16 Hardware-Based Security

We worry about computer security because important social processes keep migrating to distributed computerized settings. When considering security, it's important to take a holistic view—because we care about the security of the social process itself, not only some component. However, in this chapter, we take a reductionist point of view and look at one of the components in particular: the hardware.

Historically, probably the main path toward thinking about hardware security came from considering the protection of computation from adversaries with direct physical access to the computing machinery. We've always liked framing this question in terms of dependency. Alice's interests may depend on certain properties of computation *X*—perhaps integrity of its action or confidentiality of some key parameters. However, if *X* occurs on Bob's machine, then whether these properties hold may depend on Bob. For example, if Bob's machine is a standard PC and Bob is root, then he pretty much has free reign over *X*. He can see and modify data and code at will. As a consequence, preservation of Alice's interests depends on the behavior of Bob, since Bob could subtly subvert the properties that Alice depends on. These circumstances force Alice to trust Bob, whether or not she wants to. If Bob's interests do not coincide with Alice's, this could be a problem.

This main path—reducing Alice's dependency by modifying Bob's computer—leads to several lines of inquiry. The obvious path is using hardware itself to protect data and computation. Another path toward thinking about hardware security comes from considering that computing hardware is the underlying physical

environment for computation. As such, the nature of the hardware can directly influence the nature of the computation it hosts. A quick glance at *BugTraq* [Sec06] or the latest Microsoft security announcements suffices to establish that deploying secure systems on conventional hardware has proved rather hard. This observation raises another question: If we changed the hardware, could we make it easier to solve this problem?

In this chapter, we take a long look at this exciting emerging space.

- Section 16.1 discusses how memory devices may leak secrets, owing to physical attack.
- Section 16.2 considers physical attacks and defenses on more general computing devices.
- Section 16.3 reviews some larger tools the security artisan can use when considering the physical security of computing systems.
- Section 16.4 focuses on security approaches that change the hardware architecture more fundamentally.
- Section 16.5 looks at some future trends regarding hardware security.

(The first author's earlier book [Smi04c] provides a longer—but older—discussion of many of these issues. Chapter 3 in particular focuses on attacks.)

16.1 Data Remanence

One of the first challenges in protecting computers against adversaries with direct physical contact is protecting the stored data. Typically, one sees this problem framed as how a device can hide critical secrets from external adversaries, although the true problem is more general than this, as we discuss later. Potential attacks and defenses here depend on the type of beast we're talking about.

We might start by thinking about *data remanence:* what data an adversary might extract from a device after it has intended to erase it.

16.1.1 Magnetic Media

Historically, nonvolatile magnetic media, such as disks or once ubiquitous tapes, have been notorious for retaining data after deletion. On a physical level, the contents of overwritten cells have been reputed to be readable via magnetic-force microscopy; however, a knowledgeable colleague insists that no documented case exists for any modern disk drive. Nonetheless, researchers (e.g., [Gut96]) and government standards bodies (e.g., [NCS91]) have established guidelines for overwriting cells

in order to increase assurance that the previously stored data has been destroyed. (The general idea is to write a binary pattern, then its complement, then repeat many times.)

A complicating factor here is the existence of many layers of abstraction between a high-level request to delete data and what actually happens on the device in question. In between, many things could cause trouble.

- For example, a traditional filesystem usually breaks a file into a series of chunks, each sized to occupy a disk sector, and distributes these chunks on the disk according to various heuristics intended to improve performance. Some type of index table, perhaps in a sector of its own, indicates where each chunk is. In such a system, when the higher-level software deletes or even shrinks a file, the filesystem may respond by clearing that entry in that file's index table and marking that sector as "free." However, the deleted data may remain on the disk, in this now-free sector. (Issues such as this led to the *object reuse* worries of the Orange Book world of Chapter 2.)
- *Journaling filesystems*, a more advanced technology, make things even worse. Journaling filesystems treat the disk not as a place to store files so much as a place to store a log of changes to files. As with Word's Fast Save option (see Chapter 13), the history of edits that resulted in a file's current state may be available to the adversary inspecting the disk itself.
- Computing hardware has seen a sort of trickle-down (or perhaps smarting-down) effect, whereby traditionally "dumb" peripherals now feature their own processors and computing ability. Disk controllers are no exception to this trend, leading to yet another level of abstraction between the view the computing system sees and what actually happens with the physical media.

16.1.2 FLASH

In recent years, semiconductor FLASH memory (e.g., in USB thumbdrives) has probably become more ubiquitous than magnetic media for removable storage. FLASH is also standard nonvolatile storage in most embedded devices, such as cell phones and PDAs. The internal structure of a FLASH device is a bit more complex than other semiconductor memories (e.g., see [Nii95]). FLASH is organized into *sectors*, each usually on the order of tens or hundreds of kilobytes. When in "read" mode, the device acts as an ordinary ROM. To write a sector, the system must put the FLASH device into write mode, which requires writing a special sequence of bytes, essentially opcodes, to special addresses in the FLASH device. Typically, the stored bits can be written only one way (e.g., change only from 0 to 1). To erase a

sector (e.g., clearing all the bits back to 0), another sequence of magic bytes must be written. Often, FLASH devices include the ability to turn a designated sector into ROM by wiring a pin a certain way at manufacture time.

FLASH gives two additional challenges for system implementers. First, writing and erasing sectors both take nontrivial time; failure, such as power interruption, during such an interval may lead to undetermined sector corruption. Second, each FLASH cell has a relatively small (e.g., 10,000) lifetime of erase-write cycles.

These technical limitations lead to incredible acrobatics when designing a filesystem for FLASH (e.g., [GT05, Nii95]). In order to avoid wearing out the FLASH sectors, designers will use data structures that selectively mark bits to indicate dirty bytes within sectors and rotate usage throughout the sectors on the device. For fault tolerance, designers may try to make writes easy to undo, so that the old version of a file can be recovered if a failure occurs during the nontrivial duration of a write. Even relatively simple concepts, such as a directory or index table, get interesting—if you decide to keep one, then you'll quickly wear out that sector, even if you're clever with the rest of the files.

FLASH architecture has several consequences for security.

- Because of these log-structured and fault-tolerant contortions, old data may still exist in the device even if the higher levels of the system thought it was erased.
- Because an error in a product's ROM can be expensive, at least one vendor includes an undocumented feature to rewrite the ROM sector by writing a magic series of bytes to the chip. (The complexity of the legitimate magic-byte interface makes it hard to otherwise discover such back doors.)
- Because of the large market demand for low-cost thumbdrives and the smarting down of computation into peripherals, much engineering has gone into commercial FLASH drives, leading to a gap between even the API the encapsulated device provides and the internal state.

16.1.3 RAM

Random-access memory (RAM) is the standard medium for memory during active computation. *Dynamic RAM (DRAM)* stores each bit as an electrical charge in a capacitor. Since these charges tend to be short-lived, data remanence is not as much of an issue here. This short lifetime leads to additional functionality: The devices need to continually read and restore the charges, before they decay. (One wonders whether this continual processing of stored data might lead to side-channel exposures.) However, the capacitors do not take much real estate on the chip;

as a consequence, DRAM tends to be favored when large amounts of memory are required.

In contrast, *static RAM (SRAM)* stores each bit via the state in a flip-flop, a small collection of logic gates. This approach takes more real estate but does not require the extra functionality and its extra power. As a consequence, when a device needs memory with the properties of RAM (e.g., none of this sector business) but otherwise nonvolatile, it may end up using battery-backed SRAM, which is sometimes referred to as *BBRAM*.

SRAM, however, is not without remanence issues. Long-term storage of the same bits can cause memory to *imprint* those values and retain them even after power-up. Environmental factors, such as cold temperatures and radiation, can also cause imprinting. (Gutmann [Gut01] and Weingart [Wei00] both provide more discussion of these issues.)

16.1.4 The System

So far, we've discussed properties of the memory medium itself. However, the memory is embedded in the context of a larger system, and this larger context can lead to issues. For example, at many levels in the software stack, software optimization might decide that a write to a data location that will no longer be used is unnecessary and silently eliminate it. This can undo a programmer's efforts to clear sensitive data. Researchers at Stanford recently used a form of virtualization (see Section 16.4.2) to explore this issue of data lifetime in the context of an entire system—and uncovered many surprising cases of data living longer than the designers or programmers intended or believed [CPG+04].

16.1.5 Side Channels

Devices that instantiate computation in the real world must exist as physical machines in the real world. Because of this physical existence, computational actions the device takes can result in real-world physical actions that the designer can easily fail to foresee but that an adversary can exploit. We discussed many examples of this in Section 8.4.

16.2 Attacks and Defenses

16.2.1 Physical Attacks

So far, we've discussed how a computing device may or may not keep secrets from an adversary with physical access. We now discuss some ways an adversary may use physical access to mount an attack. To start with, we might consider the security

perimeter: what the designers regarded as the boundary between the internal trusted part of the system and the external part under the potential control of the adversary.

