RC21416(96675)19FEB1999

Computer Science/Mathematics

Research Report

Validating a High-Performance, Programmable Secure Coprocessor

S. W. Smith, R. Perez, S. Weingart, V. Austel
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

Validating a High-Performance, Programmable Secure Coprocessor

S.W. Smith, R. Perez, S. Weingart, V. Austel
Secure Systems and Smart Cards
IBM T.J. Watson Research Center

February 15, 1999

Abstract

This paper details our experiences with successfully validating a trusted device at FIPS 140-1 Level 4—earning the world's first certificate at this highest level. Over the last several years, our group designed and built a physically secure PCI card containing a general-purpose processor with crypto support. However, for this device to function as a *trusted* platform for secure coprocessor applications, we needed to *establish* that assurance through independent validation. We chose FIPS 140-1, since discussions of secure hardware usually cite that standard, and Level 4, since the weaker levels did not provide sufficient assurance for many proposed applications.

Successful validation at Level 4 required withstanding a fairly open-ended suite of physical attacks, and preparing formal modeling and verification of the internal software—as well as meeting a number of other sizable challenges that were not initially apparent. In some sense, our validation effort was an experiment to quantify the design and work effort necessary to achieve this previously unachieved security assurance level. Since our device is a *programmable* platform, we hope this work substantially lowers the barrier for others to develop, deploy, and validate secure coprocessor applications.

1. Introduction

Secure coprocessors enable secure applications in hostile environments, by providing trusted sanctuary for the computation and data storage that such applications require. However, for this technology to take root in practice a sufficiently high-performance device must *exist* that provides general-purpose programmability by third parties in a way that is both practical and secure, and *independent evaluation* must establish that the device is indeed trustworthy.

Over the past several years, our group has been working on transforming these research ideas into real security solutions that can be applied in the real world. Other reports [11, 12] summarize the design problems we encountered and the solutions we applied, in order to bring this technology into existence. This paper describes our experiences in assuring its security by earning the first successful FIPS 140-1 Level 4 validation. Section 2 describes the device we built. Section 3 describes the FIPS 140-1 standard, and validation process. Section 4 presents our experiences with this process, and Section 5 presents some observations on potential areas to improve the process.

2. Background: Design and Validation Goals

Many existing—and potential—computing applications suffer from a vulnerability: potential adversaries may have direct access to the hardware, software, and data storage used by an application. This data may include private and public cryptographic keys; this software may include the implementation of the cryptographic algorithms that generate and use these keys; and these adversaries may include the end user of the application. The potential of both building high-performance programmable secure coprocessors, as well as using such platforms to build secure applications, have been long-standing areas of research, both at our laboratory (e.g., [1, 2, 9, 13, 14, 20, 21]) and elsewhere (e.g., [8, 16, 17, 18, 19]).

In order for secure coprocessing technology to enable real solutions in the real world, we felt that a flexible, generic platform must exist as a mass-produced product, not just a hand-built laboratory prototype. This device must provide physical security against tamper attacks; high-speed cryptographic performance; general-purpose programmability (so that parties other than the manufacturer can develop and securely deploy software for these devices, in quantities as small as one); and a security architecture that knits this all together.

However, bringing such a platform into existence only solves part of the problem. Applications founded on trusted platforms require assurance of that trust. The FIPS 140-1 standard is commonly cited (e.g., [15]) as a metric for the resistance of a device against physical and logical attacks—for applications as basic as cryptographic accelerators or as advanced as postal meters or network auditors. Only Level 4, the highest level of this standard, provides sufficient physical or software security assurance for mid-to-high level applications, especially when potential adversaries may have direct access to the device—but no device had ever achieved this level of validation.

This situation created another challenge for us. By successfully validating our platform against this highest (and previously unachieved) level of security, we would clearly break new ground. In addition, since our device is a programmable platform, its validation would substantially lower the barrier for validation of third-party applications that add custom software to our platform.

