S.W. Smith.
"Hardware Security Modules."
in B. Rosenberg (editor).
Handbook of Financial Cryptography and Security.
Chapman and Hall/CRC. 2010. 257--278.

DRAFT.

Chapter 1

Hardware Security Modules

force label chap:trustcomp (1.1)

force label chap:fu (1.2)

1.1 Introduction

Say the word "bank" to the average person, and he or she will likely think of thick iron safe, housed in a stately marble building, with security cameras and watchful guards. For a variety of reasons — to deter robbers, to insure employees remain honest, to assure customers and the community that the institution is trustworthy, the brick-and-mortar financial industry evolved a culture that valued strong and visible physical security. These values from the brick-and-mortar financial world over to the electronic. Augmenting host computer systems with specialized Hardware Security Modules (HSM) is a common practice in financial cryptography, which probably constitutes the main business driver for their production (although one can trace roots to other application domains such as defense and anti-piracy).

This chapter explores the use of HSMs. Section 1.2 considers the goals the use HSMs is intended to achieve; section 1.3 considers how the design and architecture of HSMs realizes these goals; section 1.4 considers the interaction of HSMs with broader systems; and section 1.5 considers future trends relevant to HSMs. The chapter concludes in section 1.6 with some suggestions for further reading.

1.2 Goals

Building and installing a specialized hardware module can appear a daunting, expensive task, compared to just using a commodity computing system to carry out one's work. Because of this, we begin by considering motivations. Why bother? What is it that we are trying to achieve?

1.2.1 Cryptography

One set of motivations derives directly from cryptography (a term enshrined in the very title of this book).

To start with, many types of cryptography (including the standard public-key techniques of RSA, DSA, and Diffie-Hellman) can be very difficult to do on a standard CPU. This challenge provides nice fodder for a classroom exercise for computer science undergraduates.

- First, one explains the basics of RSA: "$z \leftarrow x^d \bmod N$ — what could be simpler?"

- Then, one challenges students to code this, in C with no special libraries — but for the currently acceptable size of N (say, at least 2048 bits).

Students quickly realize that the "simple" math operations of x^d or $y \bmod N$ becomes not so simple when x, d, y, and N are 2048-bits long and one's basic arithmetic operands can only be 32 or 64 bits long. Naive (e.g., natural, straightforward) approaches to solve these problems are tricky and slow; clever optimizations are even trickier and still not very fast. Supplementing a standard CPU with specialized hardware for specialized tasks is a time-honored tradition (e.g., consider floating-point and graphics support even in commodity personal computers).

Even if a standard CPU can do the cryptographic operations efficiently, it may very well have better things to do with its cycles, such as handle Web requests or network packets or interactive user responses, or calculate risk evaluations or update accounting information. Consequently, another reason for supplementing a computer with specialized cryptographic hardware is just to offload mundane, time-consuming cryptographic tasks.

In the above discussions, we considered the efficiency and opportunity cost of using a standard computer for cryptographic operations. However, we did not consider the case when it might not even be possible to do the operation on a standard computer. Cryptography in the real world requires *randomness* — for things such as symmetric keys, inputs to public key generation algorithms, nonces, etc. Unlike the "random" numbers sufficient for tasks such as statistical simulations, cryptographic randomness needs to be unpredictable even by a dedicated adversary with considerable observation. Producing cryptographically strong randomness on a standard computing platform is a challenge. As computing pioneer John von Neumann quipped:

> "Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."

A standard computer is a deterministic process (albeit complex). One can try to get cryptographic randomness by harvesting inputs (such as disk events or keystrokes) that an adversary might have trouble observing or predicting, but it is messy, and it is hard to provide assurance that it really works. Once again, supplementing the standard machine with specialized hardware might help.

1.2.2 Security

It is probably safe to assert that cryptography is only necessary when we have something to protect. In financial cryptography, what we are protecting has something to do with money, which usually suffices to get anyone's attention! Another set of motivations for HSMs derives from security: using additional specialized hardware may help in protecting critical information assets.

The first step might be to think about what assets need protecting. In the context of cryptography, the first asset that might spring to mind are secret and private keys: we want to make sure that these remain secret from the adversary. However, this is only part of the picture. Besides keys, we might also want to protect other types of data, e.g., customer authenticators); besides data, we might also want to protect algorithms, e.g., a proprietary pricing strategy; besides secrecy, we might also want to ensure integrity, e.g., knowing that the AES implementation is still doing AES; or freshness,e.g., knowing that we are dealing with the most recent set of account records — or some combination of these.

We might then think about the kinds of adversaries we need to protect against — although, since we are dealing with physical devices, one often sees the discussion start first with the avenues these adversaries might use. For HSMs, perhaps due to the historical context of banks, one of the first angles we hear about is physical security: physical armor, and other tricks, to keep an evil adversary from trying to physically attack the device to extract secrets or cripple protections or something similarly nefarious. However, over the last fifteen years, *side-channel attacks* — indirectly inferring secrets via non-intrusive observations of things such as power consumption — have become a more serious class of physical attacks.

However, a less obvious but no less dangerous avenue of attacks is via the software running on the host computer. In fact, one could construct a good argument this is a *more* serious threat — after all, it is hard for an adversary to drill into encapsulated circuit board from over the Internet, but it can be easy for her to access a flaw in a network-facing interface. As anyone running a standard commodity personal or office computer knows, modern computing environments provide a wealth of weak points: applications with flawed interfaces; operating systems with flawed protections and flawed interfaces; users with flawed authentication; hard-to-configure security policies. It might not be as sexy as physical armor, but simply moving sensitive operations *away* to their own dedicated platform may provide a significant security advantage.

The adversaries against whom an HSM defends may constitute a wider, less easily-defined group. As with the avenue of exotic physical attacks, it is tempting to postulate a cloak-and-dagger adversary who sneaks into a facility and carries the HSM off to a well-funded analysis lab — but less exotic adversaries might be the bigger threat. For physical thieves, enterprises might imagine a cloak-and-dagger burglary, but regularly encounter low-level insiders, such as cleaning staff, removing small and easily-fenced electronic equipment. Other adversaries could include insider attackers motivated by greed or anger or blackmail and mounting more sophisticated attacks, by exceeding authorization or exploiting excess authorization on systems and applications. Focusing on malicious insiders can let us overlook insiders who are merely sloppy. Poor computing hygiene — such as rogue software, rogue Web surfing, poorly configured network

firewalls — can provide vectors for an adversary to enter critical systems. As some recent penetration tests have shown, rogue USB flashdrives sprinkled in a parking lot can also be effective [44]; some enterprises even seal USB ports with epoxy, just for that reason.

The above discussion all presented scenarios where an enterprise might use HSMs to protect itself from rogue individuals. HSMs can also be used to protect against rogue enterprises, e.g., an HSM can house not just SSL private keys but the entire server end of an e-commerce Web application, thus protecting customers from rogue commerce sites.

1.3 Realization

Section 1.2 laid out some goals of using HSMs in financial cryptography. We now consider how design and development of HSMs might achieve those goals.