Individual Chips. Perhaps the first model to consider is the single trusted chip. The designer/deployer wants to trust the internal operation of the chip, but the adversary controls the outside. Over the years, this model has received perhaps the most attention—in the public literature, anyway—owing to the long and widespread use of low-cost *chip cards*—often considered synonymous with *smart cards*—in commercial applications, such as controlling the ability to make telephone calls or to view licensed satellite TV. The ubiquity creates a large community of adversaries; the applications give them motivation; and the cost makes experimentation feasible.

The work of Anderson and Kuhn provides many nice examples of attack techniques on such single-chip devices [AK96, AK97]. Perhaps the most straightforward family of attacks are the many variations of "open up the device and play with it." Various low-cost lab techniques can enable the adversary to open up the chip and start probing: reading bits, changing bits, resetting devices back to special factory modes by re-fusing fuses, and so on. Historically, we've seen a cycle here.

- The vendor community claims that such attacks are either not possible or are far too sophisticated for all but high-end state-sponsored adversaries.
- The adversary community demonstrates otherwise.
- The vendor community thinks a bit, reengineers its defense technology, and the loop repeats.

It's anyone's guess where we will be in this cycle—and whether the loop will keep repeating—when this book is published.

By manipulating the device's environment, the adversary can also use more devious ways to influence the computation of such devices. For an amusing and effective example of attacks, we refer back to Anderson and Kuhn. The device may execute an internal program that brings it to a conditional branch instruction. Let's say that the device compares a register to 0 and jumps to a different address if the two are equal. However, in typical chip card applications, the device obtains its power from an outside source. This means that the adversary can deviously manipulate the power, such as by driving it way out of specification. Generally speaking, the CPU will not function correctly under such conditions. If the adversary applies such a carefully timed spike at the moment the device is executing this comparison instruction, the adversary can cause the CPU to always take one direction of the branch—whether or not it's correct. Finding examples where such

an attack lets the adversary subvert the correctness of the system is an exercise for the reader.

In some sense, such environmental attacks are the flip side of side-channel attacks. Rather than exploiting an unexpected communication path coming out of the device, the adversary is exploiting an unexpected communication path going into it. Another example of this family of attack is *differential fault analysis (DFA)*, sometimes also known as the *Bellcore attack*. Usually framed in the context of a chip card performing a cryptographic operation, this type of attack has the adversary somehow causing a transient hardware error: for example, by bombarding the chip with some kind of radiation and causing a gate to fail. This error then causes the chip to do something other than the correct cryptographic operation. In some situations, the adversary can then derive the chip's critical secrets from these incorrect results.

Bellcore attacks were originally suggested as a theoretical exercise (e.g., [BDL97]). However, they soon became a practical concern (e.g, [ABF+03]), to the point where countermeasures became a serious concern. How does one design a circuit to carry out a particular cryptographic operation but that also doesn't yield anything useful to the adversary if a transient error occurs? Some researchers have even begun formally studying this model: how to transform a circuit so that an adversary who can probe and perhaps alter the state of a limited subset of wires still cannot subvert the computation [ISW03]. We touch on these attacks again in Section 16.5.

Larger Modules. Multichip modules provide both more avenues for the attacker and more potential for defense.

Getting inside the chassis is the first step. Here we see another cat-and-mouse game, featuring such defenses as one-way bolt heads and microswitches on service doors, and corresponding counterattacks, such as using a pencil eraser as a drill bit or putting superglue on the microswitch after drilling through the door.

An attacker who can get inside the chassis might start monitoring and manipulating the connections on the circuit boards themselves. The attacker might hook logic analyzers or similar tools to the lines or insert an *interposer* between a memory or processor module and the circuit board, allowing easy monitoring and altering of the signals coming in and out. Other potential attacks misusing debugging hooks include using an *in-circuit emulator (ICE)* to replace a CPU and using a *JTAG* port to suspend execution and probe/alter the internal state of a CPU.[1]

1. JTAG stands for Joint Test Action Group, but that's not important. What is important is that the name denotes an industry standard for physical interfaces to ease testing of hardware.

The attacker might also exploit properties of the internal buses, without actually modifying hardware. For one example, the PCI bus includes a *busmastering* feature that allows a peripheral card to communicate directly with system memory, without bothering the CPU. Intended to support *direct memory access (DMA)*, occasionally a desirably form of I/O, busmastering can also support malicious DMA, through which a malicious PCI card reads and/or writes memory and other system resources illicitly.

API Attacks. When focusing on these subtle ways that an adversary might access secrets by sneakily bypassing the ways a system might have tried to block this access, it's easy to overlook the even more subtle approach of trying to use the front door instead. The APIs that systems offer through which legitimate users can access data are becoming increasingly complex. A consequence of this complexity can be extra, unintended functionality: ways to put calls together that lead to behavior that should have been disallowed. Bond and Anderson made the first big splash here, finding holes in the API for the *Common Cryptographic Architecture (CCA)* application that IBM offered for the IBM 4758 platform [BA01]. More recently, Jonathan Herzog has been exploring the use of automated formal methods to discover such flaws systematically [Her06].

16.2.2 Defense Strategies

As with attacks, we might start discussing defenses by considering the trust perimeter: what part of the system the designer cedes to the adversary.

Chips. As we observed earlier, attacks and defenses for single-chip modules have been a continual cat-and-mouse game, as vendors and adversaries take turns with innovation. In addition, some new techniques and frameworks are beginning to emerge from academic research laboratories. Researchers have proposed *physical one-way functions:* using a device's physical properties to embody functionality that, one hopes, cannot be accessed or reverse engineered any other way. The intention here is that an adversary who tries to use some type of physical attack to extract the functionality will destroy the physical process that generated the functionality in the first place.

In an early manifestation of this concept, researchers embedded reflective elements within a piece of optical-grade epoxy [PRTG02]. When entering this device, a laser beam reflects off the various obstacles and leaves in a rearranged pattern. Thus, the device computes the function that maps the input consisting of the laser

angle to the output consisting of the pattern produced by that input. Since the details of the mapping follow randomly from the manufacturing process, we call this a *random function:* the designer cannot choose what it is, and, one hopes, the adversary cannot predict its output with any accuracy, even after seeing some reasonable number of x, $f(x)$ pairs. (Formalizing and reasoning about what it means for the function to resist reverse engineering by the adversary requires the tools of theoretical computer science—recall Section 7.1 or see the Appendix.)

It's hard to use these bouncing lasers in a computing system. Fortunately, researchers [GCvD02] subsequently explored *silicon physical random functions (SPUF)*, apparently from the earlier acronym *silicon physical unknown functions*. The central idea here is that the length of time it takes a signal to move across an internal connector depends on environmental conditions, such as temperature and, one hopes, on random manufacturing variations. If we instead compare the relative speed of two connectors, then we have a random bit that remains constant even across the environmental variations. Researchers then built up more elaborate architectures, starting with this basic foundation.

Outside the Chip. Even if we harden a chip or other module against the adversary, the chip must still interact with other elements in the system. The adversary can observe and perhaps manipulate this interaction and may even control the other elements of the system. A number of defense techniques—many theoretical, so far—may apply here. However, it's not clear what the right answer is. Figuring out the right balance of security against performance impact has been an area of ongoing research; many of the current and emerging tools we discuss later in this chapter must wrestle with these design choices.

For example, suppose that the device is a CPU fetching instructions from an external memory. An obvious idea might be to encrypt the instructions and, of course, check their integrity, in order to keep the adversary from learning details of the computation. Although perhaps natural, this idea has several drawbacks. One is figuring out key management: Who has the right to encrypt the instructions in the first place? Another drawback is that the adversary still sees a detailed trace of instruction fetches, with only the opcodes obfuscated. However, there's nothing like the real thing—the most damning indictment of this technique is the way Anderson and Kuhn broke it on a real device that tried it [AK96].

We might go beyond this basic idea and think about using external devices as memory, which makes sense, since that's where the RAM and ROM will likely be. What can the adversary do to us? An obvious attack is spying on the memory

contents; encryption can protect against this, although one must take care with using initialization vectors (IVs) or clever key management to prevent the same plaintext from going to the same ciphertext—or the same initial blocks from going to the same initial blocks. (Note, however, that straightforward use of an IV will cause the ciphertext to be one block larger than the plaintext, which might lead to considerable overhead if we're encrypting on the granularity of a memory word.)

Beyond this, two more subtle categories of attacks emerge:

1. *Learning access patterns.* The adversary who can see the buses or the memory devices can see what the trusted chip is touching when. One potential countermeasure here lies in aggregation: If it has sufficient internal storage, the chip can implement virtual memory and *cryptopage* to the external memory, treated as a backing store [Yee94].