3. The FIPS 140-1 Standard

In order to establish well-defined computing metrics for critical applications, the U.S. government established a set of *Federal Information Processing Standards (FIPS)*. Many of these pertain to computer security; U.S. law requires that government entities abide by these standards, and many private-sector and international entities voluntarily follow suit. FIPS 140-1, *Security Requirements for Cryptographic Modules*, is the Federal standard used for tamper-resistant

devices in general, and cryptographic modules in particular. The U.S. *National Institute of Standards and Technology (NIST)* and Canada's *Communications Security Establishment (CSE)* jointly administer FIPS 140-1.

Unlike many other standards, FIPS 140-1 consists of two components: the *standard itself*, a set of rules specifying security of such devices, and a *validation process,* by which independent, government-sanctioned laboratories certify that a given module actually complies with these rules. The FIPS 140-1 standard was drafted over several years by a panel of government and private-sector scientists and engineers (including one of the authors of this paper). The FIPS 140-1 validation program went into effect in 1994; the legal requirement that all modules used in government applications be validated against this standard went into effect in 1997.

Due to its nature as both a set of rules and an evaluation process for adherence to these rules, FIPS 140-1 consists of several components: the original standards document [7]; *Derived Test Requirements* document [3], which expands the standard into a lengthy sequence of explicit validation tests; *online implementation guidance*, which expand and clarify various aspects of the standard and the tests; the accumulation of precedent and interpretation at NIST, CSE, and the evaluation laboratories; and the various standards and draft standards from FIPS, ANSI and others for cryptographic algorithms and protocols, cited by the above components.

Different applications may require a different cost-benefit tradeoffs for security. The FIPS 140-1 process acknowledges this by providing for multiple *levels* of validation, specifying increasing levels of security assurance. Level 1, the weakest level, requires some design assurance, but does not require physical security. Level 2 adds a basic level of physical and software security, and places some limits on the handling of secrets. Level 3 was intended to require substantial physical security, and places substantial limitations on the handling of secrets, and requires substantial software documentation and review. Level 4 approaches impenetrability: the physical security must resist *any* attack the evaluation laboratory attempts, and software documentation must extend to a full formal mathematical model, and formal proof of security within that model.

Any developer can claim "FIPS 140-1 compliance" with no further scrutiny. However, only a module that has successfully undergone the tests and analysis performed by the independent laboratory—and then reviewed by the U.S. and Canadian governments—can claim "FIPS 140-1 *validation.*" The rating a module earns is the lowest of its rating across a suite of validation criteria, which itself can vary depending on what the module is: e.g., software-only, or single-chip, or embedded multi-chip, or embedded multi-chip with on-board hostile code.

Delta The FIPS 140-1 process also allows for *delta* validation for modules that build on a currently validated module. Such scenarios might include when a module's original vendor ports the architecture onto follow-on hardware, or when an OEM vendor adds functionality to an existing module from someone else. In such cases, the vendor can work with the laboratory that performed the original validation and avoid repeating tests and documentation that were part of the original validation.

4. Our Experiences

Lurking in the validation process are several tasks not easily visible in a high-level overview (such as the chart on the NIST web site and distributed by the validation laboratories). In this section, we review some of the principal tasks and tests that we went through as part of our Level 4 validation. Section 4.1 addresses hardware; Section 4.2 addresses software; Section 4.3 addresses algorithm validation; and Section 4.4 addresses some additional documentation and testing requirements. (We stress, however, that only a brief overview is possible in the scope of this report. The full documentation we prepared for the process approached 1000 pages; the complexity and length of just the software formal validation exceeded two Ph.D. theses.)

4.1. Hardware

The most fundamental hardware requirement for a device at Level 4 is that it has to be basically impenetrable: no physical attack can reveal internal secrets. Our defense strategy consisted of two primary techniques: *tamper detection* (ensuring that the device detects all attacks) and *tamper response* (ensuring that the device responds to detected attacks by *zeroizing* any internal secrets before they are exposed). Our security architecture paper [12] contains more details.

An attacker might use many strategies to attempt to penetrate our package. In our design process, we had to anticipate as many of those ways as we could—with the knowledge that, for Level 4, the evaluation laboratory would also test as many potential attacks as they could.

