Dyad: A System for Using Physically Secure Coprocessors

J. D. Tygar   Bennet Yee

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Partial support was provided by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. Additional support was provided in part under a Presidential Young Investigator Award, Contract No. CCR-8858087 and matching funds from Motorola Inc. and TRW. Additional partial support was provided by a contract from the U.S. Postal Service. We gratefully acknowledge the generous support of IBM through equipment grants and loans.

Abstract

Physically secure coprocessors, as used in the Dyad project at Carnegie Mellon University, provide easily implementable solutions to perplexing security problems. This paper presents the solutions to five problems: (1) protecting the integrity of publicly accessible workstations; (2) tamper-proof accounting/audit trails; (3) copy protection; (4) electronic currency without centralized servers; and (5) electronic contracts.

Introduction and Motivation

The Dyad project at Carnegie Mellon University is using physically secure coprocessors to achieve new protocols and systems addressing a number of perplexing security problems. These coprocessors can be produced as boards or integrated circuit chips and can be directly inserted in standard workstations or PC-style computers. This paper presents a set of security problems and easily implementable solutions that exploit the power of physically secure coprocessors.

Standard textbook treatments of computer security assert that physical security is a necessary precondition to achieving overall system security. While meeting this condition may seem reasonable for yesterday's computer centers with their large mainframes, it is no longer so easy today. Most modern computer facilities consist of workstations within offices or of personal computers arranged in public access clusters, all of which are networked to file servers. In situations such as these where computation is distributed, physical security is very difficult — if not impossible — to realize. Neither computer clusters, nor offices, nor networks are secure against intruders. An even more difficult problem is posed by a user who may wish to subvert his own machine; for example, a user who wishes to steal a copy protected program can do so by gaining read access to the system memory while the system runs the nominally "execute only" program. By making the processing power of workstations widely and easily available, we've also made the system hardware accessible to casual interlopers. How do we remedy this?

Researchers have recognized the vulnerability of network wires and have brought the tools from cryptography to bear on the problem of non-secure communication networks, and this has led to a variety of key exchange and authentication protocols [15, 16, 20, 35, 37, 45, 46, 53, 54] for use with end-to-end encryption to provide privacy on network communications. Others have noted the vulnerability of workstations and their disk storage to physical attacks in the office workstation environment, and this has led to a variety of *secret sharing* algorithms for protecting data from isolated attacks [24, 42, 48]. Tools from the field of consensus protocols can also be applied [24]. These techniques, while powerful, still depend on some measure of physical security.

Cryptography allows us to slightly relax our assumptions about physical security; with cryptography we no longer need to assume that our network is physically shielded. However, we still need to make strong assumptions about the physical protection of hosts. We cannot entirely eliminate the need for physical security.

All security algorithms and protocols depend on physical security. Cryptographic systems depend on the secrecy of keys, and authorization and access control mechanisms crucially depend on the integrity of the access control database. The use of physical security to provide privacy and integrity is the foundation upon which security mechanisms are built. With the proliferation of workstations to the office and to open computation clusters, the physical security assumption is no longer valid. The recent advent of powerful mobile computers only exacerbates this problem, since the machines may easily be physically removed.

The gap between the reality of physically unprotected systems and this assumption of physical security must be closed. With traditional mainframe systems, the security firewall was between the users' terminals and the computer itself — the mainframe was the physically secure component in the system.[1] With loosely administered, physically accessible workstations, the security partition can no longer encompass all the machines. Indeed, with most commercially available workstations, the best that can be found is a simple lock in the front panel which can be easily picked or bypassed — there really is no physically secure component in these systems.

This paper discusses the use of physically secure processors to achieve new, powerful solutions to system security problems. (Physically secure coprocessors were first introduced in [6].) A secure coprocessor embodies a physically secure hardware module; it achieves this security by advanced packaging technology [62]. We focus on systems and protocols that can exploit the physical shielding to achieve novel solutions to challenging problems. There are many applications that need to use secure coprocessors; we discuss five of them here.

1. Consider the problem of protecting the integrity of publicly accessible workstations. For normal workstations or PCs, it is very easy to steal or modify data and programs on the hard disks. Operating system software could be modified to log keystrokes to extract encryption keys that you may have used to protect data. There is neither privacy nor integrity when the attacker has physical access to the machine, even if we prevent the attacker from adding Trojan horses to hardware (e.g., a modified keyboard which records keystrokes or a network interface board which sends the contents of the system memory to the attacker).

2. The problem of providing tamper-proof audit trails and accounting logs is

[1] Even greater security would be achieved if the terminals were also secure; otherwise the users would have the right to wonder whether their every keystroke is being spied upon.

similar to that of workstation integrity, except that instead of protecting largely static data (operating system kernels and system programs), the goal is to make the generated logs unforgeable. For normal workstations or PCs, nothing prevents attackers from modifying system logs to erase evidence of intrusion. Similarly, secure system accounting is impossible because nothing protects the integrity of the accounting logs.

3. The problem of providing copy protection for proprietary programs is also insoluble on traditional hardware. Distributing software in encrypted form does not help, since to run it the user's machine must have the software decrypted in its memory. Because we cannot guarantee the integrity of the machine's operating system, we have no assurances as to the privacy of this in-memory copy of the software.

4. Another difficult problem is that of providing electronic currency without centralized control. Any electronic representation of currency is subject to duplication — data stored in computers can always be copied, regardless of how our software may chose to interpret them. When electronic currency no longer remains on trusted, centralized server machines, there is no way to guarantee against tampering.