1.3.1 Silicon, for the Software Person

Financial cryptography (and security in general) is typically the realm of computer scientists, not electrical engineers. Consequently, it can be easy for us to just imagine that computation happens via logic circuits in some magical thing called "silicon." However, not all magical silicon is created equally, and some of the variations can be relevant when considering HSMs.

To start with, logic circuits are composed of transistors, which one can think of an electrically controlled electrical switch. The way that logic circuits work is that the transistors switch on and off; the flow of electricity embodies the binary state. However, nothing comes for free: both switching and not switching can require electric power. Furthermore, these devices exist as real physical objects in the real world: consumption of power can mean generation of heat; rapid state changes can mean sudden bursts in power consumption.

It is also tempting to imagine that the process of building transistors and gates into "custom silicon" performing specific functionality corresponds to building lines of code into custom software. However, going from design to actual custom silicon is an expensive and not particularly malleable process. This situation creates a strong incentive to build one's HSM out of off-the-shelf chips, rather than "rolling one's own."

When "rolling one's own" is necessary, it can be far preferable to use a field programmable gate array (FGPA) — an off-the-shelf chip composed of generic circuit blocks that one can configure to be a circuit of one's choosing. One can think of the "configuration" as state held in internal bits of memory, and the varying kinds of memory — ROM "burned" at the factory, or some type of non-volatile RAM which can reloaded in the field, or even volatile RAM, which must reloaded each power-up — lead to various types of FPGAs.

Instead of using an FPGA, one could instead build one's own Application Specific Integrated Circuit (ASIC). With an ASIC, the efficiency and performance is likely to be better than an FPGA. Also, the per-unit cost will likely be lower — but one would need a large quantity of units to amortize the vastly higher initial engineering cost.

Furthermore, with ASIC, one is stuck; one does not have the ease of changes and updates that one has with FPGAs.

1.3.2 Processor

An HSM often is a small computer of its own, installed as a peripheral on a larger host. Although it is easy to rush ahead to think about security and cryptography and such, it is important not to overlook the basic questions of computer architecture for this small embedded system. Economic pressure and concern about heat and power dissipation (especially for HSMs physically encapsulated for security) often to lead to using an older and slower processor than what's currently fashionable on desktops. Things won't be as fast.

When planning and analyzing software for a generic computer, one usually assumes implicitly that the program will have essentially infinite memory — because the address space will be at least 32 bits wide, if not larger, because there will be a big, cheap hard disk available, and because the system will have an operating system that will invisibly and magically take care of virtual memory and paging to disk. In the embedded system inside an HSM, none of these implicit assumptions will necessarily hold. For reasons of size and performance,the OS may be very limited; for economics and heat, the RAM may be limited; for reliability, there might not be a hard disk.

Similarly, for most applications on a generic computer, one can safely assume that the internal buses, interconnecting the CPU with memory and other internal peripherals, are as fast as necessary, and hence invisible; similarly, for many applications, details of the interconnection between the system and the outside world can remain invisible. Again, in an HSM, these assumptions may no longer hold — things inside can be smaller and slower, and hence no longer invisible.

1.3.3 Cryptography

Since cryptographic acceleration is typically the first application imagined for HSMs, let's first consider that.

Public Key Cryptography A natural first step is to identify the computation that is hard on a traditional CPU and design custom hardware for that operation. As observed earlier, for traditional public key cryptography, modular exponentiation, $z \leftarrow x^d \bmod N$, for large integers is hard on traditional CPUs. Consequently, the first thing we might try is simply to build some special silicon (or FPGA) to perform this operation, and then hook it up as some type of I/O device for the HSM's internal processor.

That is the straightforward view. However, thinking through the rest of what is necessary for such an HSM to provide high-performance public-key operations for an external host illuminates less straightforward issues.

To start with, providing a cryptographic operation is more than modular exponentiation alone. Minimally, the CPU needs to determine what operands to use, send these to the exponentiation engine, collect the results from the engine, and store them in the correct place. Consequently, the process of "just use the HSM to continually crank out

simple RSA operations" can make surprising use of internal CPU. In one real-world instance, we found that the exponentiation engine was actually sitting idle half the time, and re-factoring the internal CPU code to use threading doubled the throughput. As another consequence, such typically "invisible" factors such as how the I/O between the internal CPU and the exponentiation engine is structured (is it interrupt-driven, or does it use programmed I/O?) suddenly come into play. Furthermore, performing services for the host requires that the internal CPU somehow receive those requests and return those responses — so I/O across the internal/external boundary is also a factor.

Simple public key operations can also require more than just shuttling around operands and results for modular exponentiation. Basic RSA encryption usually requires expanding the input data to the length of the modulus, via an algorithm carefully designed to avoid subtle cryptographic weaknesses (as the literature painfully demonstrates, e.g. [14]). Signature verification and checking also requires performing cryptographic hashes. Because of operations such as these, an HSM's cryptographic performance can incur an unexpected dependence on the internal CPU — which can be a bottleneck, especially if the HSM designer has matched a too-slow CPU with a fast state-of-the-art exponentiation engine.

Symmetric Cryptography Performing symmetric cryptography can be another application of HSMs. As with public key cryptography, one might consider augmenting the internal CPU with special hardware to do accelerate the operation. Most of the performance and design issues we discussed above for public key can also show up in the symmetric setting. However, hardware acceleration of symmetric cryptography (and hashing too, for the matter) can also give rise to some new issues. Since symmetric cryptography often operates on very large data items (e.g., orders of magnitude larger than an RSA modulus length), the question of how to move that data around becomes important — having a fast engine doesn't help if we can only feed it slowly. Techniques used here include standard architecture tricks such as direct memory access (DMA), as well as special pipelining hardware.

However, thinking about moving data requires thinking about where, ultimately, the data comes from and where it ends up. In the straightforward approach to internal HSM architecture, we would configure the symmetric engine as an I/O device for the internal CPU, perhaps with DMA access to the HSM's RAM. But since the HSM is proving services to the external host, it is probably likely that at least the final source of the data or the final sink, or both, lie outside the HSM. This arrangement gives rise to the bottleneck of HSM to host I/O, as noted earlier. This arrangement can also give rise to a more subtle bottleneck: fast I/O and a fast symmetric engine can still be held up if we first need to bring the data into slow internal RAM. As a consequence, some designs even install fast pipelines, controlled by the internal CPU, between the outside and the symmetric engine.

Of course, optimizing symmetric cryptography hardware acceleration for massive data items overlooks the fact that symmetric cryptography can also be used on data items that are not so massive. The symmetric engine needs to be fed data; but it also needs to be fed per-operation parameters, such as keys, mode of operation, etc, and it may also have per-key overhead of setting up key schedules, etc. Enthusiasm to reduce

the per-data-byte cost can lead one to overlook the performance of these per-operation costs — which can result in an HSM that advertises fast symmetric cryptography, but only achieves that speed when data items are sufficiently massive; on smaller data items, the effective speed can be orders of magnitude smaller, e.g. [30].