The most difficult issue was quantifying the largest allowable hole that could be made in the package without triggering the tamper detection system. "No undetected penetration at Level 4" naively translates to "*any* hole must be detected." However, for this requirement (and testing) to be feasible, it also has to address *how* such holes can be made. After much discussion, negotiation and work with both the evaluation laboratory and NIST, we reached a specification regarding the state-of-the-art in penetration technology that all parties could then work with.

Once we had the tamper barrier design set, we next approached the zeroization circuitry. It is very difficult to guarantee that, under any tamper scenario, sufficient energy will be available to overwrite the data to be zeroized. Consequently, we used an approach of keeping secret data in static RAM—which we can quickly zeroize by de-powering and then shorting its power connection to ground. Ensuring that this RAM is always kept in a state that will permit its fast, reliable, and complete erasure was a complex task—especially since (from a security perspective) "always" should mean "always": at any point in the life of the device, whether or not it is executing, or even has host power.

It is actually quite difficult to guarantee that the contents of RAM are really gone. There are several actions or conditions that will cause the contents of RAM to *imprint*—so the contents will remain, and may be recovered, even after the power has been removed. Furthermore, for the zeroization circuitry to be useful, something has to make sure that the prevailing conditions will support that zeroization. That task is part of the responsibility of the *Environmental Failure Protection (EFP)* circuitry required at Level 4. Although the validation process does not require EFP circuitry—a device could instead undergo exhaustive *Environmental Failure Testing (EFT)*—this was not feasible in the design of our device, and would probably be impractical for any device containing more than one complex component. (Another report [12] contains more details of our EFP design.) As with direct penetration, the EFP design had to be thoroughly tested and proved to the satisfaction of the evaluation laboratory.

Once the physical security design was completed, the remainder of the electronic design was relatively straightforward; however, the documentation sufficient for validation needed to be more thorough than the documentation for a typical industrial project.

During the design process, specific FIPS 140-1 requirements raised several special considerations. We added a serial port to the processor to allow for the ability to load plaintext keys, since (for Level 3 and Level 4) plaintext keys and other secrets may not be loaded via the common bus. (However, our security architecture does not currently use this feature.) Other requirements—such as needing two independent hardware actions to allow bypass—were design details that had to be explained to circuit designers who would otherwise have saved those extra components.

So far, this section has focused on the *hardware design* issues for Level 4 validation. This focus overlooks a significant and very complex part of our validation effort: developing *manufacturing expertise* to build a secure physical package like this on a production line. Manufacturing feasibility raised another set of tasks that were at least as complex as developing the design.

4.2. Software

Validation requires that the vendor prepare a *security policy*. However, vendors may be surprised that the policy required in the validation process must conform to a specialized rubric, and as a result may differ from the policy documents a vendor might have used throughout development. The classic motivation for constructing a security policy—particularly for modules with sizable software components—is to provide a much simpler blueprint to guide and then verify the implementation. During design and development of our module, we used such a concise policy. But we then found that the detailed access-control policy acceptable for validation grew to a considerable complexity—even though each access control decision this validation-policy expressed followed from our more concise design-policy. (Our concise policy survived as a discussion of security invariants maintained by the device.)

Another potential wrinkle comes from the NIST practice of *publishing* the security policy document for a module that earns validation. For some vendors, this may comprise another instance of the awkward conflict between the research goal of sharing knowledge and the business goal of protecting intellectual property (such as implementation

details). A common solution is, after the vendor prepares the complete security policy and the evaluation laboratory determines that it satisfies the necessary requirements, to partition the policy into "public" and "proprietary" parts.

Finite State Machine FIPS 140-1 validation requires that the vendor prepare a *finite state machine (FSM)* model of all software. In practice, the FSMs for software are expected to be essentially flowcharts, carefully structured to make it easy for evaluators to analyze how the software implements the security policy and preserves security invariants, and *also* to make it easy for evaluators to understand the source code. These two goals were not always consistent: often, we needed to add structure to the FSMs that made sense for one goal, but seemed completely irrelevant for the other.

Additional confusion arose from the terminology used in software FSMs. In many parts of the academic software verification community, "state" is the configuration of the system, which is transformed by execution of code. However, the 140-1 validation process as practiced inverts these terms: "state" corresponds to execution of a portion of software, and another term must be invented for the configuration of the system that is transformed during these "states."