   Given that we cannot trust the system software on our publicly accessible computers, any electronic currency on our machines might be arbitrarily created, destroyed, or sent over a network to the attacker. Alternatively, an untrustworthy user can record the state of his computer prior to "spending" his electronic currency, after which he simply resets the state of his computer to the saved state. Without a way to securely manage currency, attackers may "print" money at will. Furthermore, the attacker may take advantage of a partitioned network in order to use the same electronic currency in transactions with machines in different partitions. Since no communication is possible between these machines, users (or computers acting as service-providers) have no way to check for duplicity.

5. A closely related problem arises when we want to provide "electronic contracts" that obligate secure coprocessors to perform certain actions or enforce certain restrictions. The notion of an electronic contract provides a mechanism for controlling the configuration of security constraints listed in the above items. Note that it does not suffice to merely cryptographically sign

contracts; we must ensure that contracts are enforced on all machines that are parties in the contract.

All of these problems are vulnerable to physical attacks which result in a loss of privacy and integrity. Software protection systems crucially rely on the physical security of the underlying hardware and are completely useless when the physical security assumption is violated.

We can, however, close the assumption/reality gap in computer security. By adding physically secure coprocessors to computer systems, real, practical security systems can be built. Not only are secure coprocessors necessary and sufficient for security systems to be built; placing the security partition around a coprocessor is the natural model for providing security for workstations. Moreover, they are cost effective and can be made largely transparent to the end user.

The rest of this paper outlines the theory of secure coprocessors. First we discuss a model for physically secure coprocessors and describes a number of platforms that use secure coprocessor technology. Then we consider several important security problems that are solved by using secure coprocessors. We next present a hierarchy of traditional and new approaches to physical security, and demonstrates naturally induced security partitions within these systems. We give an approach which allows secure coprocessors to be integrated into existing operating systems; we continue with a machine-user authentication section, which tackles the problem of verifying the presence of a secure coprocessor to users. Finally, we discuss previous work.

Secure Coprocessors

What do we mean by the term *secure coprocessor*? A secure coprocessor is a hardware module containing (1) a CPU, (2) ROM, and (3) NVM (non-volatile memory). This hardware module is physically shielded from penetration, and the I/O interface to this module is the only means by which access to the internal state of the module can be achieved. (Examples of packaging technology are discussed later in this section.) Such a hardware module can store cryptographic keys without risk of release. More generally, the CPU can perform arbitrary computations (under control of the operating system) and thus the hardware module, when added to a computer, becomes a true coprocessor. Often, the secure coprocessor will contain special-purpose hardware in addition to the CPU and memory; for example, high speed encryption/decryption hardware may be used.

Secure coprocessors must be packaged so that physical attempts to gain access to the internal state of the coprocessor will result in resetting the state of the secure coprocessor (i.e., erasure of the NVM contents and CPU registers). An intruder might be able to break into a secure coprocessor and see how it is constructed; the intruder cannot, however, learn or change the internal state of the secure coprocessor except through normal I/O channels or by forcibly resetting the entire secure coprocessor. The guarantees about the privacy and integrity of non-volatile memory provide the foundations needed to build security systems.

Physical Assumptions for Security

All security systems rely on a nucleus of assumptions. For example, it is often assumed that it is infeasible to successfully cryptoanalyze the encryption system used for security. Our basic assumption is that the coprocessor provides private and tamper-proof memory and processing. Just as attackers can exhaustively search cryptographic key spaces, it may be possible to falsify the physical security hypothesis by expending enormous resources (possibly feasible for very large corporations or government agencies), but we will assume the physical security of the system as an axiom. This is a physical work-factor argument, similar in spirit to intractability assumptions of cryptography. Our secure coprocessor model does not depend on the particular technology used to satisfy the work-factor assumption. Just as cryptographic schemes may be scaled to increase the resources required to penetrate a cryptographic system, current security packaging techniques may be scaled or different packaging techniques may be employed to increase the work-factor necessary to successfully bypass the physical security measures.

In the section on applications, we will see examples of how we can build secure subsystems running partially on a secure coprocessor by leveraging off the physical security of the coprocessor.

Limitations of Model

Even though confining all computation within secure coprocessors would ideally suit our security needs, in reality we cannot – and should not – convert all of our processors into secure coprocessors. There are two main reasons: the first is the inherent limitations of the physical security techniques in packaging circuits, and the second is the need to keep the system maintainable. Fortunately, as we shall see in the section below, we do not need the entire computer need not be physically shielded. It suffices to physically protect only a portion of the computer.

Current packaging technology limits us to approximately one printed circuit board in size to allow for heat dissipation. Future developments may eventually relax this and allow us to make more of the solid-state components of a multiprocessor workstation physically secure, perhaps an entire card cage; the security problems of external mass storage and networks, however, will in all likelihood remain a constant.

While it may be possible to securely package an entire multiprocessor in a physically secure manner, it is likely to be impractical and is unnecessary besides. If we can obtain similar functionalities by placing the security concerns within a single coprocessor, we can avoid the cost of making all the processors (in multiprocessors) secure.

