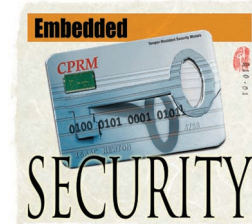# Building the IBM 4758 Secure Coprocessor

Meeting the challenge of building a user-configurable secure coprocessor provided several lessons in hardware and software development and continues to spur further research.

Joan G. Dyer

Mark Lindemann

Ronald Perez

Reiner Sailer

Leendert van Doorn
IBM T.J. Watson Research Center

Sean W. Smith
Dartmouth College

Steve Weingart
Cryptographic Appliances

Work leading toward the IBM 4758 started, arguably, in the 1980s when the Abyss project began exploring techniques to build tamper-responsive hardware and use that technology to protect against software piracy.[1,2] Influenced by this earlier work, we sought to build a secure coprocessor, defined as a tamper-responding device derived from the Abyss, Citadel, and 4755 work. We wanted to provide a single multipurpose platform that third parties could use to develop and deploy secure coprocessor applications, with minimal IBM participation. To accomplish this, we sought to achieve several goals:

- Ensure that the device can be identified externally as to contents, using some form of outbound authentication and public-key interface (PKI).
- Design the device and its software to be securely configurable and updatable in the field.
- Construct the software architecture to accommodate layers of code from different parties, who may or may not trust each other.
- Avoid letting the compromise of one device breach the security of another.
- Validate all these assertions through an external party.

IBM's Common Cryptographic Architecture product group realized that its next-generation product required properties possessed by the secure coprocessor that IBM Research advocated. This knowledge gave the research team a unique and perhaps nonrepeatable opportunity: funding and authority to design and produce the product we thought should exist, as long as it could be transformed into a CCA follow-on and meet the appropriate deadlines.

Seeking to provide an environment where applications could run securely forced us to focus not only on security mechanisms and their implementation and management, but also on various flavors of security policies they must support. Clearly, the hardware on which applications run must be secure, as must the operating system and runtime environment in between, while offering a reasonable API for applications developers. To fix problems in the field and enable fast and inexpensive reaction to changing customer needs, we implemented part of the code as firmware, rather than read-only memory. Figure 1 shows the 4758's three major components and their interrelationships.

Subdividing the software into different layers raises issues of trust because upper components rely on the security that lower layers offer. Applications cannot be more secure than the kernel functions they call, and the operating system cannot be more secure than the hardware that executes its commands.

Thus, if the lower layers are robust, higher layers can choose whether to relinquish some security. We designed the lower layers to be relatively permanent firmware, or static hardware, thereby offering fewer points of attack throughout operation. Decreasing complexity, careful evaluation, and more effective security mechanisms justify the increasing trust along the vertical arrow. Management of the different layers must reflect the demand for increasing trust in lower layers, and be flexible enough to support multiple security policies in various organizations.

As the "Design Decisions" sidebar notes, we

October 2001   **57**
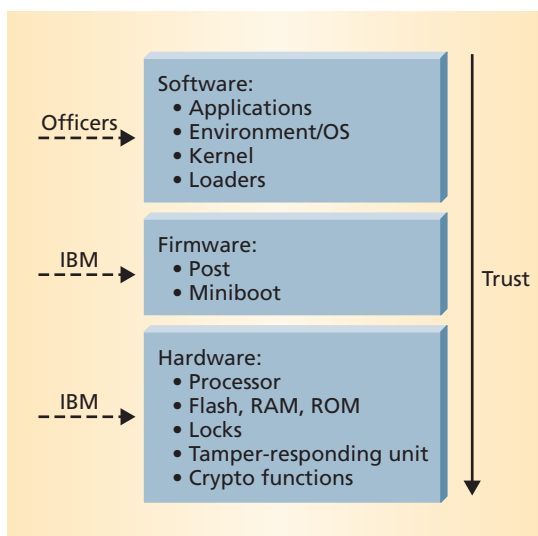
achieved our major goals, but learned through hindsight several techniques for improving the secure-coprocessor design process.

SECURITY AND TRUST

We based the 4758's security on an architecture and implementation validated from manufacture to deployment.

Manufacturing considerations

Because we sought to provide a third-party programmable device, the manufacturing process could not incorporate anything related to the eventual software. Programming, loading of code, or both are completed outside the factory at the customer's discretion. Consequently, when a 4758 leaves the factory, it must be fully armed against tampering.