The FSM requirements for FIPS requires annotated diagrams, documentation of the states, state transition tables, and annotation of source code according to the FSM. Coordinating and cross-referencing these items introduces a great deal of complexity—which can be reduced by (as [4] noted) investing some work in the proper tools. (We built various custom tools based on LaTeX.)

Formal Mathematical Model and Verification Perhaps one of the most-feared requirements for Level 4 validation is formal mathematical analysis of module software. This process consists of three basic vendor tasks: building a formal mathematical model of how the system behaves; building a formal specification of security invariants (e.g., in terms of this model, what system behavior or condition is "secure"); then proving that these security invariants remain true under system behavior, as expressed in this model. The fact that no one had successfully completed this requirement before added to the challenge; the process was new not just to us, but also to the evaluation laboratory and to the U.S. and Canadian governments.

Even in describing the requirements, we stumble over the above-noted problem with the FIPS FSM use of the term "state." Many software verification colleagues measure verification complexity in terms of the number of system configurations, which they term "states." But the number of possible system configurations is not a function of the number of FSM-states; a system with 100 FSM-states may only have 100 possible configurations, or 2^{100}, or more. (It all depends on the size of the system state and how the FSM transitions change it). Consequently, effectively communicating the complexity of verifying software modeled with an FSM was a continual problem.

Our Plan Initially, we planned a linear attack. First, we would abstract the FSM to a mathematical model describing the relevant system conditions. Then, we would abstract the security invariants that drove our policy into statements within this model. We then would embed the model in a mechanical theorem prover, and embed the invariants as proposed assertions in the prover. Finally, we would run the prover and obtain the proof of security within the model. (A preliminary report [10] provides details of our initial strategy in developing the mathematical model and expressing the security solutions.)

Strictly speaking, the FIPS process does not require *mechanical* verification. Although (in retrospect) a hand-proof would have been much easier, we were concerned that a hand-proof had too a high a chance of leaving flaws undiscovered. Since it was designed and developed by hand, the system appeared correct to a human eye. But history is full of flawed systems that appear correct; the impartiality and rigor of mechanical verification would provide significantly more assurance.

Formal verification offers two main approaches: *model checkers* search the space of reachable configurations and report any counter-examples to a given assertion; *mechanical theorem provers* produce a formal (and lengthy) proof that a given assertion about a model is true. We opted for theorem-proving because we were fairly confidant that our design and implementation were sound (so we would not need any counter-examples to help debug it), and because the existence of a checkable proof would provide more assurance than a simple statement of "no counter-examples were found." Because we had access to an existing knowledge base for it, we chose the ACL2 prover [5] (after clearing the choice with our evaluation laboratory). A follow-on paper providing a fuller treatment of this model, and its embedding in the mechanical theorem prover, is in preparation.

Target Software and Security Properties As we have discussed elsewhere [11], many subtle and not-so-subtle security issues arise when designing a generic secure coprocessor platform that must satisfy constraints such as: an untampered device must always be able to prove that it's the real thing, doing the right thing; each device has multiple software layers—security configuration/control, operating system, application—which each may come from a different developer; these software developers do not trust each other any more than they have to; each rewritable software layer may be malicious or simply faulty; and the hardware vendor manufactures and ships generic devices, and has no idea how any particular instance of device will be configured and controlled.

We address these constraints with a security architecture [12] that, besides the hardware protections against physical attack, includes security configuration/control and power-on self-test software—as well as this software's positioning within the lifecycle of the device, and its use of special security hardware added to the device. The security configuration/control software evaluates commands sent from various external, remote *officers* (or imposters), which request configuration changes: to device software (including the security configuration/control software), to its officer set, to how its officers are authenticated. This control software must also respond correctly to various non-software events—such as power interruptions, hardware failures, and tamper response–which may occur at any point during execution. When appropriate, the control software must pass control to officers' programs.

The goal of the mechanical verification was to validate that this architecture—hardware, software, and lifecycle—achieved the security goals, over the threats, failures, and other system behavior expressed in the model.