Composition Thinking simply about "public key operations" or "symmetric operations" can lead the HSM designer to overlook the fact that in many practical scenarios one might want to compose operations. For example, consider an HSM application in which we are using the host to store private data records whose plaintext we only want visible to a program running safely in the shelter of the HSM. The natural approach is to encrypt the records, so the host cannot see the internals, and to use our fast symmetric engine and data paths to stream them straight into the HSM and thence to internal RAM. However, at best that only gives us *confidentiality* of the record. Minimally, we probably also want *integrity*. If the HSM architecture was designed with the implicit assumption that we'd only perform one cryptographic operation at a time, then we would have to then do a second operation: have the CPU scan through the plaintext in RAM, or send the RAM image back through a hash engine, or set up the symmetric engine for a CBC MAC and send the RAM image back through it. If we are lucky, we have doubled the cost of the operation. If we are unlucky, the situation could be worse — since the initial decryption may have exploited fast pipelines not available to the second operation.

For even worse scenarios, imagine the HSM decrypting a record from the host and sending back to the host — or sending it back after re-encrypting. Without allowing for composition of encryption/decryption and integrity checking acceleration, we move from scenarios with no internal RAM or per-byte CPU operation to scenarios with heavy involvement.

Randomness As we observed back in section 1.2.1, another goal of an HSM is to produce high-quality cryptographic randomness — that is, bits that appear to be the result of fair coin flips, even to an adversary who can observe the results of all the previous flips. Cryptographic approaches exist to deterministically expand a fixed seed into a longer sequence that appears random, unless the adversary knows the seed. Using this purely deterministic approach raises worries. Was the seed unpredictable? Could an adversary have discovered it? Do we need to worry about refreshing the seed after sufficient use? What if the cryptography breaks?

As a consequence, HSMs often include a hardware-based random number generator, which generates random bits from the laws of physics (such as via a noisy diode). Hardware-based approaches raise their own issues. Sometimes the bits need to be reprocessed to correct for statistical bias; some standards (such as FIPS 140 — see section 1.4.4 below) require that hardware-derived bits be reprocessed through a pseudorandom number generator. Since hardware bits are generated at some finite rate, we also need to worry about whether sufficiently many bits are available — requiring either pooling bits in advance and/or blocking callers who require them, or both (practices familiar to users of /dev/random in Linux).

1.3.4 Physical Security

Since "HSM" usually denotes an encapsulated multi-chip module, as opposed to a single chip, we will start by considering that.

Marketing specialists like to brag about "tamper proof" HSMs. In contrast, colleagues who specialize in designing and penetrating physical security protections insist that there is no such thing — instead, the best we can do is weave together techniques for *tamper resistance*, *tamper evidence*, *tamper detection*, and *tamper response* [50, 49, e.g.].

Tamper Resistance We can make the package literally hard to break into. For example, the IBM 4758 [43, 12][1] used an epoxy-like resin that easily snapped drillbits (and caused an evaluation lab to quip that the unit under test ended up looking "like a porcupine").

Tamper Evidence We can design the package so that it easily manifests when tampering has been attempted. Many commercial devices use various types of hard-to-reproduce special seals and labels designed break or reveal a special message when physical tamper is attempted. However, for such approaches to be effective, someone trustworthy needs to actually examine the device — tamper evidence does not help if there is no *audit* mechanism. Furthermore, such seals and labels have a reputation of perhaps not being as effective as their designers might wish.

Tamper Detection We can design the package so that the HSM itself detects when tamper attempts are happening. For example, the IBM 4758 embedded a conductive mesh within the epoxy-like package; internal circuitry monitored the electrical properties of this mesh — properties which physical tamper would hopefully disrupt. Devices can also monitor for temperature extremes, radiation extremes, light, air, etc.

Tamper Response Finally, we can design an HSM so that it actually takes defensive action when tamper occurs. In the commercial world, this usually requires that the designer identify where the sensitive keys and data are, and building in mechanisms to *zeroize* these items before an adversary can reach them. Occasionally, designers of HSMs for the financial world wistfully express envy that they cannot use thermite explosives, reputedly part of the tamper response toolkit for military HSMs.

Of course, a tradeoff exists between availability and security, when it comes to tamper response for HSMs. Effective tamper response essentially renders the HSM useless; depending on the security model the designer chose, this may be temporary, requiring painstaking reset and re-installation, or permanent, because, after all, tamper may have rendered the device fundamentally compromised. False positives — responding to tamper that wasn't really tamper — can thus significantly impact an enterprise's operations, and can also incur significant cost, if the HSM must be replaced.

[1] This chapter uses the IBM 4758 as an illustrative example because of the author's personal experience in its development.

Attack Avenues In security, absolutes are comforting; in systems engineering, promises of expected performance are standard. However, physical security of HSMs falls short of these ideals. With tamper response, the designer can make some positive assertions: "if X still works, then we will take defensive action Y with t milliseconds." However, for the others, it seems the best one can do is discuss classes of attacks the designer worried about. "If the adversary tries $A \in S$, then the device will do Y_a." The truth of these assertions can be verified by evaluating the design and explicitly testing if it works. However, the security implied by these assertions depends on the faith that the adversary will confine himself to the attacks the designer considered. The nightmare of defenders — and the hope of attackers — is that an adversary will dream up a method not in this set.

A related issue is the expense of the attack — and the skill set and tools required of an adversary. Sensible allocation of resource requires the designer consider the effort level of an attack, that is, the expected work required by an attacker to accomplish the attack. The defenses should be consistent, and not cosider an estoric attack of one type if only defending against basic attacks of another type. Sensible allocation of enterprise resources dictates a similarly balanced approach. However, how to evaluate the "difficulty" of an attack is itself difficult. One can demonstrate the attack — but that only gives an upper bound on the cost. Another nightmare of defenders is that an adversary will dream up a vastly cheaper way of carrying a previously expensive attack — and thus compromising attempts at balanced defense.

Single Chips With Moore's Law, we might expect shrinking of form factor. The HSM that had required a large multichip module to implement could instead fit into a single chip, assuming the economic drivers make sense. Physical security for single-chip modules tends to be more of a game of cat-and-mouse: a vendor claims that one cannot possibly compromise the chip, often followed by an attacker doing so. For an enlightening but dated view of this, consult Ross Anderson and Markus Kuhn's award-winning paper at the *2nd USENIX Electronic Commerce* symposium [2].

Chip internals aside, an Achilles' heel of single-chip approaches, often overlooked, is the security of the connection between the chip and the rest of the system. For example, in recent years, multiple groups have shown how the TCG's *Trusted Platform Module* can be defeated by using a single additional wire to fool the it into thinking the entire system has been reset [23]. Even if the HSM is a single chip, which typically is more tightly coupled with a motherboard than a PCI peripheral is with a host, it is still important for the designer to remember the boundary of the module — and to regard what is outside the boundary with suspicion.

1.4 The Bigger System

Of course, an HSM only is useful if it is embedded within the larger system of a host machine, and an enterprise that operates that machine as part of running a business in the real world. In this section, we consider some of the design and implementation issues that arise from looking at this bigger picture.

1.4.1 The Host Computer System

The API Typically, an HSM provides computational services to another computational entity. For example, some financial program P runs on the host machine, but for reasons of security or performance or such, subroutine R runs on the HSM instead. Enabling this to happen easily requires thinking through various mechanics. On a basic level, we need to think of how P on the host tells the HSM that it needs to run R, how to get the arguments across the electrical interface to the HSM, and how to get the results back.