Initialization and the trusted root

Each 4758 undergoes the same initialization procedure, eliminating the need for individual personalization—which is expensive, error prone, generally requires a large quantity, and doesn't fit the general-purpose secure-coprocessor model. An authentication protocol is performed in the factory to establish an initial secret using a valid source of randomness and minimal software within the 4758's ROM. When shipped, each 4758 can authenticate requests to download code and respond to tampering. Having a meaningful outbound authentication requires a securely generated root certificate, which is established in the factory environment.

Layers and codeload boundaries

Layer refers to the division of software into independent pieces. The initial layer is inserted into ROM

Design Decisions

We found the following to be the crucial ingredients needed to build a programmable, secure coprocessor.

Hardware tamper response

Relying upon software to react to tampering is too slow. However, this observation implies a well-sealed package and careful design of the communication paths between the 4758 and the outside world. Dissipating heat becomes a problem, which affects the speed at which components internal to the 4758 can be run.

The ratchet hardware provides the mechanism that assures prior firmware or software layers that subsequent layers cannot alter their secrets. Hardware enforces this assurance best, rendering areas of storage invisible, protecting areas from tamper, or both. The number of ratchet lock settings, and what they protect, requires careful iterative design—something that deadlines and other constraints do not always allow for. Ideally, the number of locks and associated regions would be configured by firmware.

Randomness

Generating initial secrets requires a validated source of randomness, which ensures that possible destruction of a few cards cannot compromise any others. We encountered two problems with this requirement, neither of them fatal. First, we could find no standard for hardware random-number generators (RNGs).

Hence, for the Federal Information Processing Standards (FIPS) validation procedure, we could only use the hardware to seed a pseudorandom-number generator (PRNG), per the FIPS 186-1 standard. Second, the mandatory statistical tests to ensure that the RNG works correctly—performed on both the hardware source and the PRNG output—slow the boot process by more than 20 seconds.

Layered design

A programmable device must have a layer responsible for implementing code load security policy. The layer should be validated, as its correctness forms the basis upon which subsequent layers depend for their own security.

Subsequent layers personalize the device. We initially selected two layers: one for an operating system and device drivers, the second for application code. This division made it possible for third parties to either write applications to an operating system we supplied, or write everything themselves.

Finally, minimal software in ROM provides for repairs or extensions of initial facilities using the secure field update architecture.

Self-initialization

A programmable device must leave the factory ready to

during the 4758's manufacturing phase, and the remaining layers are downloaded into flash memory using a public-key authentication process associated with a layer's owner. A *codeload boundary* denotes a defined point at which the system accepts a signed file authenticating a load-this-layer command.

Officers and signing keys

At each codeload boundary, an officer and key pair authenticate loading. The previous layer's officer establishes ownership and provides a certificate attesting to the new owner's initial public key. The officer owning the layer has complete control over subsequent loads, including changing the key pair and establishing various trust targeting options.

Retaining state

A secure coprocessor provides the ideal place to keep secrets, provided they disappear upon tampering and that any compromise of a single device does not affect any other coprocessor's security. Secrets are of various kinds: the initial secret key a 4758 generates, the signing keys each officer uses, keys to encrypt persistent data on behalf of applications, and application data. Thus, if hackers succeed in compromising a few 4758s, they will uncover none of the secrets stored within other 4758s.

Secure boot

A 4758 provides a secure boot of itself. The hardware and firmware guarantee that the coprocessor stores only authorized code in flash memory. Authorization is accomplished at codeload time. Checks of flash for integrity are made at each boot before passing control to the next layer. The checks run against hardware-enforced flash contents that are locked read-only by the time the operating system and application run.

The validity of these statements follows, in part, from the secure packaging. An attempt to replace the ROM or flash contents destroys the 4758's secrets. The damaged coprocessor might still run the same application software, but it could not authenticate itself when booted, and the application software's battery-backed RAM and encrypted flash information would be unavailable.

HARDWARE

Figure 2 shows the 4758's hardware architecture and building blocks. Figure 3 shows the ratchet locking mechanism that protects access to the flash segments, with the most liberal policy enforced by the guardian processor.

Tamper response

A 4758 responds to tamper attempts in hardware respond to tamper. Initialization of each card cannot involve personalization, in that third-party software ultimately determines the 4758's personality. Our procedure uses private memory and locks and requires an on-card validated source of randomness. Self-initialization is crucial to providing a self-identifying device that third parties can program.