What Happened When we actually carried out our plan, the linear sequence of tasks became quite iterative instead. The abstraction from the FSM to the mathematical model turned out to be very difficult to manage, so we reworked both in order to keep this abstraction much closer to the identity function. Both the mapping of the security goals into invariants within the model as well as producing the mechanical proof proved more challenging than expected, and required far more iterations of FSM and model tuning. For example, establishing that certain invariants held over hardware events such as power or FLASH failure, or tamper response, required that these transitions be explicitly present in the FSM—even though one naively might think of the software FSM showing *software* behavior only. As one consequence, the time we invested in building our documentation tools proved worthwhile, since these tools made it easy to keep everything correlated despite the fluidity introduced by the continual iterations.

Results As we expected, formally specifying *all* device behavior—especially when the "behavior" includes consistency enforcement across failures—proved a rigorous task. Nevertheless, we were somewhat surprised at the way time and temporal sequences showed up when formalizing the security invariants. In retrospect, this makes sense: security *bootstrap* needs to insure not that a state *now* is good, but that any state *now* is taken to a good state by the time anyone important (good or evil) notices.

The verification succeeded—in fact, we discovered many instances where we had unnecessarily redundant tests.

4.3. Algorithms

For the FIPS 140-1 validation process, any "official FIPS" cryptographic algorithms the module uses require additional scrutiny. These algorithms include cryptosystems such as DES, DSS/SHA-1 and (very recently) RSA; and any random number generators: including routines that provide random bits from hardware sources, as well as algorithms (in hardware or software) that amplify seeds into pseudorandom sequences.

This scrutiny falls into three categories. First, the vendor must *document* every use of these algorithms, where the code is, where the keys are, how the keys get generated, and how they leave the boundary. Second, the vendor's implementation of any RNG, as well as each approved cryptographic algorithm, must pass a suite of *validation tests* administered by the evaluation laboratory. Finally, each time the module runs, each algorithm in this category must pass specific tests (distribution tests for RNG; key consistency tests for public key generation; known-answer tests for the others) before being used.

Experience with Algorithm Tests Overall, we lost about a month of calendar time due to unnecessary iterations between us and our evaluation laboratory trying to debug—via long distance—our code, the testing tools, and the

test data. We failed at least one iteration of DSS testing because of typographic errors in the data formatting. The inexpressiveness of the DSS validation tool also proved frustrating—we also failed one iteration because, where the standard was ambiguous on some minor implementation point that was irrelevant from security and usage perspectives, we chose an option different from the option the test tool chose. (However, this latter data—the *reason* the test failed—was unavailable.) We revisit this issue in Section 5 below.

Experience with Randomness The FIPS 140-1 standard specifies that the output of a module's random number generator must satisfy a set of *statistical* tests (showing good distribution), satisfy an on-going *continuous* test (showing that the generator never gets stuck), and must be filtered through an approved PRNG before being used as key material.

Our device harnesses an internal source of thermal noise as a source of random bits that easily satisfy the required tests (and have the additional advantage of being unpredictable, thanks to physics). We then filter this through one of the approved PRNGs before using these bits as key material. Much to our surprise, we found that (during the validation process) the hardware noise was not considered an RNG—so we needed to revise our software to apply the statistical and continuous tests to the PRNG as well.

4.4. Additional Tasks

The FIPS 140-1 standard was written for a traditional cryptographic module: a box that provides cryptographic services. As a consequence, much of the standard and supporting documentation is written using terms and definitions whose meanings are obvious for such a traditional module. However, fitting a module (such as ours) that pushes beyond the traditional boundaries into this rubric is not always an obvious task. For such modules, a task that requires quite a bit of thought—but which does not appear as an explicit work item in the standards documents—is mapping the standards terminology into the module terminology.

Decisions on this mapping must precede many subsequent validation and documentation tasks. For example, the FIPS 140-1 documents establish many rules for the documentation and handling of *security-relevant data items (SRDI)*. One of the above-mentioned mapping tasks is: deciding what actually constitutes an SRDI, in the context of a particular module. This decision then drives subsequent work, such as documenting how each SRDI is accessed and/or modified in the context of which official module services.

The FIPS 140-1 process also defines various rules for how officers and users get authenticated, and what happens in various scenarios, such as "user logs on, then walks away." The mapping needed to be complete before we could document each of these items, and how authentication complied with the FIPS requirements.