Essentially, the required mechanism looks a lot like a Remote Procedure Call (RPC) [36], the hoary old technique for distributed computing. To enable P to invoke R, we need a function stub at P that *marshals* (also known as *serializes*) the arguments, ships them and the R request to the HSM, collects the result when it comes back, and then returns to the caller. To enable the developer to easily write many such Ps with many such invocations, we would appreciate a mechanism to generate such stubs and marshaling automatically. RPC libraries exist for this; and the more modern tool of *Google protocol buffers* [39] can also help solve the problem.

Since the coupling between an HSM and its host is typically much closer than the coupling between a client and server on opposite sides of the network, the glue necessary for the HSM-host API may differ from general RPC in many key ways. For one thing, it is not clear that we need to establish a cryptographic tunnel between the host and HSM; e.g., the PCI bus in a machine is not quite as risky as the open Internet. Fault tolerance may also not be as important; it is not likely that the HSM will crash or the PCI connection disappear without the host being aware. More subtly, the need for strict marshaling — all parameters and data must be squished flat, no pointers allowed — may no longer be necessary in an HSM that has DMA or busmastering ability on its host. As noted earlier, failure to fully consider the overheads of data and parameter transport can lead to an HSM installation woefully underexploiting the performance of its internal cryptographic hardware.

Cryptographic Applications Typically, the standard application envisioned for an HSM is cryptography; the API is some suite of R for doing various types of crypto operations. Cryptography raises some additional issues. Convenience and economics provide pressure to standardize an API, so that a program P can work with variety of HSMs, even if they are all from the same vendor, and perhaps even with a low-budget software-only version. Alternatively, security might motivate the development of HSMs that can be drop-in replacements for the software cryptography used by standard tools not necessarily conceived with HSMs in mind, e.g., think of an SSL Web server, or an SSH tool.

Another distinguishing feature of cryptography as a subject is that, historically, it has been a sensitive topic, subject to scrutiny and complicated export regulations by the U.S. government, e.g. [29]. As recently as the 1990s, a U.S.-based HSM vendor needed to take into account that customers in certain countries were not allowed to do certain crypto operations or were limited in key lengths — unless they fell into myriad special cases, such as particular industries with special approval. The need to balance streamlined design and production against compliance with arcane export

regulations can lead to additional layers of complexity on a cryptographic API, e.g., imagine additional parameters embodying the permissible cryptographic operations for the installation locality.

Attacks The cryptographic APIs provided by HSMs have, now and then, proven lucrative attack targets. Ross Anderson's group at Cambridge make a big splash demonstrating vulnerabilities in IBM's *Common Cryptographic Architecture* (CCA) API, by sequencing legitimate API calls in devious ways [7]. In some sense, the fundamental flaws in the CCA API resulted from the above-discussed drivers: the goal of trying to provide a unified API over a disparate HSM product line, and the Byzantine API complexity caused by export regulations. The author's own group at Dartmouth subsequently caused a much smaller splash demonstrating ways to bypass HSM security by hijacking the library and linking connections between the host program and the HSM [32]; one might attribute the fundamental flaw here to the need for a uniform API leading to its overly deep embedding and dependence on vulnerable commodity operating systems.

Secure Execution Another family of HSM applications is to have the HSM provide a safe sanctuary for more general types of computation. Some of us who do research in this area envision some fairly sophisticated usages here such as auctions or insurance premium calculations; however, in commercial practice, these applications often arise from an enterprise's need for a slightly customized twist to standard cryptographic operations.

Enabling developers to easily write such applications requires more RPC-like glue. Besides tools for automatically generating stubs and marshaling, the developer would also appreciate the ability to debug the code that is to reside on the HSM. One solution approach is to set up a special debugging environment inside the HSM itself — which raises security concerns, since debuggers can enable security attacks, and the stakeholder may worry whether a particular deployed HSM is configured normally or with a debugger. Another approach is to develop and debug the HSM code by first running it in a special environment on the host itself, an approach aided by RPC, and only later moving into the HSM. This approach raises effectiveness concerns: emulation software is notorious for being not quite like the real thing.

Moving more arbitrary computation into the HSM also raises a host of security concerns. One is control: who exactly is it that has the right to add new functionality to the HSM — the vendor, the enterprise, or third party developers? Other concerns follow from standard worries about the permeability of software protections. If two different entities control software running on the HSM, or even it is just one entity, but one of the applications may be customized, and hence fresher and less tested, can one of them attack or subvert the other? Can an application attack or subvert kernel-level operations inside the HSM — including perhaps whatever software controls what applications get loaded and how secrets get managed? Giving a potential adversary a computational foothold inside the HSM can also increase the exposure of the HSM to side-channel attacks (discussed later), because it becomes easier to do things like probe the internal cache.

Checking the Host In the initial way of looking at things, we increase security by using the HSM as safer shelter for sensitive data and computation. However, many researchers have explored using an HSM to in turn reach out and examine the security state of its host, e.g. [6, 24, 47, 51] For example, an HSM with *busmastering* capabilities might, at regular intervals, examine the contents of host memory to determine if critical kernel data structures show evidence of compromise [38]. The idea is moving out of the research world; in recent years, more than one commercial vendor has produced HSMs of this type.In some sense, one might think of such HSMs as *trusted platform modules* on steroids (see chapter 1.1).

Using the Host Almost by definition, an HSM is smaller than its host computing environment. As a consequence, the HSM may need to use the host to store data or code that does not fit inside. Doing so without compromising the security properties that led us to shelter computation inside the HSM in the first place can be more subtle than first appears. Obviously, we immediately increase the risk of a denial-of-service attack — what if the data the HSM needs is no longer present? And clearly sensitive data should be encrypted before being left on the host. We also should take care to apply integrity protections so that we can detect if the adversary has modified the ciphertext, and perhaps even use a technique such as randomized initialization vectors to ensure that the adversary cannot infer things from seeing the same ciphertext block stored a second time.

However, further thought reveals more challenges not addressable so easily. For one thing, encryption and integrity checking do not guarantee that the HSM will receive the most recent version of a data block it has stored on the host. What stops an adversary from replaying an old version of the data, e.g., an access control list that still lists a rogue employee as legitimate? One countermeasure is to retain a cryptographic hash of the data block inside the HSM — but this can defeat the purpose using the host because the internal storage is too small. Using a Merkle tree could work around the space concerns, but may incur a significant performance hit without special hardware, e.g. [10, 45]. Thinking about freshness also raises the issue: does the HSM have a trustworthy source of "current time?"

Beyond freshness, we also need to worry about what the HSM's access patterns — which block of data or code the HSM is touching now — will tell the adversary. Countermeasures exist to provably obfuscate this information, e.g. Asnonov [3] and Iliev and Smith [19] building on Goldreich and Ostrovsky [15], but they exact a high performance penalty — and probably still only count as "research," not ready for prime time.

Above, we have considered using the host for augment data storage for the HSM. We could also use the host to augment the HSM's computation engine, but, in the general case, this would require advanced — and, for now, probably research-only techniques, e.g., Malkhi et al. [31] and Iliev and Smith [20] building on Yao [52], to ensure that the adversary could neither subvert the computation undetectably nor illegitimately extract information from it.