Outbound authentication

This feature ensures that external parties can determine the exact applications running and the relevant history of all entities that may affect any secrets stored within the 4758. There must be a trusted root and a description of the history that can be validated against that root. The root must be established at the point of manufacture for programmable cards and relies on the hardware we've described. In our implementation, the code loading "firmware" reflects the layering into firmware, operating system, and device drivers, then applications. It relies on the protection of persistent storage areas by ratchet locks, although more standard certificate formats would have aided interoperability.

General coprocessor and auxiliary processors

A general CPU does not limit the application set, and suitable auxiliary processors expand the device's potential use to include data encryption standard, modular math, and so on.

This approach offers many trade-offs. As an example, we provided DES encryption and decryption where the data being transformed could remain on the host. This approach implies a level of trust in the host, and proved suitable for certain applications in which the main problem involved protecting data in transit. Such long-lived "sessions" complicate the device drivers that deal with host-to-4758 transfers and also complicate abort processing. The resulting interface had a performance impact on short transfers as well.

Persistent storage

This feature provides long-term storage of application or other data on the coprocessor, protected directly by the 4758 tamper response via battery-backed RAM or indirectly via encryption using keys stored in battery-backed RAM. Providing this capability lets applications like protected counters function wholly within the 4758, without host trust.

Third-party programming interface

The interface to third parties should be as comfortable as possible. We chose the most suitable operating system available at the time, given our other constraints. A more familiar API that included the host-to-4758 protocol, as well as more attention to ease of development, would be desirable.

In retrospect, we did not always follow the maxim that simpler is better.

Figure 2. The 4758's hardware architecture. Built around a 486 processor core, the coprocessor contains two types of battery-backed RAM, a random-number generator, and persistent storage space for encrypted data in flash memory.
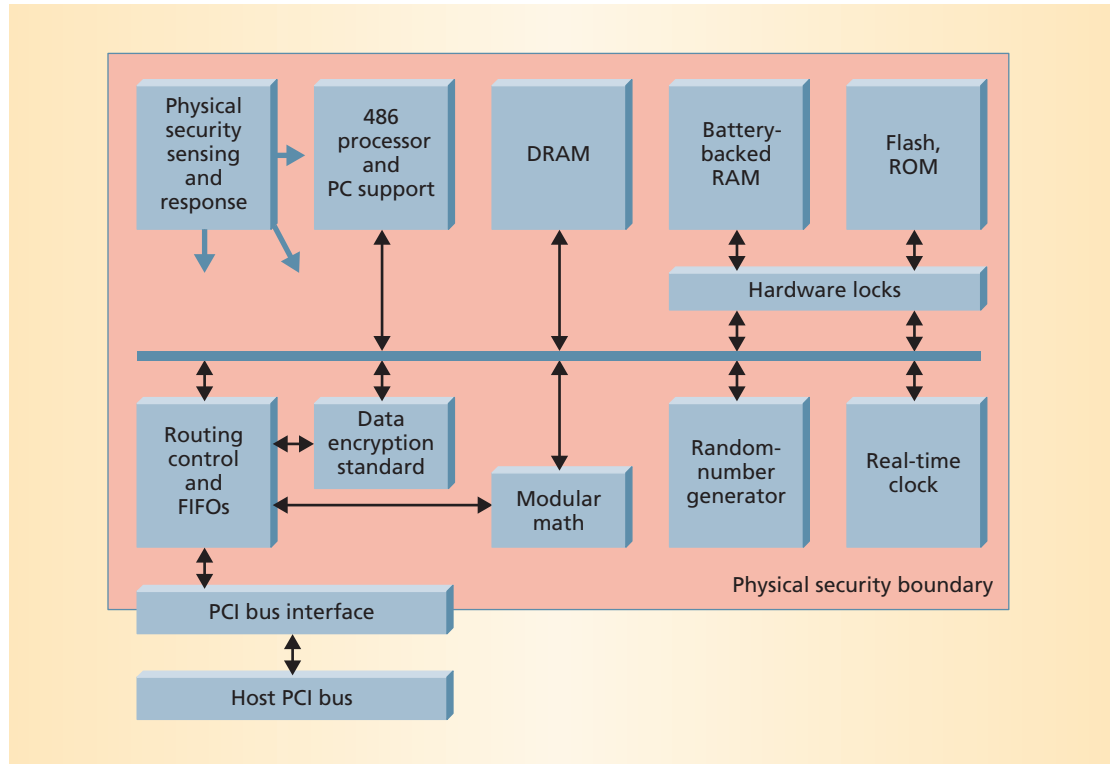
Figure 3. Hardware-based ratchet locking mechanism. The hardware lock protects access to the 4758's flash memory and battery-backed RAM segments.

by quickly zeroing its secrets and executing a co-processor state change, without requiring software intervention. Tamper conditions include penetration attempts, temperature extremes, voltage variation, and radiation.