Operational Testing An unexpectedly sizable validation task was *operational testing*. Planning and carrying this task requires integrating knowledge of many pieces of the design, including: the source code; the above mapping from system to standard; the error documentation; the authentication documentation; the FSM; and the applicable FIPS rules. In our case, this data was not even stable until late in the validation process, delaying the start of this task.

For each item in the lengthy set of testing requirements from the evaluation laboratory, we then needed to: interpret what the test means for the device; verify that the device really satisfies the test; prepare a written plan for the evaluation laboratory on how we propose to demonstrate that the device satisfies that test; once the proposal is approved, build (and test) automated tools and scripts to perform this demonstration (including demonstrating every error condition); then carry out this demonstration, live.

In theory, much of FIPS operational testing could merge with the testing a vendor carries out anyway during the course of product development. However, in practice, the operational testing depends highly on the above mapping of standard to product—and this mapping may not be clear until much of the product testing is already complete.

5. Beyond FIPS 140-1

This development and validation work gave us a unique exposure to the FIPS 140-1 standard and process. One cannot go through such a process without analyzing the process itself. In this section, we quickly present some suggestions for possibly improving the standard and the validation process in its next revision, FIPS 140-2.

Not Just Crypto Boxes As we have noted, secure coprocessors are finally migrating from research prototypes to commercial products. Recent examples include the new generation of postal meters [14, 15, 21], our device, and various proposed rights-management tokens. We suspect many more will emerge. [9]

Work in this area has long cited the FIPS 140 standard as a specification of tamper resistance—since nothing else exists. However, FIPS 140-1 was clearly aimed at only those tamper-resistant modules whose purpose was to perform some suite of cryptographic services. But in order to apply to these new generations of devices, FIPS 140 needs to be broadened to tamper-resistant modules whose purpose is something else, such as "securely load and execute various programs" or "maintain a monetary balance and modify it only under appropriate conditions."

This aim, coupled with the direction of technology, leads to a mismatch that may only grow worse. A revised standard might avoid this mismatch by generalizing the notion of what a secure module might do. For example, the standard could generalize SRDI to be not just cryptographic keys, but any data item critical to secure and correct operation of the module, and generalize the security properties that a module enforces for its SRDI beyond "keep it secret." (Natural examples already exist for integrity-only SRDIs—and requirements for currently unforeseen SRDI properties will undoubtedly emerge with new modules.) The revised standard might also generalize the notion of "users" and "officers" beyond "someone in the same room as the module." For example, it is very natural to authenticate a module service request using a public-key signature—but the signer could very well be an *organization* (e.g., PKI certificate authorities are usually not *people*) at some distant point in space and time.

Context of Module FIPS 140 intends to specify module security assurance levels. But in many cases, the security of a module depends on the broader context of its use—not just on the module itself.

For example, if a module M authenticates its officers using public-key or symmetric-key cryptography, then these officers have private keys somewhere other than module M. The revised standard might address how officers control and store these private keys—since the security of M depends on these issues.

For another example, assuring that the design of some module is tamper-resistant is meaningful only if means exist to assure some relevant set of users/officers that a particular instance of this module is genuine (meets this design) and non-tampered. How does an officer know it's really an untampered module when he first opens up the box from the factory? If a module is permitted to suspend certain protections when it is returned to a secure factory vault, how does the module itself determine when it is back in the factory? (How does the factory authenticate that this is a real, untampered module?) A revised standard could address these issues by considering broader issues of the *lifecycle* of the device, and its tamper-assurance goal. For example, it could require partitioning physical locations into "trusted" and "untrusted" sets, and then require documentation that: tamper protections are enabled before a module transitions from a trusted to an untrusted place, and remain enabled throughout the module's stay in the untrusted place; methods exist to authenticate that a genuine module is untampered whenever it transitions from an untrusted place to a trusted place; and methods exist for a "relevant party" to authenticate an untampered module as such, even in an untrusted place.

(Although not required by the standard, our architecture and our validation documentation explicitly addressed these lifecycle issues.)

