key material. Furthermore, for performance, some application programmers prefer the faster stream from a PRNG to the relatively slower hardware RNG. To accommodate this, we added a PRNG and various calling and filtering options to the software suite.

Entropy In theory, one might expect that a PRNG should be re-seeded from the hardware RNG as often as possible, to maximize entropy. In practice, this is not true. Many scenarios—such as testing algorithms, and OEAP padding—also require using a PRNG with reproducible results. Accommodating these scenarios required having our code explicitly make the PRNG a deterministic function transforming a *context* to a pair consisting of a random number and a new context.

9. The Application Layer

Our main goal in building this device was to make it easy to develop and run the secure coprocessing applications suggested by previous research. Miniboot provides a means to get the application image into the device itself. But actually connecting this image into the system consists of two main tasks: how Layer 2 initially invokes the application, and how the application then accesses the Layer 2 services.

Invoking the Application Design goals required that the Layer 3 application be changeable independent of Layer 2. For example, application developers may wish to reload successive test versions of their code; application deployers may wish to occasionally upgrade their code; and an overarching goal was maximizing independence of these developers from us, the platform vendor. [9].

As a consequence, our Layer 2 needed to have a mechanism to, at run-time, *load* a Layer 3 component that had not been present when the Layer 2 image was originally built.

Within Layer 2, the SCC Manager accomplishes this by regarding Layer 3 as a simple file system containing the executable module of the user application. During system start-up, the SCC Manager retrieves the user application from this file system and loads it as a program. (A loaded application can then dynamically create additional threads of execution from code already loaded.)

This basic mechanism also provides for much more flexible schemes, which we are only beginning to explore in prototype. For example, we could support classified or proprietary applications—that must never reside in FLASH in plaintext—by dividing Layer 3 into a ciphertext application and a plaintext unwrapper. The unwrapper is loaded at boot-time; later, it provides a key to Layer 2 and requests that the remainder be decrypted and loaded. More complex multi-application and dynamic loading scenarios are also possible—including those that transcend our current "single sandbox" model—but these will require addressing security and garbage collection problems (using solutions that build on the separation features that currently are just programming conveniences).

Using the Services The choice of the CP/Q kernel (Section 3) provides a rich programming environment for applications: developers can write code in C with standard libraries (including `printf()`, when debugging); compile and link with standard tools; and (when the card is configured in development with the debug kernel) have full access to a source-level debugger.

The application can access our additional manager services via function calls (although the library hides the underlying mechanism, which may be a system call or message passing). For potentially lengthy hardware operations, we provide both synchronous (blocking) and asynchronous versions of these calls.

Areas for Improvement Ongoing application development work by both commercial and academic partners reveals several areas for potential improvement. For example:

- The requirement that generic off-the-shelf devices be used for development leads to a two-minute delay each time the code is reloaded through Miniboot, and also permits developmental card-side and host-side applications to thoroughly confuse the host-side device driver.
- The requirement to give developers an option to replace on-card code without clearing on-card state forces application programmers to examine and respond to more initial states than might be expected.
- The design decision to make the card self-contained—*not* storing code or data on the host—has some advantages, including an architecture that will easily port to our portable PCMCIA prototypes. However, it also has some disadvantages; for example, application developers who wish to exploit *cryptopaging* [16] are forced to work with a custom-modified Layer 2.

10. Integration Issues

To some extent, real commercial deadlines for this project drove our strategy: start with an existing embedded kernel and debugger, then have independent programmers concurrently build the different managers, the security bootstrap, the initial application layer, and the host-side support software.

Successes To large extent, this strategy worked. In particular, the modularity of the CP/Q environment simplified testing. For each unfinished code module, a test version existed that provided stubs for the other modules that interacted with it. Each team could then independently exercise their module under the CP/Q kernel, using these stubs to form a skeletal system. Because of the existence of a supervisor-level kernel debugger, these skeletal systems did not even require host-to-card communications. As noted earlier, we also reused source code between Layer 1 and Layer 2—which reached its most elegant form in the common PKA, DES, and RNG source libraries. (The modularity also permitted concurrent, independent development of CCA application code.)

Challenges The modularity also led to some challenges. Independent managers that are invoked and executed as separate computational entities led to some unexpected interactions (e.g., deadlocks at boot-time), as well as to the expected problems (e.g., misunderstandings of the interface, common usage of devices) and decisions (e.g., how to tune the relative priorities of the managers).