We prefer this approach because of its speed—using a software solution would be unreliable and too slow. Responding to a tamper attempt requires a secure package, with minimal and carefully engineered access to the outer world. One consequence of the 4758's packaging is that it makes heat dissipation difficult. An effective solution to this problem requires careful interleaving of physical techniques.[3]

Battery-backed RAM

A 4758 has two types of battery-backed storage. The coprocessor's OS manages BBRAM, which protects software secrets. Lockable battery-backed ram (LBBRAM) protects firmware secrets and is only accessed through a separate guardian processor that also maintains the lock state. A tamper attempt zeros all the 4758's battery-backed storage.

Locks

Hardware locks render areas of flash read-only memory and make areas of LBBRAM completely inaccessible. A separate guardian processor controls the

locks, which are implemented as one-way-only ratchets that can be unlocked only by resetting the 4758.

This design provides the required hardware platform for building the trust hierarchy. It ensures that only authorized firmware or software can change the code to be executed and that if any level contains corrupted code, we can fix this code and verify the fix.

The number of locks and the memory regions they protect are defined in hardware. The software layers and security requirements related to codeload signature must be satisfied within the locked regions, even though we didn't know the actual sizes that would be required. Indeed, we froze the hardware design before we completely designed the security software architecture. Choosing the optimal design, in this area particularly, is an iterative process that doesn't always fit within time constraints.

Our trust design ensures that, after boot, the device goes through a sequence of phases that strictly decrease in trust. Although this design limits the damage that runtime corruption can cause, it does not permit runtime recovery from runtime corruption. However, allowing dynamic increases in trust would require hardware assistance that wasn't feasible within the allotted product development time frame.

Randomness

Each 4758 has a protected random-number generator (RNG), implemented in hardware, that generates its own initial secret using relatively straightforward code in ROM. For applications, validated randomness generates a nonce, key, or any initial secret.

Persistent storage

To preserve data across 4758 power cycles and resets, we provided persistent storage to hold downloaded code, signatures used by the firmware, space for the two software layers, and application data. We used flash memory for this purpose. No tamper-responding hardware can guarantee it has sufficient power to erase flash contents, so anything stored in flash is vulnerable to inspection, albeit by destroying the 4758. Therefore, keeping any information requires using an encryption key stored elsewhere, in BBRAM. The boundary between the portion of flash used for signed codeloads and the portion used for data storage is movable to allow for larger code when required.

This approach proved to be a good way to preserve large amounts of data. In combination with the ratchet locks, it supports secure boot of the 4758.

Guaranteeing atomic update in flash memory is difficult. Flash chips provide only sector-level erase, and we also had concerns about each chip's finite lifetime. The movable boundary between signed loads and software data required a handshake between firmware and software. This led to some dependencies between

miniboot and the contents of Layers 2 and 3, which can reflect different owners.

Other elements

Data movement within the 4758 takes place over an internal bus that the host cannot access. Sensitive applications in the 4758, which can make no assumptions about the host's reliability, require internal and external bus separation. However, this separation slows data movement.

A general-purpose computer resides within the secure environment, providing a user-supervisor mode and memory protection. This choice furnishes a general-purpose platform that can run software similar to the software running on unsecured computers. The processor's capabilities do not restrict the potential application programs and operating systems.

The 4578 also includes a modular-math chip, a data encryption standard (DES) and a secure hash engine. The modular-math chip accelerates various on-card algorithms implemented in software, increasing the 4578's potential applications.[4]

The DES engine may be used for bulk operations on external data, without requiring the data to pass through the secure coprocessor. Some applications such as streaming media require this feature because the security issues relate more to handling the package containing the media than to the security of the coprocessor's host. However, implementing long-term transfers introduced other complications—abnormal termination of a host process, speed of short operations, and use of the FIFOs for both DES data and host-4758 communication.

The hardware aspects of the device limit performance for operations such as large numbers of DES operations on short data[5] or DES with alternative chaining schemes.[6]

FIRMWARE

To emphasize the logical distinction between them, we use the terms *firmware* and *software* in a slightly nontraditional way: IBM owns the firmware, whereas other parties may own the software.