Tamper Resistance for Software FIPS 140 provides various levels of assurance that a module can resist tamper. Consequently, the FIPS 140-1 process specifies various tests and rules regarding how the module *hardware* resists penetration attempts.

However, FIPS 140-1 does not specify a similar set of rules for resisting *software* penetration. Admittedly, software penetration (offense and defense) is an ever-evolving field. But (in addition to preserving the software design and

verification requirements of FIPS 140-1), perhaps FIPS 140-2 might explicitly address some basic principles. For example, validation might require that the vendor demonstrate how maliciously malformed or out-of-spec input cannot compromise module security; how interruptions (such as maliciously timed power failures) cannot compromise module security; and what precautions exist to prevent compromise due to the unforeseen bugs (such as wild pointers) that almost always exist in complex software.

Since this level of implementation detail is largely orthogonal to the security architecture, it would make more sense to require this level of analysis in *addition* to the formal verification. Perhaps the highest level of FIPS 140-2 could also require resisting some degree of free-form software tiger-team attack, as FIPS 140-1 Level 4 does for hardware.

(Our FIPS 140-1 Level 4 module was developed in continual consultation with our own software penetration team; although not required, we included in our validation submission an explicit analysis documenting our module's resistance to this family of attacks.)

Software Engineering Practices History shows that security vulnerabilities in software often result from bugs, and that good software engineering practices can reduce the chance for such bugs. Consequently, we recommend that FIPS 140-2 be broadened to include more explicit assurance about software development and testing practices not explicitly related to security.

For example, FIPS 140-2 might require the vendor to use some subset of standard software engineering practices, such as documented unit testing, along with a good test harness and regression testing; code reviews; structure charts and data flow diagrams; function-point analysis; defect and resolution tracking; and source code configuration management. (We used many of these practices in the development of our module anyway.)

Low-Level Languages FIPS 140-1 requires that software be written in a high-level language, with rare exceptions. However, these exceptions did not explicitly include several areas where engineering dictates that using assembly language is the best approach.

In particular, consider module software that executes on an internal embedded processor. For initial power-on self tests, for transitions between software components before an operating system has been established, and for many components of standard operating systems, code needs to work in CPU modes that high-level languages have difficulty accommodating.

A revised standard might broaden the areas where low-level language is permitted (e.g., to explicitly include the above), but require that such portions of code be adequately pseudo-coded in the accompanying documentation (to reconcile this broadening with the need to make the code easy for the evaluation laboratory to analyze)

Random Number Generation We strongly recommend that FIPS 140-2 reflect the fact that high-quality hardware RNGs exist. If the output of a hardware RNG is still to be filtered first through a PRNG, we recommend that FIPS 140-2 at least encourage the PRNG to be reseeded as often as possible from the hardware RNG (in order to maximize the entropy). (We do this anyway.)

A "cryptographically secure RNG/PRNG" is the building block for many protocols, and users will assume that a RNG/PRNG that meets the FIPS standard is indeed cryptographically secure. A revised standard might take more steps to ensure that this is true. For example, if FIPS 140-2 broadens to include hardware RNGs and/or additional (unspecified) PRNGs, we recommend the addition of a test or requirement that provides assurance regarding the *unpredictability* of these bits. If FIPS 140-2 preserves the current 140-1 interpretation that "the only RNG is one of these three PRNGs," we recommend dispensing with the statistical tests. (In this context, the statistical tests only measure the quality of the PRNG algorithm, which makes no sense if one is constrained to a set of approved algorithms.) Finally, a truly random source of bits will fail some of the FIPS 140-1 tests occasionally. We recommend broadening FIPS 140-2 to explicitly address this fact.

Algorithm Validation We recommend that FIPS 140-2 revisit the test tools to eliminate the unnecessary iterations that we and others have faced. Specifically, we recommend that the test tools (with source code) be available to the

vendors themselves, so that they can check their own work before submitting to the validation lab. (If that is not possible, then perhaps a FIPS 140-2 authority can publish sufficient examples of *all the tests* in algorithm validation.)

Level 4 Penetration As we noted, the FIPS 140-1 Level 4, specification that "any physical penetration must be detected" has proven to be impractical to assure and test. We recommend that the FIPS 140-2 process establish a fixed printed specification for the maximum undetected penetration that is allowed, and any special conditions relevant to that penetration (i.e. conductive/non-conductive drill or probe, etc).