   The world of crypto and theory give us a more thorough and expensive technique: *oblivious RAM* (ORAM) [GO96]. In a basic version, the trusted device knows a permutation π of addresses. When it wants to touch location i_1, the device issues the address $\pi(i_1)$ instead. If it only ever touches one address, then this suffices to hide the access pattern from the adversary. If it needs to then touch an i_2, then the device issues $\pi(i_1)$ and then $\pi(i_2)$—unless, of course, $i_2 = i_1$, in which case the device makes up a random i_2' and issues $\pi(i_2')$ instead. The adversary knows that two addresses were touched but doesn't know which two they were or even whether they were distinct. To generalize this technique, the device must generate an *encrypted shuffle* of the external memory; the kth fetch since the last shuffle requires touching k memory addresses. (One might wonder whether we could turn around and use the same technique on the k fetches—in fact, Goldreich and Ostrevsky came up with an approach that asymptotically costs $O(\log^4 n)$ per access.)
2. *Freshness of contents.* Earlier, we mentioned the obvious attack of spying on the stored memory and the obvious countermeasure of encrypting it. However, the adversary might also change memory, even if it's encrypted. An effective countermeasure here is less obvious. Naturally, one might think of using a standard cryptographic integrity-checking technique, such as hashes or MACs, although doing so incurs even more memory overhead. However, if the device is using the external memory for both writing and reading, then we have a problem. If we use a standard MAC on the stored data, then we can replace the MAC with a new value when we rewrite the memory. But then nothing stops the adversary from simply replacing our new value-MAC pair

with an older one! We could stop this attack by storing some per location data, such as the MAC, inside the trusted device, but then that defeats the purpose of using external memory in the first place.

Two techniques from the crypto toolkit can help here. One is the use of *Merkle trees* (recall Section 7.6 and Figure 7.19). Rather than storing a per location hash inside the trusted device, we build a Merkle tree on the hashes of a large set of locations and store only the root inside the device. This approach saves internal memory but at the cost of increased calculation for each integrity/freshness check. Another idea is to use *incremental multiset hashing*, a newer crypto idea, whereby the device calculates a hash of the contents of memory—"multiset"—but can do so in an incremental fashion. (Srini Devadas' group at MIT came up with these ideas—for example, see [CDvD+03, SCG+03].)

The preceding approaches considered how the trusted device might use the rest of the system during its computation. We might also consider the other direction: how the rest of the system might use the trusted device. A general approach that emerged from secure coprocessing research is *program partitioning:* sheltering inside the trusted device some hard to reverse engineer core of the program but running the rest of the program on the external system. Doing this systematically, for general programs, in a way that accommodates the usually limited power and size of the trusted device, while also preserving overall system performance, while also being secure, appears to be an open problem.

However, researchers have made progress by sacrificing some of these goals. For example, theoreticians have long considered the problem of *secure function evaluation (SFE)*, also known as *secure multiparty computation*. Alice and Bob would like to evaluate a function f which they both know, on the input (x_A, x_B), which they each know (respectively), but don't want to share. In 1986, Yao published an algorithm to do this—an inefficient algorithm, to be sure, but one that works [Yao86]. 2004 brought an implementation—still inefficient, but we're making progress [MNPS04].

The economic game of enforcing site licenses on software also used to manifest a version of this program-partitioning problem. Software vendors occasionally provide a *dongle*—a small device trusted by the vendor—along with the program. The program runs on the user's larger machine but periodically interacts with the dongle. In theory, absence of the dongle causes the program to stop running. Many software vendors are moving toward electronic methods and are abandoning the

hardware dongle approach. For example, many modern PC games require an original copy of the game CD to be inserted into the machine in order to play the game; a copied CD generally will not work.

Modules. Building a module larger than a single chip gives the designer more opportunity to consider hardware security, as a system. For example, a larger package lets one more easily use internal power sources, environmental sensing, more robust filtering on the power the device demands from external sources, and so on.

However, colleagues who work in building "tamper-proof hardware" will quickly assert that there is no such thing as "tamper-proof hardware." Instead, they advocate looking at a systems approach interleaving several concepts:

- *Tamper resistance.* It should be hard to penetrate the module.
- *Tamper evidence.* Penetration attempts should leave some visible signal.
- *Tamper detection.* The device itself should notice penetration attempts.
- *Tamper response.* The device itself should be able to take appropriate countermeasures when penetration is detected.

Integrating these concepts into a broader system requires considering many tradeoffs and design issues. Tamper evidence makes sense only if the deployment scenario allows for a trustworthy party to actually observe this evidence. Tamper resistance can work in conjunction with tamper detection—the stronger the force required to break into the module, the more likely it might be to trigger detection mechanisms. Tamper response may require consideration of the data remanance issues discussed earlier. What should happen when the adversary breaks in? Can we erase the sensitive data before the adversary can reach it? These questions can in turn lead to consideration of protocol issues—for example, if only a small amount of SRAM can be zeroized on attack, then system software and key management may need to keep larger sensitive items encrypted in FLASH and to be sure that the sensitive SRAM is regularly inverted. The choice of tamper-response technology can also lead to new tamper-detection requirements, since the tamper-response methods may require that the device environment remain inside some *operating envelope* for the methods to work.

Antitamper, Backward. Recently, a new aspect of tamper protection has entered the research agenda. U.S. government agencies have been expressing concern about whether the chips and devices they use in sensitive systems have themselves been tampered with somehow—for example, an adversary who infiltrated the design and

build process for a memory chip might have included (in hardware) a Trojan horse that attacks its contents when a prespecified signal arrives. We can find ourselves running into contradictions here—to protect against this type of attack, we might need to be able to probe inside the device, which violates the other type of tamper protection. (Some recent research here tries to use the techniques of side-channel analysis—typically used to *attack* systems—in order to discover the presence of hardware-based Trojan horses; the idea is that even a passive Trojan will still influence such things as power consumption. [ABK+07].)

Software. So far in this section, we've discussed techniques that various types of trusted hardware might use to help defend themselves and the computation in which they're participating against attack by an adversary. However, we might also consider what software alone might do against tamper. The toolkit offers a couple of interesting families of techniques.

- *Software tamper-resistance* (e.g., [Auc96]) techniques try to ensure that a program stops working correctly if the adversary tampers with critical pieces—for example, the adversary might try to run the program without a proper license. Effective use of dongles often requires some notions of software tamper resistance. As noted earlier, if the program simply checks for the dongle's presence and then jumps to the program start, then the adversary might simply bypass this check—so the tamper response needs to be more subtle. Related to this topic are techniques to produce binary code that is difficult to disassemble.
- *Software-based attestation* techniques (e.g., [SLS+05, SPvDK04]) try to assure an external relying party that a piece of software is running on a particular platform in a trustworthy way. The basic idea is that the relying party knows full operational details of the target system and crafts a checksum program that requires using all the resources of the system in order to produce a timely but correct response; a trusted path between the relying party and the target system is usually assumed. These techniques are still early but promising.

16.3 Tools

The previous section discussed foundations: basic issues of hardware attacks and defenses. However, when putting together a secure system, one typically thinks of larger-scale components. Rather than worrying only about how to build a chip that resists an attacker, one might worry about how to use an attack-resistant chip to do

something useful within a larger system. In this section, we take a look at some of components in the toolbox.

16.3.1 Secure Coprocessors

If we're thinking about trying to protect computation from an adversary with direct physical access to the computer, the most "natural" approach might be to think about putting armor around the entire computer. However, since effective physical security raises issues about heat dissipation and internal maintenance, we usually can't count on armoring the entire computer system in question, so a more practical compromise is to armor a smaller subsystem and use that in conjunction with a larger *host*. This is the approach taken by *secure coprocessors*. Commercial examples include the IBM 4758 [SW99] and its more recent follow-on, the IBM 4764 [AD04]. (As the reader may conclude from checking out the citations in the bibliography, yes, the authors of this book had something to do with this.)

Generally, this type of device works by hiding secrets inside the armored device and using an interleaving of tamper-protection techniques to ensure that, under attack, the secrets are destroyed before the adversary can get to them. Owing to the relative ease of zeroizing SRAM compared to other forms of storage, secure coprocessors typically end up with a tiered memory architecture: a small amount of battery-backed SRAM contains the nonvolatile but tamper-protected secret; larger DRAM contains runtime data, and FLASH holds nonvolatile but non-secret data.

As a consequence, perhaps the most natural application of a secure coprocessor is to obtain confidentiality of stored data. This can be useful. However, one can also use this "protected secret" architecture to provide other properties. For example:

- *Integrity of public data.* If the secret in question is the private half of a key pair, then the coprocessor can use it to sign statements. A relying party that verifies the signature and believes that the device's physical security works and software is trustworthy, can believe that this statement came from an untampered device. If the statement pertains to the value of a stored data item, then the relying party can trust in the integrity of that value. This property may be useful in such scenarios as metering.
- *Integrity of executing program.* Is the device still running the correct, untampered software? A side effect of the private key approach just discussed is that the relying party can also verify that the software inside the device is still correct—if an adversary has tampered with it, then the private key would have, in theory, been zeroized and thus not available to the modified software.

This property can be useful in many scenarios, such as a trustworthy SSL-protected Web server. With more complex devices that permit updates and reinstallation of software and permit nontrivial software architectures, making this scheme work can become rather tricky. This idea of *outbound authentication*—enabling the untampered entity to authenticate itself as such to the outside world—foreshadowed the subsequent emphasis on *attestation*.

- *Privacy of program execution.* Some scenarios call for the program itself to be public but its execution to be private—that is, not only selected parameters but also operational details, such as which branch is taken after a comparison. For example, consider an auction. The program may need to be public, as all participants need to trust that the program evaluating the bids works correctly. However, exactly what it does when it runs on the secret bids should be secret; otherwise, observers would know details of the bids.