Guardian processor

The guardian processor runs code that is installed in ROM during the manufacturing process. This code maintains its *in-the-factory* or *initialized* configuration state, and the hardware locks' state. It also protects the secrets each layer stores, as instructed by the embedded main processor, consistent with the locks' state. Inspection can validate the guardian processor, which is a relatively small code module.

This processor forms part of the security architecture implementation, which relies on the protection of

secrets between the software and firmware layers as trust decreases. Development, testing, and maintenance are more difficult than for the main processor, which highlights the trade-off between a specialized processor for a specific job and a general-purpose processor.

Boot code

The 4758's boot code consists of two logical parts—Layer 0 in ROM and Layer 1 in flash. Because both the card and the processor are complex, we relied on a power-on self-test to provide some degree of confidence that the hardware works as designed. Both functional and security purposes required burning the POST into ROM, but POST software could pose problems. Therefore, we split the POST into two parts, placing the most basic tests in POST 0 in ROM and the remaining tests in POST 1 in flash. This keeps POST 0 simple, small, and, hopefully, relatively bug-free, while POST 1 contains the remaining tests and is reloadable, if necessary.

We also incorporated this idea into the security firmware for the same reasons. We called the code *Miniboot* and split it into two parts: a small basic part in ROM and the rest in flash.

POST 0 checks the hardware that Miniboot 0 will use, and POST 1 checks the rest. After a power-on or reset, POST 0 always runs first, then Miniboot 0, then POST 1, then Miniboot 1. The coprocessor performs authenticity and validity checking before control passes to Layer 1.

Miniboot 0 can generate a random symmetric key to perform encryption and decryption, request an increment of the hardware lock setting, generate a unique secret and store it, advance the 4758 state to initialized, and accept and recognize a properly signed flash load. We placed more complexity, including PKI and outbound authentication, in the updatable flash memory.

Establishing officers

Each layer has a security officer and a key pair which that officer uses for signing commands to the firmware. IBM is the officer for the firmware, and this association is established at the manufacturing site. An officer can assign ownership of the next layer by signing an *establish owner* command and must also certify the public key of the next owner. Beyond this, an officer doesn't need to know anything about the next layer's contents.

This process implements a trust hierarchy between independent and possibly mutually suspicious participants. It also implies that the more-trusted officer must take responsibility for ensuring that officer identifiers associated with different entities or different purposes do not collide.

Establishing keys

To establish a signing key, the officer of the previous layer in the trust hierarchy signs a certificate validating the initial public key associated with a new officer. The 4758 records this new key the first time the new officer presents a codeload command.

Relinquishing ownership. A 4758's characteristics cannot be altered without the acquiescence of the owning parties. Once a layer has an owner, only that owner can surrender ownership. An owner can save and offload sensitive data or delete such data without placing undue trust in the previous owner's intentions or security practices. Miniboot deletes an owner's BBRAM data before the next application loads.

Downloading contents. If a layer's owner has signed the command and, if necessary, has also provided a certificate validating the public key, miniboot accepts a specific layer's content. No portion of the firmware and none of the lower officers need know anything about the codeload's content. The ratchet mechanism protects the information essential to verifying contents and keys, which is stored in the protected part of flash.

Targeting

Officers can target their commands to cards with specific configuration properties. This allows a full spectrum of control, from individual cards to all, with a single one-round protocol.

We found that targeting individual cards was beneficial in at least two situations. In the first, contracted software development of crypto outside the United States had to satisfy export restrictions: The code could only be loaded on the cards under the contractee's control. In the second situation, we needed some diagnostic information for debugging some 4758s that had been in use. This required downloading a debug Layer 2 that had an officer 2 identifier associated with the nondebug Layer 2. We built the appropriate command and targeted it to the one card whose owners had requested it. The command would fail when played against any other card.

Trust

Officers can establish the conditions under which their secrets will be destroyed. This property allows officers to close a backdoor to sneaky behavior by more privileged officers. Multiple developers asked us, "Why do you have options, when clearly only X is necessary?"—but they all indicated a different value of X.

These requirements were the primary reason for choosing to have a single control point for loading and reloading subsequent segments. One central policy enforcer simplified the issues regarding which code a particular layer needs to trust to destroy its secrets under all scenarios.