The Operating Envelope When discussing physical security (section 1.3.4), it is tempting to think of the HSM statically: as a unit by itself, at a single point in time. However, HSM operations usually require nonzero duration, and are affected by the HSM's physical environment — such as temperature, pressure, radiation, external power supply, electrical behavior on I/O signal lines, etc. Indeed, "affected" might be an understatement; correct HSM operation might actually *require* that these environmental parameters fall within certain constraints. Furthermore, this set of operations we worry about may include the tamper protections we depend upon for HSM security; some designers stress the importance of carefully identifying the operating envelope and ensuring that the HSM's defenses treat actions that take the HSM out of the envelope as tamper events.

1.4.2 The Enterprise

The HSM must also embed within the broader enterprise using it.

Management and Authorization To start with, the fact that the enterprise is using an HSM implies that the enterprise almost certainly has something to protect. Keys, data, and computation are sheltered inside the HSM because the host environment is not sufficiently safe; the HSM performs cryptography for the enterprise because some storage or transmission channel is not sufficiently safe. However, for the HSM's sheltered environment to be meaningfully distinct from the generic host,let alone safer, someone needs to think through the HSM's *policy*: what services the HSM provides; the conditions under which it provides those services; the entities who can authorize such services; how the HSM determines whether these entities in fact authorized them. As in the classic undecidability-of-safety results [17], the set of services the policy speaks to can include the ability to modify and extend the policy itself. Unlike the typical host computer, an HSM usually does not have a direct user interface — so it must rely on less-trusted machines to act as intermediaries, raising questions about the security in turn of the machines and user interfaces used in authentication. For example, if we were to use a highly physically secure HSM for sensitive RSA operations but use a generic Internet-connected desktop to perform the cryptography authorizing the commands to control that HSM, it would not be clear how much we have gained. Furthermore, given the mission-critical and enterprise-centered nature of typical financial cryptography, any authentication and authorization scheme probably needs to include concepts — such as role-based access control and key escrow or other method of "emergency override" — foreign to the typical PC user.

Outbound Authentication The above discussion considered how an HSM might authenticate the entity trying to talk to it. In many application scenarios, particularly when the HSM is being used as a secure execution environment, and particularly when the enterprise is using the HSM to protect against rogue parties at the enterprise itself, it is also important to consider the other direction: how a remote relying party can authenticate the entity that is the HSM. The advent of "trusted computing" (chapter 1.1) has given rise to the notion of *attestation* as the ability of a computational platform to tes-

tify to a remote party how it is configured. However, the earlier work on the IBM 4758 HSM developed a deeper notion of outbound authentication: the HSM security architecture binds a private key to an onboard entity, with a certificate chain tying the public key to the "identity" of this entity [41]. The entity can thus participate as a full-fledged citizen in cryptographic protocols and exchanges.

Of course, such approaches raise the tricky question of exactly what *is* an "onboard computational entity." In an HSM that allows only one security domain inside itself, what happens when the software is updated, or erased and replaced, or even trickier, erased and replaced with exactly the same thing? What about the any security control or OS or management layers — are these the same as the "application," or different? An HSM that allows more general domain structures, such as multiple possibly mutually suspicious applications or temporary applications ("load this bytecode into your internal VM right now, temporarily") gives rise to even more challenging scenarios.

Maintenance In addition to management and authorization of ordinary HSM operations, we also need to think about how to manage *maintenance* of the HSM. Given the nature of physical encapsulation, hardware repairs may be beyond question. However, software repairs are another matter. In addition to esoteric issues such as updating functionality or cryptography, we also need to worry about the more straightforward problem of fixing bugs and vulnerabilities. Consider how often a typical PC operating system needs to be patched! The set of who should authorize software updates may extend beyond the enterprise, e.g., should the HSM vendor be involved? Another set of questions is what should happen to sensitive data inside the HSM during maintenance. Some enterprises may find it too annoying if the HSM automatically erases it; others may worry that retention during upgrade may provide an avenue for a malicious party such as a rogue insider at the vendor to steal sensitive secrets.

Maintenance can also interact with physical security. For example, many approaches to physical tamper protection require a continuous power source, such as onboard batteries. How does the enterprise replace these batteries without introducing a window for undetectable tamper?

Also, as we noted earlier, using tamper evidence defenses requires that someone notice the evidence. Such audit processes need also be integrated into enterprise operations.

The Trust Envelope "Classical" presentations of computer security stress the importance of defining the trusted computing base (TCB). Typically, the TCB is defined in terms of the software environment within a traditional computer: the minimal component that one *must* trust, because one has no choice; however, if one assumes the TCB is sacrosanct, then one can have faith that the rest of the system will be secure. When analyzing the security offered by a traditional computer system, a sensible step is to examine its TCB. Does the system's design sensibly pare down its security dependencies, and channel them toward a reasonable foundation? Does the stakeholder have good reason to believe the foundation actually works, and will not be compromised?

When considering HSMs, similar issues apply — except that we may need to broaden our view to account for the potentially distributed nature of both the secu-

rity protections the HSM may provide and the infrastructure that helps it provide them. For example, consider a physically encapsulated HSM that is intended to provide a secure execution environment, assuring remote relying parties that the sheltered computation is protected even from rogue insiders at the enterprise deploying the HSM. As with the TCB of a traditional computer system, we might start by looking at the internal software structure protecting the system right now. However, we need to extend from software to look at the physical security protections. We need to extend from a static view of operation to look at the full operating envelope across the duration of runtime. We may need to extend through the history of software updates to the HSM: for example, if the previous version of the code-loading code in an HSM was evil, can the relying party trust what claims to be running in there now? We also may need to extend across enterprise boundaries: for example, if the HSM vendor can issue online, automatic updates for the code-loading code in an HSM, does the relying party need to trust the continued good behavior of the HSM? We may also need to extend beyond the technical to regulatory and business processes. If security depends on auditing, who does the auditing, and who checks the result? If the relying party's confidence depends on third-party evaluations (section 1.4.4 below), what ensure the evaluator is honest?

Recall that one sometimes hears the TCB defined as "that which can kill you, if you don't watch out."[2]

The End Things come to an end, both HSMs and the vendors that make them. When an enterprise integrates an HSM into its applications, it is important to take into account that the a vendor may discontinue the product line — or the vendor itself may go out of business. In such scenarios, what had been the HSM's advantages — the physical and logical security that safely and securely keep keys and secrets inside it — can become a source of significant consternation. If the enterprise cannot migrate these sensitive items into a new HSM (possibly from a new vendor), how can it guarantee continuity of operations? But if the enterprise *can* migrate data out under these circumstances, what prevents an adversary from using the same migration path as a method of attack?

1.4.3 The Physical World

HSMs exist as physical objects in the physical world. This fact, often overlooked, can lead to significant security consequences.

Side-Channel Analysis It is easy to think of computation, especially cryptography, as something that happens in some ethereal mathematical plane. However, when carrying out computation, computational devices perform physical-world actions which do things like consume power and take time. An adversary can observe these actions; analysis of the security of the computation must not just include the official outputs of the ethereal function, but also these physical outputs, these *side channels*.