  Outbound authentication, combined with a self-contained computing environment, can provide this property.
- *Secrecy of program code.* Typically, the device may store its software in internal FLASH. However, the device could store much of this software in encrypted form and use its protected secret to decrypt it into DRAM before execution—thus using the protected-secret architecture to provide secrecy of program executables. This property may be useful for protecting proprietary pricing algorithms for insurance or pharmaceuticals.

Using a secure coprocessor in real-world applications may require dealing with some subtle design and architecture issues, owing to the exigencies of commercially feasible physical security. One basic problem is that the device may be too small to accommodate the necessary data; this problem drives some current research, as we discuss later. Another problem arises from the typical lack of human I/O on devices. If an enterprise runs a stand-alone application that has one trusted coprocessor installed but depends on input from an untrustworthy host, then the enterprise may not be benefiting much from the physical security. Nearly anything the adversary might have wanted to do by attacking the coprocessor can be achieved by attacking the host. The true value of the physical security comes into play when other parties and/or other trusted devices come into the picture: for example, remote clients connecting to a coprocessor-hardened server.

Another real-world issue with using a commercial secure-coprocessor platform is believing that it works. In our case, we had it validated against FIPS 140-1; however, going from such a validation to the conclusion that a system using such a device is sufficiently secure is a big step—see Chapter 11.

16.3.2 Cryptographic Accelerators

As discussed earlier in the book, cryptography is a fundamental building block of security in many modern computing scenarios. However, as Chapter 7 made clear, it is based on tasks that are by no means easy for traditional computers. For a basic example, RSA requires modular exponentiation: taking X and Y to X^Y mod N, where X, Y, and N are all very large integers. By current standards, RSA requires integers at least 1024 bits long to be deemed secure; currently, however, standard desktop computers operate on 32-bit words. Implementing 1024-bit modular exponentiation on a 32-bit machine is rather inefficient; this inefficiency can become an obstacle for applications, such as SSL Web servers, that must do this repeatedly.

These issues drive the idea of creating special-purpose hardware to accelerate such otherwise inefficient operations. Hardware for such operations as symmetric encryption and hashing can also be inserted in-line with data transmission (e.g., in a network card or in a disk drive) to make use of encryption in these aspects of system operation more affordable. (For example, building hardware acceleration for digital signature generation and verification into edge routers can greatly improve the performance cost of S-BGP compared to standard BGP—recall Chapter 5.)

Both the nature and the applications of cryptography introduce issues of physical security for cryptographic accelerators. For one thing, cryptographic parameters, such as private keys, may be long-lived, mission-critical data items whose compromise may have serious ramifications. For another thing, application domains, such as banking and the postal service, have a long history of relying on physical security as a component of trying to assure trustworthiness. As a consequence, cryptographic accelerators may tout tamper protection and feature APIs to protect installation and usage of critical secrets. As we noted, such devices tend to be called *hardware security modules (HSMs)* in the literature and in discussions of best practices for such application installations as certification authorities. The same architecture issues we noted earlier apply here as well. Physical security may protect against an adversary directly extracting the keys from the device and may protect against more esoteric attacks, such as subverting the key-generation code the device uses in the first place, in order to make the "randomly" generated keys predictable to a remote adversary. However, physical security on the HSM does not protect against attacks on its host.

For using cryptographic accelerators or HSMs in the real world, we advise consideration of many questions.

- Should you trust that the HSM works? Researchers have shown that one can build a crypto black box that appears to work perfectly but has adversarial back doors, like the one discussed earlier [YY96]. Here, we recommend that

you look for FIPS validations—both of the overall module (e.g., via FIPS 140-*N*) and of the individual cryptographic algorithms used (recall Chapter 11).

- Should you trust that the HSM works too well? From a perhaps a straightforward security perspective, it's better for a device to have false positives—and destroy secrets even though no attack was occurring—than the other way around. From a business perspective, however, this may be a rather bad thing. The necessity to preserve the *operational envelope* in effective tamper protection may create even more opportunities for such false positives (e.g., if the building heat fails at Dartmouth College in the winter, an IBM 4758 would not last more than a day). Using HSMs requires thinking beforehand about continuity of operations.
- What if the manufacturer goes out of business or the device reaches its end of life? In order to make its physical security mean something, an HSM design may make it impossible to export private keys to another type of device. However, what happens should the vendor cease supporting this HSM? (This happened to colleagues of ours.)
- Exactly how can you configure the cryptographic elements? Having hardware support for fast operations does not necessarily mean that you can do the combination of operations you would like to. For example, the IBM 4758 Model 2 featured fast TDES and fast SHA-1, both of which could be configured in-line with the buffers bringing data in or through the device. Doing cryptography this way on large data was much faster than bringing into the device DRAM and then using the relatively slow internal architecture to drive the operation. However, in practical settings, one usually does not want *just* encryption: One wants to check integrity as well. One natural way to do this might be to hash the plaintext and then encrypt it along with its hash. However, doing something like this with the fast IBM hardware requires being able to bring the data through the TDES engine and then sneak a copy of the plaintext into the hash engine on its way out. Unfortunately, our fast hardware did not support this!
- What if new algorithms emerge? For example, the TDES engine in the IBM 4758 Model 2 includes support for standard chaining, such as CBC. Subsequently, Jutla invented a slower chaining method that provided integrity checking for free [Jut01]. We would have liked to use this chaining method, but the hardware did not support it. For another example, one need only consider the recent demise of MD5 hashing and fears of the future demise of SHA-1.

- Should you believe performance benchmarks? The problem here is that cryptographic operations may feature several parameters; in practice, many operations may be joined together (e.g., signatures or hybrid encryption); and HSMs may include internal modules, thus confusing which boundaries we should measure across.

  For example, if one wants to attach a number to an implementation of a symmetric cryptosystem, the natural measure might be bytes per second. IBM did this for the DES engine in the IBM 4758. A customer complained; on examination, we found that the touted speed was what one could get if operations were done with very long data items. Informally, the device had a per byte cost on the data as well as a per operation cost on the overhead of setting up the keys and such. For small data, the per operation cost dominates—and the effective per byte cost could drop an order of magnitude or more.

16.3.3 Extra-CPU Functionality

These armoring approaches run into some fundamental limitations. It seems that the computational power of what can fit inside the armor always lags behind the power of a current desktop system. This delta is probably an inevitable consequence of Moore's Law (see Section 16.5.3) and the economics of chip manufacturing: What gets packaged inside armor lags behind the latest developments.

This situation raises a natural question: Can we use hardware techniques to improve the security of general systems without wrapping the CPUs in armor? In the commercial and research space here, the general trend is to use hardware to increase assurance about the integrity and correctness of the software on the machine.

Boot-Time Checking. Currently, the dominant approach is to consider boot-time protections. Figure 16.1 sketches an example sequence of what software gets executed when a system boots. The time order of this execution creates a dependency order: If software module S_1 executes before software module S_2, then correct execution of S_2 depends on S_1; if the adversary attacks or modifies S_1, then maybe it will change S_2 before loading it, or maybe it will load something else altogether.

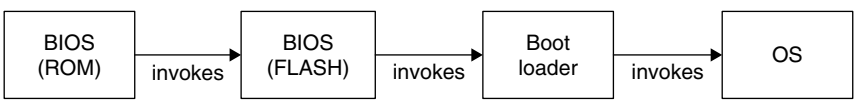

Figure 16.1 At boot time, a well-defined sequence of software modules get executed.

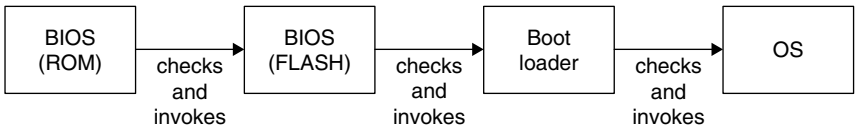

Figure 16.2 In the typical approach to system integrity checking, each element in the boot sequence checks the next before invoking it.

Boot-time approaches exploit the inductive nature of this sequence. By magic, or perhaps by hardware, we check the integrity and correctness of the first element of this chain. Then, before we grow the chain with a new element, a chain element that has been already checked checks this next candidate element. Figure 16.2 sketches an example. (In our 4758 work, we got rather formal about this and included hardware elements in this "chain.")

At the end of the process, we might have some assurance that the system is running correct, unaltered software—that is, if we have some way of knowing whether this verification process succeeded. (One will see the terms *trusted boot* and *secure boot* used for this process—sometimes as synonyms, sometimes to denote slightly different versions of this idea.)

One way to know whether verification succeeded is to add hardware that releases secrets depending on what happens. In the commercial world, the *Trusted Computing Group (TCG)* consortium[2] has developed—and still is developing, for that matter—an architecture to implement this idea in standard commercial machines. The TCG architecture adds a *trusted platform module (TPM)*—a small, inexpensive chip—to the motherboard. At the first level of abstraction, we can think of the TPM as a storehouse that releases secrets, depending on the state of the TPM's *platform configuration registers (PCRs).* Each PCR can contain an SHA-1 hash value but has some special restrictions regarding how it can be written.