Configuration and outbound authentication

The trust hierarchy depends on knowing a 4758's software history. If an erroneous or malicious software layer was installed but then overwritten, it must be impossible to detect the potential compromise of crucial data or secrets. For the 4758 Model 1, each change in Layer 1 in the IBM firmware, in which the tamper-responding package validates and protects the ROM, results in adding a new certificate to the chain-keeping history. With Model 2, we implemented "outbound authentication," which also serves Layers 2 and 3.[3]

Each significant change in software is recorded in a certificate chain and made available to applications, not just host programs that talk to Miniboot. The certificate chain is truncated at Layer 1 when all Layer 2 and 3 software and associated secrets have been deleted. A corrupted entity cannot hide its presence or impersonate a more trusted entity because changes in configuration are coupled to key pair changes. Hence, an application can attest to its environment and provide that information to any partner. The partner can then make an informed trust decision.

The current implementation follows the design decision that Layer 1 is aware of the characteristics of Layers 2 and 3, which can have separate owners. Alternative approaches to Layers 2 and 3 require revisiting this code. Also, the certificates are in a nonstandard format; we made design decisions before standards emerged and we had serious concerns about codespace.

SOFTWARE

Two layers give a 4758 its runtime personality. Assuming all is well, Miniboot 1 passes control to Layer 2. Depending on various options and the current configuration, the system may have cleared the BBRAM secrets associated with Layer 3, but it does not require Layer 3 to be runnable. Layer 2 determines the actions that will be taken with respect to Layer 3.

CP/Q operating environment

The 4758 needs a runtime environment that is configurable, efficient, secure, has a small footprint, and provides a reasonable programming API. Configurability was important because some of the 4758's hardware requires special device drivers. The 4758 needed to have a small footprint because it has no paging, which limits the actual memory to what is physically present within the coprocessor.

Given our time constraints, CP/Q—a commercial, off-the-shelf, modular operating system—offered the best available operating environment. This system also supports separation of supervisor and user tasks—a basic reliability feature. CP/Q is not a multiapplication environment with multiple security levels. The CP/Q designers did not address the problem of enforced separation between possibly hostile running applications, either sequentially or in parallel.

When we were designing the 4758, we did not find a validated, secure operating system suitable for an embedded environment that had been designed from the start for security rather than as a general-purpose OS. In hindsight, an OS with a wide community of developers and established software, such as Linux, would have been a better choice, provided that it offered a reasonable runtime footprint and good performance.

CP/Q++

We expanded CP/Q to include device drivers for the 4758's hardware. We call the resulting operating environment CP/Q++. CP/Q++ is well suited to the modular coding style the 4758's developers used to develop device drivers and applications while working at multiple sites. Moreover, we could leverage plenty of local expertise because some of the CP/Q++ developers had written CP/Q. The address-space separation made it somewhat easier to find and fix integration problems. CP/Q++ also simplified debugging, which probably shortened the development period.

Multiple address spaces imply multiple context switches, memory aliasing, and passing requests by message, all of which adds overhead. A timing exercise, performed fairly early in the initial development, showed that cross-memory operations did not contribute significantly to overhead.

We needed to choose which pieces to build into Layer 2, which were always loaded and started, and what privileges and resources to allow each module. We used CP/Q's builder and system-level debugger internally to experiment with various configurations. During the development phase in particular, we used the debugger to ensure that only required privileges were assigned and to confirm dependencies among modules. Adding test programs using the system-level debug environment simplified testing and debugging of interactions between modules. During the integration phase, we didn't always recognize that we needed to view the configuration file as part of the code base.

Each thread in CP/Q has a priority that we assigned to optimize system throughput, a fairly easy task given CP/Q's configurability. However, we could only estimate the actual mix of work because this task is highly application-dependent.

Host-4758 protocol

Communication between the 4758 and the host in which it resides involves two main activities: establishing a connection between the communicating part-

> **When designing the 4758, we did not find a validated, secure operating system suitable for an embedded environment that had been designed from the start for security rather than as a general-purpose OS.**

ners and transferring information data, requests, and results. We selected what seemed to be a minimal design, always initiated from the host, that could exchange several inbound and outbound buffers. Assuming that the host OS would have threading capabilities, we used a blocking protocol on the host side. The communicating partners identify themselves to one another via a 16-byte agent ID. We used mailboxes and interrupts to implement the protocol with a host device driver, a communications manager in the 4758, and some underlying hardware.