In the last fifteen years, researchers in the public domain have demonstrated numerous techniques, e.g. [27, 28], to obtain private and secret keys by measuring such physical observables of a computer while it operates with those secrets. Usually these attacks

[2] The author heard this attributed to Bob Morris, Sr.

require statistical calculation after extensive interaction with the device; however, the author personally saw one case in which monitoring power consumption revealed the secret key after a single operation. Researchers have also demonstrated the ability to learn cryptographic secrets by observing behavior of a CPU's cache [5] — which an adversary who can run code on the CPU concurrently might be able to exploit.

What's publicly known are these public research results. A prevailing belief is that intelligence communities have known of these avenues for a long time (sometimes reputed to be part of the *TEMPEST* program looking at electromagnetic emanations, e.g. [37]). Furthermore, the prudent HSM designer needs to assume that adversaries also know about and will try such techniques.

The published side channel work tends to focus either on specialized, limited-power devices such as smart cards (which we might view as low-end HSMs) or more general-pupose computers, such as SSL Web servers. However, the attacks may apply equally well to an HSM, and the prudent designer should try to defend against them. As with general physical attacks, however, defending against side-channel attacks can end up a game of listing attack avenues (e.g., timing, power) and defending against them — normalizing operation time, or smoothing out power consumption — and can leave the defender worrying about whether the attacker will think of a new type of side channel to exploit.

Fault Attacks With *side-channel analysis*, the adversary examines unforeseen *outputs* of the computation. The adversary can also use such unexpected physical channels as *inputs* to the computation — and "breaking" the computation in controlled ways can often enable the adversary to learn its secrets, e.g. [4, 8].

For a simple example, suppose an HSM is implementing the RSA private key operation by iterating on the bits in the private exponent. If the adversary can somehow cause the HSM to exit the loop after one iteration, then the adversary can easily learn from the output whether that bit was a 1 or a 0. Repeating this for each bit, the adversary now has an efficient scheme to learn the entire private exponent.

One approach to defending against such attacks is to try to enumerate and then close off the possible physical avenues for disruption. If an adversary might introduce carefully timed spikes or dips in the incoming voltage, the HSM might try to smooth that out; if the adversary might bombard the HSM with carefully aimed radiation of some type, the HSM might shield against that. These defenses can fall under the "operating envelope" concept, discussed earlier. Another approach is to have the HSM check operations for correctness; however, trying this approach can lead to some subtleties: the action taken when error is detected should not betray useful information, and the checking mechanism itself might be attacked.

1.4.4 Evaluation

In a very basic sense, the point of an HSM in a financial cryptography application is to improve or encourage *trust* in some broader financial process. Of course, if one talks to the sociologists, one knows that defining the verb "trust" can be tricky: they insist it is an unconscious choice, and debate fine-grain semantics for what it means to "trust X

for Y." In the HSM case, let's annoy the sociologists and consider trust as a rational, economic decision: should a stakeholder gamble their financial well-being on financial computation correctly occurring despite adversaries?

For an HSM to help achieve this correctness goal, many factors must hold: the HSM must carry out is computation correctly and within the correct performance constraints; the HSM must defend against the relevant adversaries; the defenses must be effective against the foreseen types of attacks. Factors beyond the HSM itself must also hold: for application correctness, the HSM must be appropriately integrated into the broader installation; for effective security, the protections provided by the HSM must be appropriately interleaved with the protections and exposure of the rest of the computation.

Stakeholders trying to decide whether to gamble on this correctness might appreciate some assurance that the HSM actually has these factors. Given that sensibly paranoid security consumers do not relay on vendor claims alone, third-party evaluation can play a significant role. Indeed, we have sometimes seen this cascade to n-th party, where $n > 3$ — e.g., USPS standards for electronic postal metering may in turn cite US NIST standards for HSMs.

What's Examined When it comes to considering what a third-party evaluator might examine, a number of obvious things come to mind. How tamper resistant is the case? Can the evaluator penetrate without triggering tamper detection or leaving evidence? Do tamper response defenses fire quickly enough — and actually zeroize data? Can the evaluator break security by deviously manipulating environmental factors such as power, voltage, or temperature? What about monitoring such factors — perhaps combined with time of operations?

Although perhaps not as obvious an evaluation target as physical armor, an HSM's cryptography is also critical to its security. An evaluator might test whether the HSM actually implements its cryptographic algorithms correctly, off-the-record anecdotes claim this fails more often one would think, whether sources of randomness are sufficiently strong, whether key lengths are long enough, whether key management is handled correctly. The evaluator might also examine whether, for various elements such as "block cipher" or "hash function," the HSM designer chose appropriate primitives: e.g., SHA-1 instead of MD5, or AES instead of DES. When it comes to cryptography, one can see another level of indirection in evaluation: e.g., NIST standards on HSMs in turn cite NIST standards on cryptography.

Since physical and cryptographic security only make sense if the HSM actually does something useful, evaluators also look at the overall design: what is the HSM supposed to do, and does it do it? This examination can often take the form of asking for the traditional *security policy* — a matrix of who can do what to whom — for the HSM, and then testing that the HSM allows what the policy allows, and that the HSM denies what the policy denies. The evaluator might also examine how the HSM determines authentication and authorization for the "who": is the design appropriate, and is it correctly implemented.

What's Not Always Examined Although it is easy to think of an HSM as "hardware," it is also important that an evaluator look at "software" and interface aspects, these factors tend not to receive as thorough attention. Can the evaluator give some assurance to the stakeholder that the HSM's internal software is correct? The evaluator could ask for basic documentation. The evaluator could also examine the software design, development and maintenance process for evidence of best practices for software security: things ranging from choice of language (was C really necessary — and if so, were any security analysis tools or libraries used, to reduce risk of standard C vulnerabilities) to standard software engineering practices, such as version control, to more cutting-edge techniques, such as model checking for correctness. For that matter, a paranoid evaluator should also examine the tool chain: the compiler, linker, etc. Thompson's famous Trojan Horse-inserting compiler [46] was not just a thought exercise, but a real tool that almost escaped into the wild. The evaluator should also look at the external-facing interfaces. Are there any standard flaws, such buffer overflow or integer overflow? Are there any evil ways the adversary can string together legitimate interface calls to achieve an illegitimate result? Some researchers , e.g. [18], have begun exploring automated formal methods to try to address this latter problem.

Looking at software development requires the evaluator, when evaluating HSM security, to look beyond the HSM artifact itself. Other aspects of the system beyond the artifact might also come into play. What about the host operating system? If the entities — at the enterprise using the HSM, or at the HSM manufacturer itself — responsible for authorizing critical HSM actions depend on cryptography, what about the correctness and security of *those* devices and key storage? And if not, what authenticators do these entities use — guessable passwords? What about the security of the manufacturing process, and the security of the shipping and retail channels between the manufacturer?

How It is Examined We have mentioned "standards" several times already. One of the standard approaches to provide security assurance for an HSM is to have it evaluated against against one of the relevant standards. The FIPS 140 series, from NIST, is probably the primary standard here, aimed specifically at HSMs. The initial incarnation, FIPS 140-1, and the current, FIPS 140-2 [13], matrixed four levels of security against numerous aspects of the module; increasing security required stricter rules; the newest version, 140-3, still pending, offers more options. The *Common Criteria* [11] offers more flexibility: an HSM is examined against a vendor-chosen *protection profile* at a specific *evaluation level*.