Making a system easy to maintain requires a modular design. Once a hardware module is encapsulated in a physically secure package, disassembling the module to fix or replace some component will probably be impossible. Moreover, packaging considerations, as well as the extra hardware development time required, implies that the technology used within a secure coprocessor may lag slightly behind the technology used within the host system – perhaps by one generation. The right balance between physically shielded and unshielded components will depend on the class of applications for which the system is intended. For many applications, only a small portion of the system must be protected.

Potential Platforms

Several real instances of physically secure processing exist. This subsection describes some of these platforms, giving the types of attacks which these systems are prepared against, and the limitations placed on the system due to the approaches taken to protect against physical intrusion.

The μABYSS [62] and Citadel [64] systems provide physical security by employing board-level protection. The systems include an off-the-shelf microprocessor and some non-volatile (battery backed) memory, as well as special sensing circuitry which detects intrusion into a protective casing around the circuit board. The security circuitry erases the non-volatile memory before attackers can penetrate far enough to disable the sensors or to read the memory contents from the memory chips. The Citadel system expands on μABYSS, incorporating substantially greater processing power; the physical security mechanisms remain identical.

Physical security mechanisms must protect against many types of physical attacks. In the μABYSS and Citadel systems, it is assumed that in order for intruders to penetrate the system, they must be able to probe through a hole of one millimeter in diameter (probe pin voltages, destroy sensing circuitry, etc). To prevent direct intrusion, these systems incorporate sensors consisting of fine (40 gauge) nichrome wire, very low power sensing circuits, and a long life-time battery. The wires are loosely but densely wrapped in many layers about the circuit board and the entire assembly is then dipped in a potting material. The loose and dense wrapping makes the exact position of the wires in the epoxy unpredictable. The sensing electronics can detect open circuits or short circuits in the wires and erase the non-volatile memory if intrusion is attempted. The designs show that physical intrusion by mechanical means (e.g., drilling) cannot penetrate the epoxy without breaking one of these wires.

Another physical attack is the use of solvents to dissolve the potting material to expose the sensor wires. To block this kind of attack, the potting material is designed to be chemically “stronger” than the sensor wires. This implies that solvents will destroy at least one of the wires – and thus create an open-circuit condition – before the intruder can bypass the potting material and access the circuit board.

Yet another physical attack uses low temperatures. Semiconductor memories retain state at very low temperatures even without power, so an attacker could freeze the secure coprocessor to disable the battery and then extract the memory contents at leisure. The designers have blocked this attack by the addition of temperature sensors which trigger erasure of secrets before the low temperature reaches the dangerous level. (The system must have enough thermal mass to prevent quick freezing – by being dipped into liquid nitrogen or helium, for example – so this places some limitations on the minimum size of the system.)

The next step in sophistication is the high-powered laser attack. Here, the idea is that a high powered (ultraviolet) laser may be able to cut through the protective potting material and selectively cut a run on the circuit board or destroy the battery before the sensing circuitry has time to react. To protect against such an attack, alumina or silica is added to the epoxy potting material which causes it to absorb ultraviolet light. The generated heat will cause mechanical stress, which will cause one or more of the sensing wires to break.

Instead of the board-level approach, physical security can be provided for smaller, chip-level packages. Clipper and Capstone, the NSA’s proposed DES replacements [3, 56, 57] are special purpose encryption chips. The integrated circuit chips are designed in such a way that key information (and perhaps other important encryption parameters – the encryption algorithm is supposed to be secret as well) are destroyed when attempts are made to open the integrated circuit chips' packaging. The types of attacks which this system can withstand are unknown.

Another approach to physically secure processing appears in *smart-cards* [30]. A smart-card is essentially a credit-card-size microcomputer which can be carried in a wallet. While the processor is limited by size constraints and thus is not as powerful as that found in board-level systems, no special sensing circuitry is necessary since physical security is maintained by the virtue of its portability. Users may carry their smart-cards with them at all times and can provide the necessary physical security. Authentication techniques for smart-cards have been widely studied [1, 30].

These platforms and their implementation parameters together provide the technology envelope within which secure coprocessor hardware will likely reside and this envelope will provide constraints on what class of algorithms is reasonable. As more computation power moves into mobile computers and smart-cards and better physical protection mechanisms are devised, this envelope will grow larger with time.

Applications

Because secure coprocessors can *process* secrets as well as store them, they can do much more than just keep secrets confidential. We can use the ability to compute privately to provide many security related features, including (1) host integrity verification; (2) tamper proof audit trails; (3) copy protection; (4) electronic currency; and (5) electronic contracts. [2] None of these are realistically possible on physically exposed machines.

[2] In recent work [55], we have also demonstrated the feasibility of cryptographically protecting postage franking marks printed possibly by PC-based electronic postage meters. Because such meters are located at customer sites, secure coprocessors were crucial for the protection of cryptographic keys.

Host Integrity Check

Trojan horse software dates back to the 1960s, if not earlier. Bogus login programs are the most common, though games and fake utilities are also widely used to set up back doors as well. Computer viruses exacerbate the problem of host integrity — the system may easily be inadvertently corrupted during normal use.

The host integrity problem can be partially ameliorated by guaranteeing that all programs have been inspected and approved by a trusted authority, but this is at best an incomplete solution. With computers getting smaller and workstations often physically accessible in public computer clusters, attackers can easily bypass any logical safeguards to modify the disks. How can you tell if even the operating system kernel is correct? The integrity of the computer needs to be verified. The integrity of the kernel image and system utilities stored on disk must be verified to be unaltered since the last system release.[3]

There are two main cases to examine. The first is that of stand-alone workstations that are not connected to any networks, and the second is that of networked workstations with access to distributed services such as AFS [52] or Athena [4]. While publicly accessible stand-alone workstations have fewer avenues of attack, there are also fewer options for countering attacks. We will examine both cases concurrently in the following discussion.