- At boot time, the PCRs are reset to 0s.[3]
- If a PCR currently contains a value v, the host can *extend* a PCR by providing a new value w. However, rather than replacing v with w, the TPM replaces v with the hash of the concatenation of v with w:

$$PCR \quad \longleftarrow \quad H(PCR \,||\, w).$$

2. The TCG succeeded the *Trusted Computing Platform Alliance (TCPA)*; sometimes, one still sees the acronym of the predecessor consortium used to denote this architecture.
3. The latest specification of the TPM explores some other special conditions under which a PCR can be reset; expect further developments here.

This approach to "writing" PCRs allows the system to use them to securely measure software and other parameters during the boot process (see Figure 16.1). At step $i - 1$, the system could hash the relevant software from module i and store this hash in PCR i. Suppose that module 3 is supposed to hash to h_3 but that, in fact, the adversary has substituted an untrustworthy version that hashes instead to h'_3. If the PCRs permitted ordinary writing, nothing would stop adversarial software later from simply overwriting h'_3 with h_3 in PCR 3. However, because the PCRs permit writing only via hash extension, the PCR will contain $H(0 \mid\mid h'_3)$; if the hash function is secure, the adversary will not be able to calculate a v such that

$$H\big(H(0 \mid\mid h'_3) \mid\mid v\big) = H(0 \mid\mid h_3).$$

In fact, this hash-extension approach allows the system to measure platform configuration into the PCRs using two dimensions. The system could use each PCR i to record the hash of a critical piece of the boot process. However, the system could also record a sequence of measurements within a single PCR, by successively hash-extending in each element of the sequence. By the properties of cryptographically secure hash functions, the end result of that PCR uniquely reflects that sequence of values, written in that order.

As mentioned, we can think of the TPM as essentially a place to store secrets. When we store a secret here, we can tie it to a specified subset of the PCRs and list a value for each. Subsequently, the TPM will reveal a stored secret only if each PCR in that subset has that specified value. (Note that we qualify this statement with "essentially": The actual implementation of this functionality is a bit more convoluted.) If such a secret is an RSA private key, then it can be stored with a further provision: When the PCRs are correct, the TPM will *use* it on request from the system but will never actually release its plaintext value.

The ability of the PCRs to reflect system configuration and the ability of the TPM to bind things such as RSA private keys to specific configurations enables several usage scenarios.

- Binding a key to a software configuration on that machine enables us to do similar things to what we did with secure coprocessors. The entity consisting of that software on that device can now *authenticate* itself, make verifiable statements about things, and participate in cryptographic protocols.
- If we cook things up so that we have a trusted entity that is much smaller than the entire platform, we can use a TPM-bound private key to make signed *attestations* about the rest of the platform configuration, as expressed by the

PCRs. In the TCG architecture, this entity is part of the TPM itself, but it could also be a separate software module protected by the TPM.

Moving from a rather special-purpose and expensive device (a secure coprocessor) to a generic, ubiquitous platform (standard desktops and laptops) changes the flavor of potential applications, as well. For example, moving Yee's partitioned-computation idea from a coprocessor to an encrypted subsystem or tables (protected by a TPM) can enable a software vendor to lock an application to a particular machine or OS. Attestation can enable an enterprise to shunt unpatched machines to a remedial network, thus promoting better network hygiene—this is called *trusted network connect (TNC)*. Attestation might also enable a powerful corporation to monitor everything on your machine. (Of course, all these scenarios are based on the assumption that the adversary cannot subvert the TPM's security protections!)

Realizing this approach in the real world requires worrying about exactly how to map platform configuration into the PCRs. This part of the design is rather complex and keeps changing, so we won't bother going through it all here. The initial BIOS reports itself to a PCR; as a consequence, the BIOS can break everything and thus is called the *root of trust measurement (RTM)*.[4] Subsequently, things already measured turn around and measure other things; what they are and which PCR they get measured into appear to be determined both by platform-specific specifications and random vendor choices. Platform elements also factor into the hashes; we discovered that doing things as simple as removing a keyboard or replacing a memory card caused the PCRs to change.

As Chapter 4 described, however, the software that comprises a particular application running on a contemporary operating system is by no means a monolithic entity or even a simple stack. How to glue TPM measurements to this complex structure is an area of ongoing research. In our early work here, we introduced a level of indirection—the TPM protects a trusted kernel-based module, which in turn evaluates higher-level entities [MSMW03, MSWM03]. In contrast, our colleagues at IBM Watson extended the hash-extension idea all the way up into Linux application environments [SZJv04].

We stress again that this is an area of active research by many parties. Stay tuned. In particular, as this book goes to press, researchers have developed ways to break the security of current TPM-based PCs simply by using a wire to ground the reset line on the *Low Pin Count (LPC)* bus that connects the TPM to the rest of the system.

4. So no, in this case, the acronym RTM does not stand for Read the Manual or Robert Tappan Morris.

This fools the TPM into thinking that the system has rebooted, at which point, the TPM resets all its PCRs, and the host can feed it measurements that simulate booting of the system it would like to impersonate. It looks as though Bernard Kauer [Kau07] got there first, but we were the first to do it on YouTube [Spa].

Runtime Checking. Using hardware to assist with runtime checks of platform integrity is an area that has also received renewed interest lately. *CoPilot*, an academic project currently being commercialized, is good example of this [PFMA04]. One adds to the standard platform a separate PCI card with busmastering capabilities, so it can take over the PCI bus and probe system memory. At regular intervals, this auxiliary card probes the system memory and looks for signs of malware and corruption.

Realizing this approach in the real world requires intricate knowledge of what the system memory image should look like and requires that what image the card sees is the same reality the host CPU sees. Neither of these tasks is trivial. For example, rootkits typically attack systems not by inserting themselves into something big and relatively static, like executable code, but rather by making subtle modifications to dynamic data structures. The fact that these data structures are *supposed* to change makes it hard for the coprocessor to determine when bad changes have occurred. For another example, malware might restore correct-looking data structures when the coprocessor examines memory or might even maintain a decoy set of structures where the coprocessor expects to find them. Combating this latter set of issues may require using the software-based attestation ideas from earlier to establish a dynamic root of trust within the host CPU.

Strictly speaking, the runtime approach is not necessarily disjoint from the boot-time approach. As the experimental approaches we just discussed illustrate, a boot-time-verified module can easily turn around and verify changes and events during runtime. Even standard uses of a TPM can update PCRs during runtime. As we also mentioned earlier, the TCG is currently examining approaches whereby some PCRs can be reset during special conditions at runtime; such an approach could also extend to doing regular remeasurements during runtime.

Other Approaches. So far, we've looked at approaches that use hardware either to directly harden the traditional computing platform or to detect tampering afterward. Ongoing research has been looking at more unconventional approaches: transforming the computation somehow so that a conventional, untrusted host does most of the work, but a smaller, trusted unit participates in such a way as to still provide the overall security property. We offer some examples.

- Cryptographic operations can lend themselves to situations in which part of the work can be blinded and then outsourced to a less trusted host. This approach might provide higher throughput and lower latency, while still protecting the private keys within a small hardware TCB.
- Our own *tiny trusted third party* project (e.g., [IS06]) builds on ORAM and Yao's secure multiparty computation to compile a program into a blinded circuit, which a fast untrusted host can execute with the assistance of a small piece of special-purpose hardware. This approach might provide privacy of computational details, even if the computation doesn't fit inside a small hardware TCB.
- In general, one might speculate about the space of functions in which calculating an answer requires significant resources, but verifying it requires very little. Can we build a method to provide integrity in such calculations, with only limited trusted hardware?

At some point, this approach starts to merge into the partitioned computation model with secure coprocessors.

16.3.4 Portable Tokens

It's almost a cliche that computing hardware has been getting small enough and cheap enough that substantial computing power now fits in a pocket. The truth that underlies this cliche also affects hardware-based security. Putting substantial computing and memory, perhaps with physical security, in a package that users can carry around is economically feasible in many situations; the near-ubiquity of USB slots on PCs and laptops—and the emerging ubiquity of Bluetooth and other forms of *near-field communication (NFC)*—make interaction with the standard computing environment rather easy.

Such devices have many security applications. They can be one factor for multifactor authentication. In enterprise-wide PKI installations, users might carry and wield private keys from a portable device rather than trying to bring data around. Perhaps a user's portable device could verify the integrity of a broader and untrusted system (e.g., [SS05]).

Such devices can also enable another type of security application: *honey-tokens*. Black-hat teams have penetrated enterprises by distributing "abandoned" USB memory sticks—with Trojan horses—in the parking lot. Employees of the target enterprise find the memory sticks, bring them inside the enterprise, insert them into computers, and unintentionally invoke the Trojan; the testers thus succeed in

running their own code with insider privileges on inside-the-firewall systems. (See [Sta06] for more information and some commentary by such a pen tester.)

16.4 Alternative Architectures

So far, we've considered hardware additions for security that start with a computing environment based on a traditional CPU and then either put armor around it or put armored devices next to it. However, many active areas of current research—and also current industrial development—are exploring changing the traditional CPU instead. Some of this work is explicitly motivated by security; some has other motivations but still has relevance to security.