The protocol delivers very good performance for large, bulk-DES operations and is general enough to allow experimentation with other variants, such as using long-lived buffers for data transfer. However, it is not a familiar paradigm. Setup overhead has a tremendous impact on performance for small operations. Long-lived operations, shared between the 4758 and the host device drivers, imply complex abort processing.

Different owners

We subdivided the software into two sublayers—Layers 2 and 3—with independent officers, each with its own independent keys. Various customers have used the ability to program a 4758, typically building on CP/Q++ or other IBM software. Such customization makes it possible to use the 4758 for applications that have specialized requirements and to test the results on real hardware.

The programming environment has complexities of its own. The variety of security and trust options offered to third parties, and to Layer 2, appear to be more than customers needed.

Further, Miniboot 1 makes some guarantees regarding the protection of secrets associated with Layer 3 when reloading Layer 2, Layer 3, or Miniboot 1. These guarantees may be too closely associated with CP/Q++. For example, Miniboot 1 increments the ratchet so that Layer 2 is unable to write to Layer 3's flash area, even though the guardian processor's policy, shown in Figure 3, is more permissive. This restriction avoids a vulnerability associated with having all protected flash segments on the same chip: A write to any one sector can erase all sectors. However, it precludes using Layer 3 as a read/write file system extension of Layer 2.

Outbound authentication issues

CP/Q++ offers an application programming interface that requires the cooperation of Miniboot 1. To provide services to Layers 2 and 3, the API extends the certificate chain that traces all updates to the firmware. It also provides a key pair and configuration history, which allow the entity's potential partners to make an informed trust decision. The private key remains in the 4758.

Extending or replacing CP/Q++

The separation of authority between Layers 1 and 2 allows placing alternative or third-party software in Layer 2. Reassigning Officer 2 to allow experimentation does not compromise the security of deployed cards.

As shipped by IBM, Layer 2 contains all the supervisor-level code and will load, at most, one user-level application—which is insufficient for some experiments. A research extension of CP/Q++ allows loading additional supervisor and user-level code. Instead of acquiring and learning the CP/Q configuration and build tools, the developer can use the symbolic debugging documented as part of the toolkit.[7,8] The OS extensions form part of the read-only disk image downloaded into Layer 3, so installation is the same as for application code, and the signed download command acts as the authenticator. This approach bypasses some of the outbound authentication problems associated with modular, extensible OSs.

A research adaptation of Linux for the 4758 is under development. However, the assumptions Miniboot makes with respect to the division into two owners—Layers 2 and 3—are tested, with a few surprises.

Application development and the toolkit

To ease the learning curve for 4758 application developers, we provided a familiar coding environment.[7,8] We considered several factors, including the compilation and link stages, the platform on which the new application was to run, and the CP/Q++ runtime environment. We constructed an extended CP/Q++ and configured it to contain a debug probe and various means for the probe to use either the PCI bus or the serial port to converse with the host.

We supported the C programming language and compilers supplied by both IBM and Microsoft. Unavoidable additional steps included the translation from the host-based executable to the runtime format CP/Q++ requires, and subsequent packaging and downloading of the codeload to the 4758.

The new API includes the CP/Q operating system and the API provided by the device drivers used to access the 4758's unique hardware. If the internal API proved inadequate, we provided no means to enhance CP/Q++.

Given the relatively lengthy download time, we could have reduced the development time by further extending CP/Q to deal with loading programs that reside on the host, but we didn't give that effort a high priority. In a production 4758, we cannot short-circuit the secure download process, and we cannot bypass the FIPS testing done in Miniboot and POST.

Applications

We designed the 4758 so that third parties could write their own applications without IBM's involvement, approval, or even knowledge. We believe the 4758 was the first commercially available, secure coprocessor to provide this facility. Several applications that require hardware protection, such as banking, use the initial application, IBM's CCA. The 4758 has been used to provide CCA extensions, metering applications, an implementation of PKCS-11, and support for various research projects.

Each application developer must recreate a secure protocol for communicating between a host application and its corresponding 4758 application. A larger collection of possible plug-in modules or code samples, a more mainstream coding environment, and a better-publicized API might have encouraged wider use.

In developing the 4758, we met our major research security goals and provided the following features:

- a lifetime-secure tamper-responding device, rather than one that is secure only between resets that deployment-specific security officers perform;
- a secure booting process in which each layer progressively validates the next less-trusted layer, with hardware restricting access to its secrets before passing control to that layer;
- an actual manufacturable product—a nontrivial accomplishment considering that we designed the device so that it does not have a personality until configured in the field;
- the first FIPS 140-1 Level 4 validation, arguably the only general-purpose computational platform validated at this level so far; and
- a multipurpose programmable device based on a 99-MHz 486 CPU internal environment, with a real operating system, a C language development environment, and relatively high-speed crypto.