Using a secure coprocessor to perform the necessary integrity checks solves the host integrity problem. Because of the privacy and integrity guarantees on secure coprocessor memory and processing, we can use a secure coprocessor to check the integrity of the host's state at boot-up and have confidence in the results. At boot time, the secure coprocessor is the first to gain control of the system and can decide whether to allow the host CPU to continue by first checking the disk-resident bootstrap program, operating system kernel, and all system utilities for evidence of tampering.

The cryptographic checksums of system images must be stored in the secure

[3] Sufficiently sophisticated hardware emulation can fool both users and any integrity checks. If an attacker replaced a disk controller with one which would provide the expected data during system integrity verification but would return Trojan horse data (system programs) for execution, there would be no completely reliable way to detect this. Similarly, it would be very difficult to detect if the CPU were substituted with one which fails to correctly run specific pieces of code in the operating system protection system. One limited defense against hardware modifications is to have the secure coprocessor do behavior and timing checks at random intervals. There is no absolute defense against this form of attack, however, and the best that we can do is to make such emulation difficult and force the hardware hackers to build more perfect Trojan horse hardware.

coprocessor's NVM and protected both against modification and (depending on the cryptographic checksum algorithm chosen) against exposure. Of course, tables of cryptographic checksums can be paged out to host memory or disk after first checksumming and encrypting them within the secure coprocessor; this can be handled as an extension to normal virtual memory paging. We have more to say on this subject in the section on system architecture. Since the integrity of the cryptographic checksums is guaranteed by the secure coprocessor, we can detect any modifications to the system objects and protect ourselves against attacks on the external storage.

One alternative model is to eliminate external storage for networked workstations — to use trusted file servers and access a remote, distributed file system for all external storage. Any paging needed to implement virtual memory would go across the network to a trusted server with disk storage.

What are the difficulties with this trusted file server model? First, note that non-publicly readable files and virtual memory pages must be encrypted before being transferred over the network and so some hardware support is probably required anyway for performance reasons. A more serious problem is that the workstations must be able to authenticate the identity of the trusted file servers (the host-to-host authentication problem). Since workstations cannot keep secrets, we cannot use shared secrets to encrypt and authenticate data between the workstation and the file servers. The best that we can do is to have the file servers use public key cryptography to cryptographically sign the kernel image when we boot over the network, but we must be able to store the public keys of the trusted file servers somewhere. With exposed workstations, there's no safe place to store them. Attackers can always modify the public keys (and network addresses) of the file servers so that the workstation would contact a false server. Obtaining public keys from some external key server only pushes the problem one level deeper — the workstation would need to authenticate the identity of the key server, and attackers need only to modify the stored public key of the key server.

If we page virtual memory over the network (which we assume is not secure), the problem only becomes worse. Nothing guarantees the privacy or integrity of the virtual memory as it is transferred over the network. If the data is transferred in plaintext, an attacker can simply record network packets to break privacy and modify/substitute network traffic to destroy integrity. Without the ability to keep secrets, encryption is useless for protecting their memory — attackers can obtain the encryption keys by physical means and destroy privacy and integrity as before.

A second alternative model, which is a partial solution to the host integrity problem, is to use a secure-boot floppy containing system integrity verification code to bring machines up. Let's look at the assumptions involved here. First, note that we are assuming that the host hardware has not been compromised. If the host hardware has been compromised, the "secure" boot floppy can easily be ignored or even modified when used, whereas secure coprocessors cannot. The model of using a secure removable medium for booting assumes that untrusted users get a (new) copy of a master boot floppy from the trusted operators each time a machine is rebooted from an unknown state. Users must not have access to the master boot floppy since it must not be altered.

What problems are there? Boot floppies cannot keep secrets — encryption does not help, since the workstation must be able to decrypt them and workstations cannot keep secrets (encryption keys) either. The only way to assure integrity without completely reloading the system software is to check it by checking some kind of cryptographic checksum on the system images.

There are a variety of cryptographic checksum functions available, and all obviously require that the integrity of the checksums for the "correct" data be maintained: when we check the system images on the disk of a suspect workstation, we must recompute new checksums and compare them with the original ones. This is essentially the same procedure used by secure coprocessors, except that instead of providing integrity within a piece of secure hardware we use trusted operators. The problem then becomes that of making sure that operators and users follow the proper security procedures. Requiring that users obtain a fresh copy of the integrity check software and data each time they need to reboot a machine is cumbersome. Furthermore, requiring a centralized database of all the software that requires integrity checks for all versions of that software on the various machines will be a management nightmare. Any centralized database is necessarily a central point of attack. Destroying this database will deny service to anybody who wishes to securely bootstrap their machine.

Both secure coprocessors and secure boot floppies can be fooled by a sufficiently faithful emulation of the system which simulates a normal disk during integrity checks, but secure coprocessors allow us to employ more powerful integrity check techniques to provide better security. Furthermore, careless use (i.e., reuse) of boot floppies becomes another channel of attack — boot floppies can easily be made into viral vectors.