16.4.1 The Conventional Machine

In Section 4.1.1, we reviewed the basic architecture of a conventional system. As Figure 4.1 showed, memory is an array of indexed locations; let's say that each location is 1 byte wide. The CPU interacts with a memory location by issuing the address of that location on the address bus and indicating the nature of the interaction (e.g, read or write) on a control line. The data in question is then transferred on the data bus; the direction of this transfer depends on whether the operation is a read or a write.

Programs are stored in memory like anything else. The CPU fetches an instruction from memory, internally decodes it, and carries out its operation, which may involve additional reads or writes. The CPU then proceeds to fetch the next instruction.

Current conventional architectures differ from this simple one in three fundamental ways.

1. *Memory management.* In this naive model, the address that the CPU issues is the address that the memory sees. As a consequence, when multitasking, each separate program or process must be aware of the addresses the other ones are using. This clearly creates problems for security and fault tolerance, as well as general ease of programming.

   To avoid these problems and to enable lots of other flexibility, modern systems introduce a level of indirection: A *memory-management unit (MMU)* translates the virtual or logical addresses the CPU issues into physical addresses the memory sees (see Figure 4.2). The MMU can also enforce restrictions, such as read-only, by failing to translate the address, for a write request. The MMU, in conjunction with OS trickery, can enforce more exotic

models as well, such as copy on write, whereby memory is shared between two processes until one tries to write to it.

When changing the process or memory domain currently active, the CPU can also instruct the MMU to change *address spaces:* the memory image seen by the CPU.

2. *Privileged instructions.* This is a security book, so our natural inclination is to look at everything—including address translation—from a security perspective. From this perspective, a natural question is: What's the point of using memory management to protect address spaces from rogue programs on the CPU, if the CPU itself is responsible for controlling and configuring the MMU?

   This line of thinking led to the introduction of *privilege levels*. (a) The CPU has some notion of its current privilege level. (b) What an instruction or operation does or whether it's even permitted depends on the current privilege level. (c) Transitions between privilege levels—in particular, translations to greater privilege—must be carefully controlled.

   In the standard textbook model today, a CPU has two[5] privilege levels: user and kernel—or, sometimes, unprivileged and privileged, respectively. Typically, important protection-relevant tasks, such as changing MMU settings, can be done only in kernel mode. As discussed in Chapter 4, user-level code can transition to kernel mode only via a *system call*, or *trap*, that, via hardware, changes mode but also transfers control to specially designated, and one hopes, trusted code. The standard model uses the terms *user* and *kernel* for privileges because of the general intention that operating system code runs in kernel mode, that code from ordinary users runs in user mode, and that the operating system protects itself (and the users) from the users.

3. *Caching.* In the naive model, a memory location, once translated, "lives" at some place in ROM or RAM; in this simple model, the CPU does one thing at a time and accesses memory as needed. Modern systems have achieved significant performance improvements, however, by throwing these constraints out the window. Memory no longer needs to be bound to exactly one physical device; rather, we can try to *cache* frequently used items in faster devices. Consequently, CPUs may have extensive internal caches of memory—and then play various update games to make sure that various

5. Many variations have been explored throughout history; even the standard x86 architecture today has four levels, ring 0 through ring 3. As discussed earlier, in practice, only ring 0 and ring 3 get used—as user and kernel, respectively.

devices, such as other CPUs, see consistent views of memory. Caching enables a CPU to execute sequences of instructions without touching external memory. Caching also motivates the development of fancy heuristics, such as *prefetching*, to attempt to make sure that the right items are in the cache when needed. Processors sometimes separate instruction caches from data caches.

Systems achieve additional performance improvement by doing away with the notion that the CPU execute the instructions one at a time, as written in the program. One way this is done is via *pipelining:* decomposing the execution of an instruction into several stages and making sure that the hardware for each stage is always busy. Another innovation is *superscalar* processing—after decomposing the instruction execution into stages, we add extra modules for some of the stages (e.g., a second arithmetic unit). Since idle hardware is wasted hardware, processors also use aggressive heuristics for *speculative* execution (e.g., guessing the result of a future branch and filling the pipeline based on that assumption) and *out-of-order* execution (e.g., shuffling instructions and registers around at runtime to improve optimization).

(For more information on modern system architectures, consult one of the standard books in the area. Patterson and Hennessy [PH07] is considered the default textbook; Stokes [Sto07] provides a lighter introduction that focuses more directly on the machines you probably use.)

Privilege levels, syscall traps, and memory management all clearly assist in security. (Indeed, consider how susceptible a modern Internet-connected computer would be if it lacked kernel/user separation.) Sometimes, the lack of sufficient control can be frustrating. For example, if the MMU knew when a CPU's memory read was really for an instruction fetch, we could cook up a system in which memory regions had "read but do not execute" permission—thus providing a line of defense against stack-code injection attacks (recall Section 6.1). Indeed, this relatively straightforward idea was touted as the revolutionary *NX* feature by the technical press in recent years.

However, features such as internal caching and pipelining/out-of-order execution make things a bit harder. The relationship between what the internal system is doing and what an external device (such as a PCI card verifying integrity of kernel memory structures) can perceive is much less well defined. For example, suppose that we wanted to ensure that a certain FLASH device could be reprogrammed only when the system was executing a certain trusted module within ROM. Naively, we might add external hardware that sensed the address bus during instruction fetches and enabled FLASH changes only when those fetches were from ROM addresses.

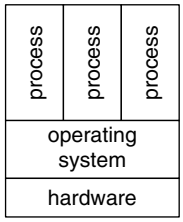

Figure 16.3 In the conventional system model, the OS provides protection between separate processes.

However, if we can't tell which cached instructions are actually being executed at the moment (let alone whether other code is simply "borrowing" parts of ROM as subroutines or even whether a memory read is looking for data or an instruction), then such techniques cannot work.

16.4.2 Virtualization

As Chapter 4 discussed, in the standard software architecture, an operating system provides services to user-level processes and enforces separation between these processes. As Section 16.4.1 discussed, hardware architecture usually reinforces these features. The aspiration here is that we don't want processes interfering with or spying on each other or on the OS, unless it's through a channel explicitly established and monitored by the OS. Figure 16.3 sketches this model.

In many scenarios, this approach to controlled separation may not be sufficient. Rather than providing separation between userland processes, one may prefer separation at the machine level. Rather than an OS protecting processes from each other, an OS and its processes are hoisted from a real machine up to a virtual machine, and another software layer—-usually called a *virtual machine monitor (VMM)*—protects these virtual machines from each other. Figure 16.4 shows one approach—although, as Section 16.5.1 discusses, many approaches exist here.

This idea of creating the illusion of multiple *virtual machines* within one machine is called *virtualization*. Initially explored in the early days of mainframe computing, virtualization has become fashionable again. What's the motivation for virtualization? Since this is a book about security, we tend to think about security first. And indeed, some reasons follow from security.

- For one thing, the API between the OS and userland applications can be extraordinarily rich and complicated. This complexity can make it hard to reason about and trust the properties one would like for this separation.

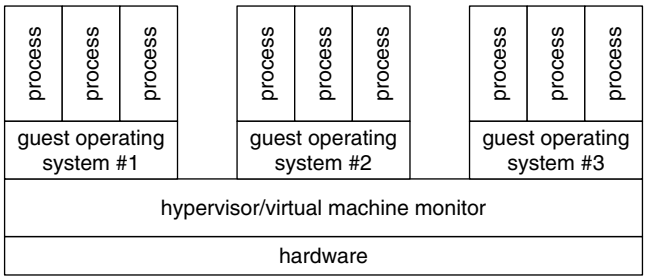

Figure 16.4 In virtualization models, processes are partitioned among virtual machines. Here, we sketch the type I approach, whereby the VMM runs on the hardware and provides separation between separate OS instances. In this case, the VMM is often called a *hypervisor*. Other models exist—see Section 16.5.1.

Complexity of API will also likely lead to complexity of implementation: increasing the size and likely untrustworthiness of the TCB.

- For another example, modern platforms have seen a continual bleeding of applications into the OS. As a consequence, untrustworthy code, such as device drivers and graphics routines, may execute with kernel privileges. (Indeed, some researchers blame this design choice for Windows' endless parade of vulnerabilities.)

Other reasons follow from basic economics.

- The API an OS gives to the application is highly specialized and may thus be unsuitable—for example, one can't easily run a mission-critical Win98 application as a userland process on OSX.
- According to rumor, this practice of giving each well-tested legacy application/OS its own machine leads to CPU utilization percentages in the single digits. Being able to put many on the same machine saves money.

Section 16.5.1 discusses old and new approaches to virtualization and security implications in more detail.

16.4.3 Multicore

Another trend in commercial CPUs is *multicore:* putting multiple processors (*cores*) on a single chip. The vendor motivation for this trend is a bit murky: increased performance is touted, better yield is rumored. However, multicore also raises the potential for security applications: If virtualization can help with a security idea,

then wouldn't giving each virtual machine its own processor be much simpler and more likely to work rather than mucking about with special modes?