We are continuing to build on the success of the 4758 project. Currently, we are pursuing work on secure coprocessors in several areas. For example, we're exploring the use of Linux as the operating system, with security enhancements such as SELinux,[9] a socket-based card-to-host protocol, and a host-based file system that allows booting the operating system and applications from the host for faster development. We are also exploring other embedded processors, such as the Power PC 405GP,[10] the addition of a network communication channel, and other form factors, particularly those appropriate for laptops. ✳

References

1. S. Weingart, "Physical Security for the microABYSS System," *Proc. IEEE Symp. Security and Privacy*, IEEE Press, Piscataway, N.J., 1987, pp. 52-58.
2. S. White and L. Comerford, "ABYSS: A Trusted Architecture for Software Protection," *Proc. IEEE Symp. Security and Privacy Conf.*, 1987, pp. 38-51.
3. S. Smith and S. Weingart, *Building a High-Performance, Programmable Secure Coprocessor,* vol. 31, Elsevier Science, New York, 1999.
4. J. Dyer et al., "Application Support Architecture for a High-Performance, Programmable Secure Coprocessor," *Proc. 22nd Nat'l Information Systems Security Conf.*, National Computer Security Center, Ft. George G. Meade, Md., 1999.
5. M. Lindemann and S. Smith, *Improving DES Hardware Throughput for Short Operations,* tech. report RC-21798, IBM T.J. Watson Research Center, Hawthorne, N.Y., 2000.
6. C.S. Jutla, "Encryption Modes with Almost Free Message Integrity," to be published in *Eurocrypt*, 2001.
7. Toolkit Code Distribution, http://www.alphaworks.ibm.com/tech/4758toolkit (current Sept. 2001).
8. Toolkit Documentation, http://www.ibm.com/security/cryptocards (current Sept. 2001).
9. Security Enhanced Linux, http://www.nsa.gov/selinux (current Sept. 2001).
10. IBM PowerPC 405GP/CR, http://www.chips.ibm.com/products/powerpc (current Sept. 2001).

Joan G. Dyer is a research staff member at the IBM T.J. Watson Research Center, Hawthorne, New York. Her work focuses on operating systems, device drivers, linkers, loaders, and other software development tools. Dyer received a PhD in mathematics from New York University. Contact her at joandy@watson.ibm.com.

Mark Lindemann is an advisory programmer at the IBM T.J. Watson Research Center. His work centers

> **We designed the 4758 so that third parties could write their own applications without IBM's involvement, approval, or even knowledge.**

on operating systems, including system software and hardware architecture. He is EPA certified. Contact him at markl@watson.ibm.com.

Ronald Perez is a senior software engineer in the Network Security and Cryptography department at the IBM T.J. Watson Research Center. His research interests include secure systems, operating systems, and complex embedded systems. Perez received a BA in computer science from University of Texas, Austin. He is a member of the IEEE and the ACM. Contact him at ronpz@watson.ibm.com.

Reiner Sailer is a research staff member at the IBM T.J. Watson Research Center. His research interests include network security, general-purpose secure runtime environments, and security architectures for distributed applications. Sailer received a PhD in engineering from the University of Stuttgart, Germany. Contact him at sailer@watson.ibm.com.

Leendert van Doorn is a research staff member at the IBM T.J. Watson Research Center. His research interests include security and operating systems. Van Doorn received a PhD in computer science from Vrije University in Amsterdam. He is a member of the IEEE, the ACM, and Usenix. Contact him at leendert@watson.ibm.com.

Sean W. Smith, an assistant professor in the Department of Computer Science at Dartmouth College, is currently on leave from the IBM T.J. Watson Research Center. His research focuses on the practical and theoretical aspects of security and privacy in distributed systems. Smith received a PhD in computer science from Carnegie Mellon University. He is a member of the ACM and Usenix. Contact him at sws@cs.dartmouth.edu.

Steve Weingart is a senior hardware engineer at Cryptographic Appliances, Sacramento, California. His work focuses on cryptographic coprocessors, with emphasis on physical-security circuitry and packaging. Weingart received a BSEE from the University of Miami. Contact him at steve@cryptoapps.com.