Along with integrity, secure coprocessors offer privacy; this property allows the use of a wider class of cryptographic checksum functions. There are many cryptographic checksum functions that might be used, including Rivest's MD5

[44], Merkle's Snefru [31], Jueneman's Message Authentication Code (MAC) code [26], IBM's Manipulation Detection Code (MDC) [25], chained DES [60], and Karp and Rabin's family of fingerprint functions [28]. All of these require integrity; the last three require privacy of keys. The strength of these rely on the difficulty of finding *collisions* — two different inputs with the same checksum. The intractability arguments for the first four of these are based on conjectured numbers of bit operations required to find collisions, and so are weak with respect to theoretical foundations. MDC, chained DES, and the fingerprint functions also keep the identity of the particular checksum function used secret — with MDC and DES it corresponds to keeping encryption keys (which select particular encryption functions) secret, and with fingerprint functions it corresponds to keeping an irreducible polynomial (which defines the fingerprint function) secret. DES is less well understood than the Karp-Rabin functions.

The secrecy requirement of MDC, chained DES, and the Karp-Rabin functions is a stronger assumption which *can* be provided by a secure coprocessor and it allows us to use cryptographic functions with better theoretical underpinnings, thus improving the bounds on the security provided. Secrecy, however, cannot be provided by a boot floppy. The Karp-Rabin fingerprint functions are superior to chained DES in that they are much faster and much easier to implement (thus the implementation is less likely to contain bugs), and there are no proven strong lower bounds on the difficulty of breaking DES.

Secure coprocessors also greatly simplify the problem of system upgrades. This is especially important when there are large numbers of machines on a network: systems can be securely upgraded remotely through the network. Furthermore, it's easy to keep the system images encrypted while they are being transferred over the network and while they are resident on secondary storage. This provides us with the ability to keep proprietary code protected against most attacks. As noted below in the section on copy protection, we can run (portions of) the proprietary software only within the secure coprocessor, allowing vendors to have execute-only semantics — proprietary software need never appear in plaintext outside of a secure coprocessor.

The later section on operational requirements discusses the details of host integrity check as it relates to secure coprocessor architectural requirements, and the section on key management discusses how system upgrades would be handled by a secure coprocessor. Also relevant is the problem of how the user can know if a secure coprocessor is running properly in a system; our section on machine-user authentication discusses this.

Audit Trails

In order to properly perform system accounting and to provide data for tracing and detecting intruders on the host system, audit trails must be kept in a secure manner. First, note that the integrity of the auditing and accounting logs cannot be completely guaranteed (since the entire physically accessible machine, including the secure coprocessor, may be destroyed). The logs, however, can be made tamper evident. This is quite important for detecting intrusions — forging system logs to eliminate evidence of penetration is one of the first things that a system cracker will attempt to do. The privacy and integrity of the system accounting logs and audit trails can be guaranteed (unless the secure coprocessor is removed or destroyed) simply by holding them inside the secure coprocessor. It is awkward to have to keep everything inside the secure coprocessor since accounting and audit logs can grow very large and resources within the secure coprocessor are likely to be tight. Fortunately, it is also unnecessary.

To provide secure logging, we use the secure coprocessor to seal the data against tampering with one of the cryptographic checksum functions discussed above; we then write the logging information out to the file system. The sealing operation must be performed within the secure coprocessor, since all keys used in this operation must be kept secret. By later verifying these cryptographic checksums we make tampering of log data evident, since the probability that an attacker can forge logging data to match the old data's checksums is astronomically low. This technique reduces the secure coprocessor storage requirement from large logs to the memory sufficient to store the cryptographic keys and checksums, typically several words per page of logged memory. If the space requirement for the keys and checksums is still too large, they can be similarly written out to secondary storage after being encrypted and checksummed by master keys.

Additional cryptographic techniques can be used for the cryptographic sealing, depending on the system requirements. Cryptographic checksums can provide the basic tamper detection and are sufficient if we are only concerned about the integrity of the logs. If the accounting and auditing logs may contain sensitive information, privacy can be provided by using encryption. If redundancy is required, techniques such as secure quorum consensus [24] and secret sharing [48] may be used to distribute the data over the network to several machines without greatly expanding the space requirements.

Copy Protection

A common way of charging for software is licensing the software on a per-CPU, per-site, or per-use basis. Software licenses usually prohibit making copies for use on unlicensed machines. Without a secure coprocessor, this injunction against copying is unenforceable. If the user can execute the code on any physically accessible workstation, the user can also read that code. Even if we assume that attackers cannot read the workstation memory while it is running, we are implicitly assuming that the workstation was booted correctly — verifying this property, as discussed above, requires the use of a secure coprocessor.

When secure coprocessors are added to a system, however, we can quite easily protect executables from being copied and illegally utilized by attackers. The proprietary code to be protected — or at least some critical portion of it — can be distributed and stored in encrypted form, so copying it without obtaining the code decryption key is futile.[4] Public key cryptography may be used to encrypt the entire software package or a key for use with a private key system such as DES. When a user pays for the use of a program, a digitally signed certificate of the public key used by his secure coprocessor is sent to the software vendor. This certificate is signed by a key management center verifying that a given public key corresponds to a secure coprocessor, and is prima facie evidence that the public key is valid. The corresponding private key is stored only within the NVM of the secure coprocessor; thus, only the secure coprocessor will have full access to the proprietary software.

What if the code size is larger than the memory capacity of the secure coprocessor? We have two alternatives: we can use *crypto-paging* or we can split the code into protected and unprotected segments.