Usually, official validation requires that a vendor hire a third-party lab, blessed by the governmental standards agency, to validate the module against the standard. Typically, this process can be lengthy and expensive — and perhaps even require some product re-engineering; sometimes, vendors even hire third-party consulting firms, sometimes affiliated with evaluation labs, to guide this design in the first place. On the other hand, unofficial validation requires nothing at all: in their marketing literature, a vendor need merely claim "compliance."

Although valuable, the standards process certainly has drawbacks. Because the rules and testing process are, by definition, standardized, they can end up being insuf-

ficiently flexible.

- The standards process can be expensive and cumbersome — smaller vendors may not be able to afford it; bigger vendors may not be able to afford re-validating all product variations; all vendors might have trouble reconciling a slow standards process with Internet-time market forces.
- Should the standard list exactly which tests an evaluation lab can try? What if a particular HSM suggests a new line of attack which could very well be effective, but is not one of the ones in the list?
- As security engineers well know, the threat environment continually and often rapidly evolves. Can official standards keep up?
- Not all HSMs have the same functionality. How does an official standard allow for the various differences in design and behavior, without allowing for so many options as to lose meaning?
- As we have observed, the end security goal of an HSM is usually to increase assurance in some broader process. How can an official standard capture the varied nuances of this integration?
- As we have also observed, the security of an HSM can depend on more than just the artifact of a single instance of the HSM. Does the official standard examine the broader development and maintenance process?

These are areas of ongoing work.

An alternative to official validation is to use a third-party lab to provide customized vulnerability analysis and penetration tests.

1.5 Future Trends

1.5.1 Emerging Threats

The evolving nature of security threats will likely affect HSM design and evaluation.

For example, an area of increasing concern in defense and presumably in financial cryptography is the risk of tampered components, e.g. [1]. An enterprise might trust the vendor of an HSM — but what about the off-the-shelf components that went *into* the HSM? It is long been well-known that some FLASH vendors provide undocumented functionality ("write this sequence of magic numbers to these magic addresses") to provide convenience to the vendor; these conveniences can easily be security backdoors. More recent research [25] has demonstrated how little need be changed in a standard processor to provide all sorts of malicious features. What *else* could be in the chips — and what if it was inserted by a few rogue engineers as a service to organized crime or a nation-state enemy? How can an HSM vendor — or anyone else — test for this?

1.5.2 Evolving Cryptography

Ongoing trends in cryptography may also affect HSMs.

Hashing Hash functions play a central role in cryptographic operations of all types. However, recent years have brought some entertainment here, with the MD5 hash function beginning to break in spectacular ways, e.g. [26], and the standard alternative SHA-1 deemed not strong enough to last much longer. HSMs with hardwired support for only MD5 are likely now obsolete; HSMs with hardwired support for only SHA-1 may become obsolete soon. That different hash functions may have different hash lengths, and the embedded role that hash functions play in larger algorithms such as HMAC further complicate the task of building in *hash agility*.

Elliptic Curve Our discussion in section 1.3.3 of public key cryptography focused on the traditional and relatively stable world of exponentiation-based cryptography. However, elliptic curve cryptography (ECC), e.g. [16], based on different mathematics, has been receiving increasing practical attention in recent years. One reason is that it can get "equivalent" security for shorter keylengths and signatures than traditional cryptography, and consequently is attractive for application scenarios where channel bandwidth or storage capacity is an issue. Another reason is that ECC has proved a wonderland for innovative new tricks, such as *aggregate signatures* [9], which allow n parties to sign a message, but in the space of just one signature. Because of these advantages, one might expect to see increased usage and demand for ECC support in HSMs; but because of its volatility, picking exactly which operation to commit to hardware acceleration is risky.

Modes of Operation When it comes to symmetric cryptography, we typically see *block ciphers* in financial cryptography. Typically the choice of block cipher is fairly straightforward, based on standards and standard practice in one's community: for a while, we had DES, TDES or IDEA; since it was standardized and since it was designed outside the U.S., AES. However, using a block cipher requires more than picking one; it requires choosing a *mode of operation* — how to knit the block operations together. Which modes are desirable for HSM applications can evolve over time — and it can be frustrating if the acceleration hardware does not support the mode required for an application, such as the recently emerging ones, e.g. [22], that give authentication/integrity as well as confidentiality .

Is Hardware Acceleration Even Necessary? It can be easy to think about HSMs in the context of a single moment. Is the host CPU too slow — can we do cryptography faster in special hardware? However, as time progresses, host CPUs tend to get faster, thanks to Moore's Law — but special hardware stays the same. When integrating an HSM into some application scenario, optimization of engineering and resource allocation needs to take this different aging into account.

1.5.3 Emerging Technology

As we have discussed, besides accelerating cryptography, HSMs can also serve as secure coprocessors: protected, sometimes simpler computational domains, possibly with additional attestation properties. However, rather than looking at such auxiliary

computation engines, we might want to start looking at the main computation engine itself. Currently emerging technology may enable us to start obtaining these HSM-provided properties within the main processor itself. A few examples:

- The notion of *virtual machines* has re-emerged from the dustbin of computer science; vendors and developers are re-tuning operating systems and even hardware to support virtualization, e.g. [40, 48]. Although perhaps primarily motivated by economics, virtualization can also provide simpler, protected, and perhaps, via *virtual machine introspection* attested computing environments, e.g. [35] — but on the main engine, rather than an HSM.

- Newer commercial processor architectures, such as Intel's TXT, formerly code-named LaGrande [21], extend the traditional partition of kernel-user mode into quadrants: a kernel-user pair for "ordinary" operation, and another for more secure, protected operation. Perhaps this new pair of quadrants can provide the environment we wanted from an HSM.

- Because of continuing exponential improvements in computing technology (smaller! faster! cheaper!) many features and applications originally envisioned for high-end HSMs may migrate toward personal tokens, such as smart cards.

- As chapter 1.1 discusses, a consortium of industry and research labs are promoting a *trusted computing architecture* augmenting the main CPU with a smaller *trusted platform module* (TPM) which can measure software, hardware, and computational properties of the main engine, and attest and bind secrets to specific configurations. Partnering the main CPU with a TPM can start to provide some of the protections we wanted in an HSM, albeit against adversaries of perhaps less dedication.

- Combining a TPM with some of these other technologies above can bring us even closer to an HSM in the main engine. For example, using a TPM to bind secrets only to a specifically equipped "secure" pair of quadrants in a TXT-style processor, as CMU prototyped [33], almost creates a dual to the traditional secure coprocessor of research.

As this technology further penetrates commercial environments, it will be interesting to see what happens.

Looming improvements in other aspects of computing technology can also affect HSM design and development. For example, the emerging practicality of semiconductor-based "disks" may make it practical to consider installing "hard disks" inside the computing environment of an HSM — changing both their power, but also changing the requirements and complexity of an internal OS.

1.6 Further Reading

For a longer history of the notion of HSMs and secure coprocessors, the reader might consult the author's previous book, *Trusted Computing Platforms: Design and Applications* [42].