One commercial multicore processor, CELL, touts security goals; it features an architecture in which userland processes get farmed off to their own cores, for increased protection from both other userland processes and the kernel. (We discuss this further in Section 16.5.3.)

16.4.4 Armored CPUs

Modern CPUs cache instructions and data internally and thus fetch code and data chunks at a time, instead of piecemeal. As we noted earlier, this behavior can make it hard for external security hardware to know exactly what the CPU is doing. However, this difficulty can be a feature as well as a bug, since it also can be hard for an external adversary to observe what's happening.

Consequently, if we assume that the adversary cannot penetrate the CPU itself, we might be able to achieve such things as private computation, by being sure that code lives encrypted externally; integrity of code and data, by doing cryptographic checks as the chunks move across the border; and binding data to code, by keeping data encrypted and decrypting it only internally if the right code came in.

Several research projects have built on this idea. XOM (Stanford) explored this idea to implement execute-only memory via simulators. AEGIS (MIT) made it all the way to real FPGA-based prototypes, which also incorporate the SPUF idea, to provide some grounds for the physical-security assumption.

16.4.5 Tagging

When lamenting the sad state of security in our current cyberinfrastructure, some security old-timers wistfully talk about *tagged* architectures, which had been explored in early research but had been largely abandoned. Rather than having all data items look alike, this approach *tags* each data item with special metadata. Implemented in hardware, this metadata gets stored along with the data in memory, gets transmitted along with it on buses—and controls the ways in which the data can be used. Systems in which permissions are based on capabilities (recall Chapter 4) might implement these capabilities as data items with a special tag indicating so; this keeps malicious processes from simply copying and forging capabilities.

Some of these ideas are finding expression again in modern research. For example, many buffer overflow attacks work because the adversary can enter data, as user input, which the program mistakenly uses as a pointer or address. Researchers

at the University of Illinois have built, via an FPGA (field-programmable gate array) prototype, a CPU retrofit with an additional metadata line to indicate that a data item is tainted [CXN+05]. The CPU automatically marks user input as tainted. Attempts to use a tagged data item as an address throw a hardware fault. However, certain comparison instructions—as code does when it sanity checks user input—clear the taint tags. Stanford's *TaintBochs* project uses software-based virtualization to explore further uses of taintedness and tagging in security contexts [CPG+04].

16.5 Coming Trends

So far, we have looked at the basic foundations of physical security and some of the ways it is embodied in tools available for system design. We close the chapter by looking at some new trends.

16.5.1 Virtualization and Security

Much security research—both old and new—is driven by a basic challenge. It can be useful to have separate compartments within a machine, with high assurance that malicious code in one compartment cannot spy on or disrupt the others. However, we often don't want *complete* separation between the compartments—but we want to make sure that only the right types of interaction occur. How do we provide this separation? How do we mediate the interaction? How do we provide assurance that this all works, that this all provides the desired properties, and that it doesn't kill performance?

In some sense, the challenge motivated the notion of separate processes within an operating system. However, most of the field has accepted the unfortunate notion that the standard OS model will not provide an appropriate solution here. This conclusion comes from several beliefs. Standard operating systems provide too rich and complex an interaction space between processes; standard operating systems are written too carelessly; target applications require more than simply the OS-level interface.

As we discussed in Section 16.4.2, these drawbacks have led to renewed thinking about virtualization: other ways to provide these separate virtual compartments and to mediate interaction between them. However, it's not clear what the "right" way to do this is. Right now, in both academia and industry, we see lots of approaches swirling around to enable machines to have highly compartmented pieces. This exploration raises many issues.

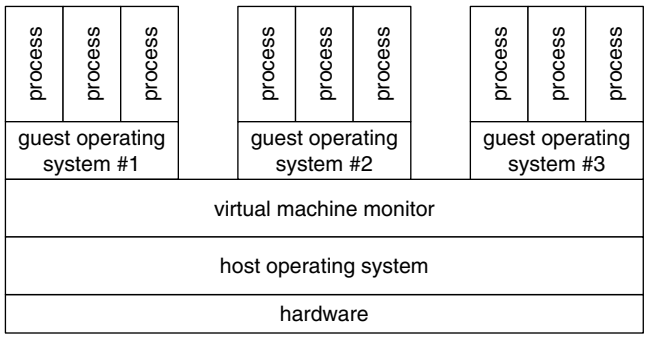

Figure 16.5 In the type II approach to virtualization, the VMM runs above the host operating system.

- What's the right level to split the machine into compartments?
- Should we use hardware-based support?
- Should we use a virtual machine monitor/hypervisor running above the hardware? (This is called a *type I* virtual machine—recall Figure 16.4.)
- Should we use a virtual machine monitor running within or above the host OS? (This is called a *type II* virtual machine—see Figure 16.5.)
- Should we instead virtualize some image above the OS? (Examples here include UML, BSD Jails, and Solaris Zones—see Figure 16.6.) We have seen some researchers call this approach *paenevirtualization*.
- Does the guest software—in particular, the OS—need to be rewritten in order to accommodate the virtualization, or it can be run unmodified? *Para-virtualization* refers to the former approach.

Some projects to watch in this space include *VMWare* [VMW07] and *XEN* [BDF+03, Xen07].

Another set of issues arise pertaining to the mediation between the compartments. How do we define the APIs? How do we have assurance that the APIs, if they work as advertised, work as intended? How do we have assurance that they are implemented correctly? Furthermore, an often neglected issue is how easily human designers and programmers can craft policies that capture the intended behavior. One might remember that these same issues vexed OS design—and one might cynically ask why virtualization research will do any better.

As we discussed earlier, one can take many approaches to providing this illusion. However, if the goal is to provide the illusion of the conventional architecture,

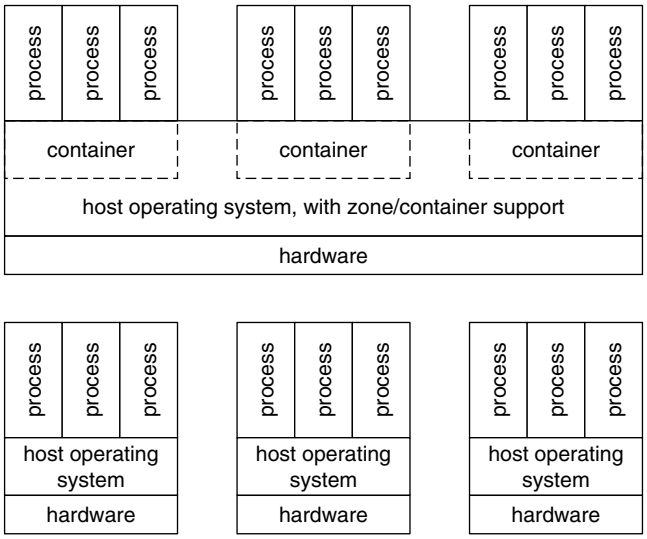

Figure 16.6 In yet another approach to virtualization, an enhanced OS groups processes into sets called *zones* or *containers* (above). One OS installation manages all the containers—however, from the perspective of the userland processes, each container appears to have its own instance of the OS and the machine (below).

doing so with conventional architecture is problematic. The guest system running inside a virtual machine expects to have user and kernel privileges. If guest kernel mode runs inside real user mode, then kernel-mode instructions won't necessarily operate properly. But if guest kernel mode runs inside real kernel mode, then nothing stops the guest from interfering with other virtual machines. (Indeed, the hacker community celebrates its *red pills*:[6] techniques to determine whether a program is running in a virtualized environment, via exploiting such differences. These properties were well known in the security literature, however.) For that matter, even if we could straighten out the privileges for virtual machines, how do we manage them all? Which code should do that, and how does the architecture enforce that? ([U+05] provides a nice overview of these challenges.)

Hardware vendors have been developing a new generation of processor architectures to address these issues. Intel's *Vanderpol technology (VT)*, sometimes defined as *virtualization technology*, removes the privilege and address space obstacles to virtualization. Intel's *LaGrande technology (LT)* adds support for secure management of virtualization; essentially, turning the kernel/user model into a quadrant:

6. Named after an instrument in the *Matrix* film series.

kernel/user for VM and kernel/user for hypervisor, which has special privileges. As of this writing, LT details are still limited to innuendo; according to rumor, the reason VT and LT seem to overlap somewhat in functionality is that they were rival, not complementary, projects. (Also, new marketing terms, such as TXT, appear to be supplanting these older project names.) AMD's *Pacifica* and *Presidio* architectures correspond to VT and LT, respectively.

Although not necessarily designed for security, these virtualization-friendly architectures have security applications. Such platforms as an IE/Windows Web-browsing installation, which typically are lost causes for security, can be safely confined in a VM. Watchdog modules that check system integrity no longer have to be constrained to looking at RAM and guessing at the internal state of the CPU; they can run inside the CPU, with their fingers inside the target VM. (*Virtual machine introspection* is a term used for this sort of thinking.) On the other hand, the *SubVirt* project illustrates another type of security application: malware that inserts itself as a malicious hypervisor (wags suggested calling it a "hypervirus") and shifts the victim system into a virtual machine, where it can't easily detect or counteract the malware [KCW+06].