We discuss crypto-paging in greater detail in the section on system architecture below, but the basic idea is to dynamically load the relevant sections of memory from the disk as needed. Since good encryption chips are fast, we can decrypt on the fly with little performance penalty. Similarly, when we run out of memory space on the coprocessor, we encrypt the data as we flush it out onto secondary storage.

Splitting the code is an alternative to this approach. We can divide the code

[4] Allowing the encrypted form of the code to be copied means that we can back up the workstation against disk failures. Even giving attackers access to the backup tapes will not release any of the proprietary code. Note that our encryption function should be resistant to known-plaintext attacks, since executable binaries typically have standardized formats.

into a security critical section and an unprotected section. The security-critical section is encrypted and runs only on the secure coprocessor. The unprotected section runs in parallel on the main host processor. An adversary can copy the unprotected section, but if the division is done well, he or she will not be able to run the code without the secure portion.

A more primitive version of the copy protection application for secure coprocessors originally appeared in [63].

Electronic Currency

With the ability to keep licensed proprietary software encrypted and allow execute access only, a natural application would be to allow for charging on a pay-per-use basis. In addition to controlling access to the software according to the terms of software licenses, some mechanism must be available to perform cost accounting, whether it is just keeping track of the number of times a program has run or keeping track of dollars in the users' account. More generally, this accounting software provides an *electronic currency* abstraction. Correctly implementing electronic currency requires that account data be protected against tampering — if we cannot guarantee integrity, attackers will be able to create electronic money at will. Privacy, while perhaps less important here, is a property that users expect for their bank balance and wallet contents; similarly, electronic money account balances should also be private.

There are several models that can be adopted for handling electronic funds. The first is the cash analogy. Electronic funds can have similar properties to cash: (1) exchanges of cash can be effectively anonymous; (2) cash cannot be created or destroyed; and (3) cash exchanges require no central authority. (Note that these properties are actually stronger than that provided by currency — serial numbers can be recorded to trace transactions, and the national treasuries regularly prints and destroys money.)

The second model is the credit cards/checks analogy. Electronic funds are not transferred directly; rather, promises of payment — perhaps cryptographically signed to prove authenticity — are transferred instead. A straightforward implementation of the credit card model fails to exhibit any of the three properties above. By applying cryptographic techniques, anonymity can be achieved [10], but the latter two requirements remain insurmountable. Checks must be signed and validated at central authorities (banks), and checks/credit payments en route "create" temporary money. Furthermore, the potential for reuse of cryptographic signed checks requires that the payee must be able to validate the check with the central authority prior to committing to a transaction.

The third model is analogous to a rendezvous at the bank. This model uses a centralized authority to authenticate all transactions and so is even worse for large distributed applications. The bank is the sole arbiter of the account balance and can easily implement the access controls needed to ensure privacy and integrity of the data. This is essentially the model used in Electronic Funds Transfer (EFT) services provided by many banks — there are no access restrictions on deposits into accounts, so only the depositor for the source account needs to be authenticated.

Let us examine these models one by one. What sort of properties must electronic cash have? We must be able to easily transfer money from one account to another. Electronic money must not be created or destroyed by any but a very few trusted users who regulate the electronic version of the Treasury.

With electronic currency, integrity of the accounts data is crucial. We can establish a secure communication channel between two secure coprocessors by using a key exchange cryptographic protocol and thus maintain privacy when transferring funds. To ensure that electronic money is conserved (neither created nor destroyed), the transfer of funds should be failure atomic, i.e., the transaction must terminate in such a way as to either fail completely or fully succeed — transfer transactions cannot terminate with the source balance decremented without having incremented the destination balance or vice versa. By running a transaction protocol such as two-phase commit [8, 12, 65] on top of the secure channel, the secure coprocessors can transfer electronic funds from one account to another in a safe manner, providing privacy as well as ensuring that money is conserved throughout. With most transaction protocols, some "stable storage" for transaction logging is needed to enable the system to be restored to the state prior to the transaction when a transaction aborts. On large transaction systems this typically has meant mirrored disks with uninterruptible power supplies. With the simple transfer transactions here, the per-transaction log typically is not that large, and the log can be truncated once transactions commit. Because each secure coprocessor needs to handle only a handful of users, large amounts of stable storage should not be needed — because we have non-volatile memory in secure coprocessors, we only need to reserve some of this memory for logging. The log, the accounts data, and the controlling code are all protected from modification by the secure coprocessor, so account data are safe from all but bugs and catastrophic failures. Of course, the system should be designed so that users should have little or no incentive to destroy secure coprocessors that they can access — which should be natural when their own balances are stored on secure coprocessors, much like cash in wallets.

Note that this type of decentralized electronic currency is *not* appropriate for smart cards unless they can be made physically secure from attacks by their owners. Smart cards are only quasi-physically-secure in that their privacy guarantees stem solely from their portability. Secrets may be stored within smart cards because their users can provide the physical security necessary. Malicious users, however, can easily violate smart card integrity and insert false data.

If there is insufficient memory within the secure coprocessor to hold the account data for all its users, the code and the accounts database may be cryptographically paged to host memory or disk by first obtaining a cryptographic checksum. For the accounts data, encryption may also be employed since privacy is typically desired as well. The same considerations as those for checksums of system images apply here as well.