For further research results, the *Cryptographic Hardware and Embedded Systems* (CHES) conference focuses on developments in technology core to many HSMs. With the emergence of "trusted computing" (chapter 1.1) as a research area in its own right, some cutting-edge work in HSM design and applications lands there instead.

To avoid both the appearance of bias and the futility of chasing a moving target, this chapter has avoided discussing specific commercial products, unless the author had direct personal experience, and then only as illustrative examples. For a good snapshot of current commercial offerings, the reader might visit the web site for NIST's cryptographic module validation program [34], which will list currently validated devices and thus point to vendors with active development efforts here.

Bibliography

[1] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan Detection using IC Fingerprinting. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 296–310. IEEE Computer Society Press, May 2007.

[2] R. Anderson and M. Kuhn. Tamper Resistance—A Cautionary Note. In *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, pages 1–11, 1996.

[3] D. Asnonov. *Querying Databases Privately: A New Approach to Private Information Retrieval*. Springer-Verlag LNCS 3128, 2004.

[4] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures. In *Cryptographic Hardware and Embedded Systems—CHES 2002*, pages 260–275. Springer-Verlag LNCS 2523, 2003.

[5] D. Bernstein. Cache-timing attacks on AES. cr.yp.to/antiforgery/cachetiming-20050414.pdf, April 2005.

[6] R. Best. Preventing Software Piracy with Crypto-Microprocessors. In *Proceedings of the IEEE Spring Compcon 80*, pages 466–469, 1980.

[7] M. Bond and R. Anderson. API-Level Attacks on Embedded Systems. *IEEE Computer*, 34:64–75, October 2001.

[8] D. Boneh, R. DeMilllo, and R. Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology, Proceedings of EUROCRYPT '97*, pages 37–51. Springer-Verlag LNCS 1233, 1997. A revised version appeared in the *Journal of Cryptology* in 2001.

[9] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. A Survey of Two Signature Aggregation Techniques. *RSA CryptoBytes*, 6:1–10, 2003.

[10] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. Suh. Incremental Multiset Hash Functions and their Application to Memory Integrity Checking. In *Advances in Cryptology—ASIACRYPT*, pages 188–207. Springer-Verlag LNCS 2894, 2003.

[11] Common Criteria for Information Technology Security Evaluation. Version 2.2, Revision 256, CCIMB-2004-01-001, January 2004.

[12] J. Dyer, M. Lindemann, R. Perez, R. Sailer, S. Smith, L.van Doorn, and S. Weingart. Building the IBM 4758 Secure Coprocessor. *IEEE Computer*, 34:57–66, October 2001.

[13] Federal Information Processing Standard 140-2: Security Requirements for Cryptographic Modules. http://csrc.nist.gov/cryptval/140-2.htm, May 2001. FIPS PUB 140-2.

[14] M. Girault and J.-F. Misarsky. Cryptanalysis of countermeasures proposed for repairing ISO 9796-1. In *Proceedings of Eurocrypt 2000*, volume LNCS 1807, pages 81–90. Springer-Verlag, 2000.

[15] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[16] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[17] M. Harrison, W. Ruzzo, and J. Ullmann. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–470, August 1976.

[18] J. Herzog. Applying Protocol Analysis to Security Device Interfaces. *IEEE Security and Privacy*, 4:84–87, 2006.

[19] A. Iliev and S. Smith. Private Information Storage with Logarithmic-space Secure Hardware. In *Information Security Management, Education, and Privacy*, pages 201–216. Kluwer, 2004.

[20] A. Iliev and S. Smith. Faerieplay on Tiny Trusted Third Parties. In *Second Workshop on Advances in Trusted Computing (WATC '06)*, November 2006.

[21] Intel Trusted Execution Technology. http://www.intel.com/technology/security/, 2009.

[22] C. Jutla. Encryption Modes with Almost Free Message Integrity. In *Advances in Cryptology EUROCRYPT 2001*, 2001.

[23] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*, pages 229–237, 2007.

[24] S. Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, 1980.

[25] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and Implementing Malicious Hardware. In *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[26] V. Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute (extended abstract). Technical Report 2006/105, IACR ePrint archive, March 2006.

[27] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology—Crypto 96*. Springer-Verlag LNCS 1109, 1996.

[28] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *Advances in Cryptology—Crypto 99*. Springer-Verlag LNCS 1666, 1999.

[29] S. Levy. *Crypto: How the Code Rebels Beat the Government Saving Privacy in the Digital Age*. Diane Publishing, 2003.

[30] M. Lindemann and S. Smith. Improving DES Coprocessor Throughput for Short Operations. In *Proceedings of the 10th USENIX Security Symposium*, pages 67–81, August 2001.

[31] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: a Secure Two-Party Computation System. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[32] J. Marchesini, S. Smith, and M. Zhao. Keyjacking: the Surprising Insecurity of Client-side SSL. *Computers and Security*, 4(2):109–123, March 2005. http://www.cs.dartmouth.edu/~sws/papers/kj04.pdf.

[33] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal TCB Code Execution (Extended Abstract). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[34] Module Validation Lists. http://csrc.nist.gov/groups/STM/cmvp/validation.html, 2000.

[35] K. Nance, B. Hay, and M. Bishop. Virtual Machine Introspection: Observation or Interference? *IEEE Security and Privacy*, 6:32–37, 2008.

[36] B. Nelson. *Remote Procedure Call*. PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1981.

[37] NSA Tempest Documents. http://cryptome.org/nsa-tempest.htm, 2003.

[38] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot—a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pages 179–194, 2004.

[39] Protocol Buffers-Google Code. http://code.google.com/apis/protocolbuffers/, 2009.

[40] J. Robin and C. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[41] S. Smith. Outbound Authentication for Programmable Secure Coprocessors. *International Journal on Information Security*, 2004.

[42] S. Smith. *Trusted Computing Platforms: Design and Applications*. Springer, 2004.

[43] S. Smith and S. Weingart. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks*, 31:831–860, April 1999.

[44] S. Stasiukonis. Social Engineering, the USB Way. http://www.darkreading.com/document.asp?doc_id=95556&WT.svl=column1_1, June 2006.

[45] G. Suh, D. Clarke, B. Gassend, M. Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *International Symposium on Microarchitecture (MICRO-36)*, 2003.

[46] K. Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27:761–763, 1984.

[47] J. Tygar and B. Yee. Strongbox: A System for Self-Securing Programs. In *CMU Computer Science: A 25th Anniversary Commemorative*, pages 163–197. Addison-Wesley, 1991.

[48] R. Uhlig et al. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, May 2005.

[49] S. Weingart. Physical Security for the μABYSS System. In *Proceedings of the 1987 Symposium on Security and Privacy*, pages 52–59. IEEE, 1987.

[50] S. Weingart. Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses. In *Cryptographic Hardware and Embedded Systems—CHES 2000*, pages 302–317. Springer-Verlag LNCS 1965, 2000.

[51] S. White and L. Comerford. ABYSS: A Trusted Architecture for Software Protection. In *IEEE Symposium on Security and Privacy*, 1987.

[52] A. C. Yao. How to Generate and Exchange Secrets. In *27th Annual Symposium on Foundations of Computer Science*, pages 162–167, 1986.

Index