Drawbacks On the other hand, this strategy also led to some negative things. Sometimes, the modularity divided design and programming tasks that should have been a unit: for example, in hindsight, a strong case exists for generic public-key and private-key operations and data structures, rather than different styles for RSA and DSA (and, eventually, elliptic curve). Sometimes the modularity also led to an illusion of locality: for example, since statistical testing of the hardware RNG takes approximately 30 seconds on our first-generation device, we attempted to streamline Miniboot operations with no key generation by having the RNG Manager test itself when it is first called—but, to our surprise, this did not improve performance, due to the unexpected intertwining of DSA and RSA code with the RNG.

11. Conclusions and Future Directions

Numerous areas exist for evaluation and tuning of this application support architecture.

For example, did we provide the right set of operation primitives? It turns out that some applications need to use BBRAM for frequently updated data, where speed and long life is more important than transparency and atomicity. As a result, we are extending the PPD API to include such an “updatable” item. For another example, some application developers might appreciate a modular math API, perhaps to complement RSA and DSA with their own elliptic curve implementation. Quantitatively analyzing sample applications, to see how performance could be improved by combining primitives or offering new services as primitives, could also prove fruitful.

Many potential areas exist for tuning the architecture to better achieve the goal of high-performance cryptography—since raw speed of individual pieces of hardware or software do not always result in high throughput. Between a host-side call for cryptographic services and its card-side fulfillment lie many routing and buffer choices, that can be balanced among host/card CPU loads and speeds, as well as other security and performance concerns. The relative priorities and scheduling of the on-card managers and application code is another area for examination. As noted earlier, we are currently exploring other several techniques to increase cryptographic throughput.

As a side-effect of quantifying and explaining the cryptographic performance of our device, we continually encounter a fundamental unsolved problem: *benchmarking* cryptographic performance for meaningful comparison. Does one measure performance from the host or from the internal CPU—or does one merely extrapolate a theoretical speed from the advertised spec for the raw engine? For RSA, does one consider full random exponents, or exponents carefully chosen to optimize performance? (Does one even consider the overhead of safely handling the private key, or of software blinding?) For DES, what size plaintext does one consider? For 3DES, which mode?

However, our overall goal was to build not just a cryptographic accelerator, but a general-purpose secure coprocessor. The true test of our support architecture will come as future experimental (and commercial) applications begin to exploit the potential of putting computation, not just cryptography, inside a trusted, tamper-protected environment.

Acknowledgments

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

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.

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] E. Biham. *Cryptanalysis of Triple Modes of Operation*. Technical Report CS-885, Technion. August 1996.

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

[4] *IBM 4758 PCI Cryptographic Coprocessor*. Product Brochure G325-1118. August 1997.

[5] P. Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. Manuscript, Cryptography Research, Inc. 1995.

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

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

[8] 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.

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

[10] S.W. Smith, R. Perez, S. Weingart, V. Austel. “Validating a High-Performance, Programmable Secure Coprocessor.” *Proceedings, 22nd National Information Systems Security Conference*. October 1999.

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

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

[13] J. D. Tygar, B.S. Yee. *Cryptography: It’s Not Just for Electronic Mail Anymore*. Computer Science Technical Report CMU-CS-93-107, Carnegie Mellon University.

[14] J.D. Tygar, B.S. Yee, N. Heintze. “Designing Cryptographic Postage Indicia.” *Proceedings of ASIAN ’96*. Singapore, December 1996.

[15] S.H. Weingart. *Physical Security Attacks and Defenses*. Draft Research Report, IBM T.J. Watson Research Center. 1999.

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

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

â€œApplication Support Architecture for a High-Performance, Programmable Secure Coprocessor.â€ 22nd National Information Systems Security Conference. October 1999.Â The traditional notion of a cryptographic module is a black box that performs cryptographic operations. With our IBM 4758 product family, we have been extending this concept to high-performance secure coprocessors: devices that can perform high-performance cryptography as well as other sensitive computation beyond the reach of adversaries who may have physical access to the device. We built our device as a generic secure coprocessor platform, in order to enable external developers (as well as IBM) to build and deploy secure coprocessor applications.