Level 3$\frac{1}{2}$ A vast difference exists between the physical security necessary for a multi-chip module to pass FIPS 140-1 Level 3, and the physical security necessary for FIPS 140-1 Level 4. We are concerned that Level 3 is currently too soft, but Level 4 may be to difficult/costly for applications of low to moderate value.

We recommend that FIPS 140-2 fill the gap between Level 3, where simply potting the unit may fully suffice as physical security, and Level 4, where the module must detect and respond to virtually any penetration. To this end, we propose a "Level 3$\frac{1}{2}$" tamper detection envelope as in FIPS 140-1 Level 4, but with less stringent requirements. For example, if Level 4 ends up with a maximum sized penetration detection requirement of X, then Level 3$\frac{1}{2}$ should have a maximum sized penetration detection requirement of approximately $10X$ to $100X$, and a significant reduction of the stringency of testing. This will permit designers to produce a good, full-featured, design without the extreme manufacturing requirements that FIPS 140-1 Level 4 now entails. (Rather than adding a new level, another alternative might be to increase the physical security requirements of FIPS 140-2 Level 3 to those described here for Level 3$\frac{1}{2}$.)

Power Analysis One often encounters reluctance in discussing *power analysis* due its rumored part of classified TEMPEST. However, since power analysis has become a well-known public topic [6], a revised standard should have a clear specification about EM leakage and how testing will be performed (as with the current EMI requirements), or a statement that this is beyond the scope of FIPS 140. (However, given the success of power attacks, it seems that it should be included somewhere in the rubric.)

Terminology We also recommend that a revised standard clarify two areas of potentially confusing terminology.

- As noted above, the use of "state" in FIPS FSMs needs to be distinguished the use of "state" by the software verification community.
- FIPS 140-1 requires *role-based authentication* at the lower levels and *identity-based authentication* at the upper levels. The relatvely recent security research area of *role-based access control (RBAC)* is similar to the former in name. However, in content, RBAC is sufficiently similar to the latter (basing access control on both the identity and the role of the subject) that colleagues familiar with RBAC would sometimes transpose the FIPS 140-1 terms.

6. Conclusions and Future Work

For a long time, our group has believed that secure coprocessors could help solve many real security problems in the real world. Our work in earning the first-ever FIPS 140-1 Level 4 certificate completes the foundation for this vision: a mass-produced programmable coprocessor exists, whose trustworthiness has been independently established.

The next task, for ourselves and for our research partners, is to develop application software that transforms this platform into solutions for these problems, and then to delta-validate this combination—application plus platform. One measure of the success of our effort will be how many follow-on applications come into existence, and earn FIPS 140 validation. It would be gratifying to cite some significant fraction of FIPS 140 certificates as based on this work.

Acknowledgments

Clearly, this paper would not have been possible if we had not had the opportunity to actually build this device, and carry it through the first successful FIPS 140-1 Level 4 validation. Consequently, 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, 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; InfoGard Laboratories for their continual guidance; and Bill Arnold, Liam Comerford, Doug Tygar, Steve White, and Bennet Yee for their inspirational pioneering work.

References

[1] D. Chaum. "Design Concepts for Tamper Responding Systems." *CRYPTO 83.*

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

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

[4] J. Hines. "FIPS 140-1 Validation: the Netscape Experience." *The 1998 RSA Data Security Conference.*

[5] 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.

[6] P. Kocher, J. Jaffe and B. Jun. *Introduction to Differential Power Analysis and Related Attacks.* Manuscript, Cryptography Research, Inc. 1998.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] S.W. Smith, S.H. Weingart. "Building a High-Performance, Programmable Secure Coprocessor." *Computer Networks and ISDN Systems,* Special Issue on Network Security (to appear).

[13] 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.

[14] 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.

[15] J.D. Tygar, B.S. Yee, N. Heintze. "Designing Cryptographic Postage Indicia." *Proceedings of ASIAN '96.* Singapore, December 1996.

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

[17] 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.

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

[19] 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.

[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.