This electronic currency transfer is analogous to the transfer of rights (not to be confused with the copying of rights) in a capability-based protection system. Using the electronic money — e.g., expended when running a pay-per-use program — is analogous to the revocation of a capability.

What about the other models for handling electronic funds? With the credit card/check analogy, the authenticity of the promise of payment must be established. When the computer cannot keep secrets for users, there can be no authentication because nothing uniquely identifies users. Even when we assume that users can enter their passwords into a workstation without having the secrecy of their password compromised, we are still faced with the problem of providing privacy and integrity guarantees for network communication. We have similar problems as in host-to-host authentication in that cryptographic keys need to be exchanged somehow. If communications is in plaintext, attackers may simply record a transferral of a promise of payment and replay it to temporarily create cash. While security systems such as Kerberos [53], if properly implemented, can help to authenticate entities and create session keys, they reverted to the use of a centralized server and have similar problems to the bank rendezvous model.

With the bank rendezvous model, a "bank" server supervises the transfer of funds. While it is easy to enforce the access controls on account data, this suffers from problems with non-scalability, loss of anonymity, and easy denial of service from excessive centralization.

Because every transaction must contact the bank server, access to the bank service will be a performance bottleneck. The system does not scale well to a large user base — when the bank system must move from running on a single computer to several machines, distributed transaction systems techniques must be brought to bear in any case, so this model has no real advantage over the use of secure coprocessors in ease of implementation. Furthermore, if to the bank host becomes inaccessible, either maliciously or as a result of normal hardware failures, no agent can make use of any bank transfers. This model does not exhibit graceful degradation with system failures.

The model of electronic currency managed on a secure coprocessor not only can provide the properties of (1) anonymity, (2) conservation, and (3) decentralization, but it also degrades gracefully when secure coprocessors fail. Note that secure coprocessor data may be mirrored on disk and backed up after being properly encrypted, and so even the immediately affected users of a failed secure coprocessor should be able to recover their balance. The security administrators who initialized the secure coprocessor software will presumably have access to the decryption keys for this purpose. Careful procedural security must be required here, both for the protection of the decryption key and for auditing for double spending, since dishonest users might attempt to back up their secure coprocessor data, spend electronic money, and then intentionally destroy their coprocessor in the hopes of using their electronic currency twice. The amount of redundancy and the frequency of backups depends on the reliability guarantees desired; in reliable systems secure coprocessors may continually run self-checks when idle and warn of impending failures.

Contract Model

Our electronic contract model is built on the following two secure coprocessor provided primitive objects: (1) unforgeable tokens and (2) computer-enforced contracts.

Tokens are protected objects that are conserved by the secure coprocessors; they are freely transferable, but they can be created and destroyed only by the agent that issued them. Tokens are useful as electronic currency and to represent the execute-only right to a piece of software (much as in capability systems). In the case of rights such as execute-only rights, the token provides access to cryptographic keys that may be used (only) within the secure coprocessors to run code.

Contracts are another class of protected objects. They are created when two parties agree on a contract draft. Contracts contain binding clauses specifying actions that each of the parties must perform — or, in reality, actions that the secure coprocessors will enforce — and "method" clauses that may be invoked by certain parties (not necessarily restricted to just the parties who agreed on the contract). Time-based clauses and other event-based clauses may also exist. Contractual obligations may force the transfer of tokens between parties.

Contract drafts are typically instantiated from a contract template. We may think of a contract template as a standardized contract with blanks which are filled in by the two parties involved, though certainly "custom" contracts are possible. Contract negotiation consists of an offerer sending a contract template along with the bindings (values with which to fill in blanks) to the offeree. The offeree either accepts or rejects the contract. If it is accepted, a contract instance is created whereby the contract bindings are permanent, and any immediate clauses are executed. If the draft is rejected, the offeree may take the contract template and re-instantiate a new draft with different bindings to create a counter-offer, whereupon the roles of offerer and offeree are reversed.

From the time that a contract is accepted until it terminates, the contract is an active object running in one or more secure coprocessors. Methods may be invoked by users or triggered by external events (messages from the host, timer expiration). The method clauses of a contract are access controlled: they may be optionally invoked by only one party involved in the contract — or even by a third party who is under no contractual obligations.

Contractual clauses may require one of the parties to accept further contracts of certain types. One example of this is a requirement for some action to be performed prior to a certain time. Another is a contract between a distributor and a software house, where the software house requires the distributor to accept sales contracts from users for upgrading a piece of software.

Security Partitions In Networked Hosts

Network hosts, regardless of whether they use cryptography, have a de facto security partitioning that arises because different system components have different vulnerabilities to various attacks. Some of these vulnerabilities diminish when cryptography is used; similarly, the use of a secure coprocessor can be thought of as adding another layer with fewer vulnerabilities to the partitioning. By bootstrapping our system using a secure coprocessor and thus ensuring that the correct operating system is running, we can provide privacy and integrity guarantees on memory that were not possible before. In particular, public workstations can use secure coprocessors and cryptography to guarantee the privacy of disk storage and provide integrity checks. Let us see what kind of privacy/integrity guarantees are already available in the system and what new ones we can provide.