16.5.2 Attestation and Authentication

We began our discussion of hardware-based security by considering how hardware can bind secrets to the correct computational entity. We see the potential for much industrial churn and ongoing research here.

One of the first issues to consider is which *a*-word applies: *attestation* or *authentication* or perhaps even *authorization*. When considering the general scenario of interacting with a remote party, the primary security question is: Who is it? Is it the party I think it is? Resolving this question is generally considered the domain of *authentication*. As discussed in Section 9.7.2, some dissidents instead see authentication as addressing the binding between entity and name and preach that *authorization*, as the binding of entity to property, is the true goal.

When the entity in question is a computational device, both identity and properties depend in part on the basic configuration of the device: what the hardware is and what the software is. However, as some of our own work demonstrated (e.g., [Smi04a]), the question is more subtle. The "correct" device can and perhaps should change software while remaining the same entity; the same hardware with a fresh reinstallation of the same software may in fact be a different entity. Consequently, consideration of the *a*-words in this realm often leads to an initial focus on *attesting* to a manifest of software and hardware configuration.

Some colleagues insist that, if checking software configuration is involved, then it must be attestation. We disagree.

- One cannot have meaningful attestation without authentication. An entity can claim any configuration it wants to; without authentication, the relying party has no reason to believe this claim.
- On the other hand, one can easily have authentication without attestation. When connecting to a hardware-hardened Web server, what the relying party cares about is the fact that it's a hardened Web server. The relying party does not necessarily need to know the full state of libraries and Apache versions—just that they're okay.

Whichever *a*-word one uses, we see two current sets of unresolved problems. The first is the "right" way to implement this structure within a multicompartmented machine. (We use the general term *multicompartmented* because we see this applying to a range of beasts, from SELinux boxes with TPMs to advanced virtualization and multicore.) Should a hardware-based root provide a full manifest for each compartment? Should a hardware-based root provide a manifest for a software-based root that in turn certifies each compartment? (And for that matter, why not other combinations of one or more hardware roots with one or more software roots?)

The second set of unresolved problems pertains to what should get listed in a manifest. What is it that the relying party really wants to know about a remote machine? We often joke that giving a TCG-style set of hashes is akin to the uniformed person at the door providing a DNA sample when asked to prove that he or she is really a bona fide police officer—it's a detailed answer that does not really give the right information. Current researchers are exploring *property-based attestation*, based on third parties' providing bindings, and *semantic remote attestation*, based on programming language semantics. This space will be interesting.

16.5.3 The Future of Moore's Law

In 1965, Gordon Moore observed that the number of tranistors on an integrated circuit doubles every 2 years. Subsequently blurred and transformed (e.g., the time-line is often presented as every 18 months), this curve has now entered popular folklore as *Moore's Law*,[7] usually stated as "every *N* years, the number of transistors on chips will double."

7. We have encountered a student who confused Moore's Law with Murphy's Law. The implications are worth considering.

So far, industry has stayed true to Moore's Law. However, insiders (e.g., [Col05]) observe that the causality is a bit more complicated than might meet the eye. Yes, Moore's Law was a good predictor of the trend of technology. But also, the industry came to use Moore's Law as a road map for its business model. For example, the generation N processor might currently be manufactured and the generation $N + 1$ almost ready to fab; however, the design for the generation $N + 2$ processor, to be fabbed k years later, was already under way and was counting on the fact that chip technology supporting far more transistors would be ready when the processor was ready to be manufactured.

Recently, hardware researchers have begun to express concern about the future of Moore's Law. Among many, the conventional wisdom is that, in order for Moore's Law to continue to hold, the transistors themselves will become less reliable—in terms of increased failure rate during manufacture and also, perhaps, in terms of increased failure rate in the field.

Some conjecture that the increased failure rate at manufacture will lead to a stronger emphasis on *multicore* devices. Committing to one large monolithic processor is risky, since faulty transistors might render the entire chip useless. An architecture that instead consisted of many smaller, somewhat independent modules is safer—the vendor can include a few extra modules in the chip, and sufficiently many should turn out to be good, even with faulty transistors.

However, we might also conjecture that an increased failure rate in the field might lead to a resurgance of work on Bellcore attacks and countermeasures (recall the discussion in Section 16.2.1).

16.5.4 Personal Tokens of the Future

The personal tokens common in the past decade were *smart cards*: credit-card-sized pieces of plastic with small chips on them, typically used in small-value commercial transactions. As we observed in Section 16.3.4, USB devices are common now. What's coming next?

Personal digital assistants (PDAs) are one possible candidate. For designers of security protocols, PDAs offer the advantage of having an I/O channel the user trusts, thus avoiding some of the problems of traditional smart cards. However, one might be cynical as well. As PDAs become more like general-purpose computing environments, the greater their risk of contamination—and the less advantage they offer over a risky general-purpose platform. Some economic observers predict that *cell phones* will displace PDAs. For the security designer, cell phones offer the challenge that it can be harder to experiment and deploy new applications; vendors

tend to keep things locked up. (Looking at the students and young professionals who surround us, we might wonder whether iPods might be usable as a personal token.)

16.5.5 RFID

The burgeoning use of *RFID (radio frequency identification)* devices also offers potential for security applications and abuses. Of course, the first step in this discussion is to nail down exactly what RFID devices are. Everyone agrees that these are electronic devices that use some type of close-range radio to communicate. However, the sophistication assigned to these devices varies, from simple replacements for optical barcodes to more complex devices that are armed with environmental sensors, state, and batteries and that participate in interactive protocols with "reader" devices.

Anyone who has tried to wrestle groceries into the right position for a laser to read the barcode printed on them can immediately appreciate the advantages of inexpensive RFID tags that can be read from any orientation, without line-of-sight. Indeed, discussions of application scenarios often begin on such use cases: replacing clumsy optically read tags with easy and efficient RF-read ones, on items such as groceries, library books, warehouse inventory, and passports. No need to manually wrestle the item into the right position—the RF makes the connection automatically!

Of course, the same ease of use that motivates the application of RFID technology is also the source of its security and privacy worries. A machine-readable barcode is typically big enough for a human to see as well—so humans can be aware of its presence. The physical manipulation required for a barcode to be scanned is typically big enough for a human to notice—so humans can make judgments about what's being scanned and when. Humans also understand the notion of "sight" and thus have a good intuition of how to keep an optical tag from being seen.

These artifacts, which made it possible for human end users to control and understand the use of optical identifiers, disappear with RFID. Which objects have tags? Who is reading them and when and from how far away? What are the privacy implications of this quantum leap in automated information gathering?

Of course, the general notion of an inexpensive device communicating over an open medium raises the more standard security questions of physical security of the end device and communications security between them.

Juels's survey [Jue06] and the Garfinkel-Rosenberg anthology [GR05] provide more discussion of this problem space. Recently, NIST even published guidelines for RFID security [KEB+07].

16.6 The Take-Home Message

We often think of our system as the software we've written. However, a complete view of "the system" includes the hardware that executes the instructions we've written. As we've been discussing throughout this chapter, the set of hardware components that we rely on to run our applications can make or break the security of the system. Hardware can aid us in building more secure and resilient systems; it can also make that job much more difficult.

Even if you never design or build a hardware component, understanding the features and limitations of hardware will help you design better systems. Where can we store secrets? Does the hardware protect computation and data? What types of adversaries are they protected from? These types of questions should be part of the standard checklist when it comes to building secure systems; they are certainly part of any good attacker's.

There's something strangely Gödellian in thinking that we can make software more secure by simply writing more, perhaps better, software. If we end up with either incompleteness or inconsistency, there's a good chance that some security trouble is lurking just around the corner. If designed and used correctly, hardware might be able to help. As with anything else, it's no magic bullet. It's a tool that, when applied appropriately, can solve certain issues.

16.7 Project Ideas

1. In Section 16.1.1, we noted that a colleague insists that it's impossible to read data from disks once the cells themselves have been overwritten. Nonetheless, rumors persist (e.g., "just hack the disk controller code to change the head alignment so it reads the edges of the tracks instead!"). Prove our colleague wrong!
2. In Section 16.2.1, we discussed how an attack that forces the CPU to take one direction of the branch—whether or not it's correct—could let the adversary subvert the correctness of the system. Can you find some real code examples of this?
3. Learn a bit about how dongles are used to protect software. Can you think of ways to break them? Can you design a better scheme?
4. Implement modular exponentiation for 1024-bit integers on a 32-bit machine. Time the result of your software-only implementation, and compare it to numbers given by your favorite cryptographic accelerator's hardware implementation.

5. Assume that we had a TPM whose PCRs used MD5 as a hash algorithm instead of SHA1. Knowing that MD5 has some problems (see Chapter 8), think about ways that you can exploit MD5 weaknesses to hack a TPM.
6. Sketch a design for new CPU interfaces that would make it easier to determine what code was being executed in what context. (Extra credit: Prototype your design with OpenSPARC.)
7. One popular use of virtualization is in Linux honeypots that rely on *user-mode linux (UML)*. Design (and code, for extra credit) a red pill to determine whether your program is running on UML. How does "real" virtualization (i.e., under the OS) improve the situation?