Subsystem	Vulnerabilities	
	Integrity	Privacy
Secure Coprocessor	None	None
Host RAM	On-line Physical Access	On-line Physical Access
Secondary Store	Off-line Physical Access	Off-line Physical Access
Network (communication)	On-line Remote Access	Off-line Analysis

Table 1: Subsystem Vulnerabilities Without Cryptographic Techniques

Table 1 shows the vulnerabilities of various types of memory when no cryptographic techniques are used. That memory within a secure coprocessor is protected against physical access is one of our axioms, and correctly using that to provide privacy and integrity at the logical level is a matter of using the appropriate software protection mechanisms. With the proper protection mechanisms within a secure coprocessor, data stored within a secure coprocessor can be neither read nor tampered with. Since we assume that we have a working secure coprocessor, we will also assume that the operating system was booted correctly and thus host RAM is protected against unauthorized logical access.[5] It is not, however, well protected against physical access — it is a simple matter to connect logic analyzers to the memory bus to listen passively to memory traffic. Furthermore, replacing the memory subsystem with multi-ported memory in order to allow remote unauthorized memory accesses is also a conceivable attack. While the effort required to do this in a way that is invisible to users may make it impractical, this line of

[5] We can assume that the operating system provides protected address spaces. Paging is assumed to be performed on either a local disk which is immune to all but physical attacks or a remote disk via encrypted network communication (see section below on coprocessor architecture). If we wish to protect against physical attacks for the former case, we may need to encrypt the data anyway or ensure that we can erase the paging data from the disk prior to shutting down.

attack can certainly not be entirely ruled out. Secondary storage may be more easily attacked than RAM since the data can be modified off-line; to do this, however, an attacker must gain physical access to the disk. Network communication is completely vulnerable to on-line eavesdropping and off-line analysis, as well as on-line message tampering. Since networks are inherently used for remote communication, it is clear that these may be remote attacks.

What protection guarantees can we provide when we use encryption? By using encryption when appropriate, we can guarantee privacy. Integrity of the data, however, is not guaranteed. The same vulnerabilities which allowed data modifications still exist as before; tampering, however, can be detected by using cryptographic checksums as long as the checksum values are stored in tamper-proof memory. Note also that the privacy that can be provided is relative to the data usage. If data in host RAM is to be processed by the host CPU, encrypting it within the secure coprocessor is useless — the data must remain vulnerable to on-line physical attacks on the host since it must appear in plaintext form to the host CPU. If the host RAM data is simply serving as backing store for secure coprocessor data pages, however, encryption is appropriate. Similarly, encrypting the secondary store via the host CPU protects that data against off-line privacy loss but not on-line attacks, whereas encrypting that data within the secure coprocessor protects that data against on-line privacy attacks as well, as long as that data need not ever appear in plaintext form in the host memory.

Subsystem	Vulnerabilities	
	Integrity	Privacy
Secure Coprocessor	None	None
Host RAM	On-line Physical Access	Host Processor Data
Secondary Store	Off-line Physical Access (detectable)	None
Network (communication)	On-line Remote Access (detectable)	None

Table 2: Subsystem Vulnerabilities With Cryptographic Techniques

For example, if we wish to send and read secure electronic mail, the encryption and decryption can be performed in the host processor since the data must reside within both hosts for the sender to compose it and for the receiver to read it. The exchange of the encryption key used for the message, however, requires secure coprocessor computation: the encryption for the key exchange needs to use secrets that must remain within the secure coprocessor, regardless of whether the key exchange uses a shared secret key or a public key scheme.[6]

System Architecture

This section discusses one possible architecture for a secure coprocessor software system. We will start off with a discussion of the constraints placed upon a secure coprocessor by the operational requirements of a security system – during system initialization and during normal, steady-state operation. We will next refine these constraints, examining various security functions and what their assumptions imply about trade-offs in a secure coprocessor. Following this, we will discuss the structure of the software in a secure coprocessor, ranging from a secure coprocessor kernel and its interactions with the host system to user-level applications.

Operational Requirements

We will start by examining how a secure coprocessor must interact with the host hardware and software during the bootstrap process and then proceed with the kinds of system services that a secure coprocessor should provide to the host operating system and user software.

The first issue to consider is how to fit a secure coprocessor into a system. This will guide us in the specification of the secure coprocessor software.

To be sure that a system is bootstrapped securely, secure hardware must be involved in the bootstrap process. Depending on the host hardware – whether a secure coprocessor could halt the boot process if it detects an anomaly – we may need to assume that the bootstrap ROM is secure. To ensure this, the system's address space either could be configured such that the boot vector and the boot code are provided by a secure coprocessor directly or we may simply assume that the boot ROM itself is a piece of secure hardware. Regardless, a secure coprocessor verifies the system software (operating system kernel, system related user-level

[6] The public key encryption requires no secrets and may be performed in the host; signing the message, however, requires the use of secret values and thus must be performed within the secure coprocessor.

software) by checking the software's signature against known values. We need to convince ourselves that the version of the software present in external, non-secure, non-volatile store (disk) is the same as that installed by a trusted party. Note that this interaction has the same problems faced by two hosts communicating via a non-secure network: if an attacker can completely emulate the interaction that the secure coprocessor would have had with a normal host system, it is impossible for the secure coprocessor to detect this. With network communication, we can assume that both hosts can keep secrets and build protocols based upon those secrets. With secure coprocessor/host interaction, we can make very few assumptions about the host — the best that we can do is to assume that the cost of completely emulating the host at boot time is prohibitively expensive.

At boot time, the primary duty of a secure coprocessor is to make sure that the system boots up securely; after booting, a secure coprocessor's role is to aid the host operating system by providing security functions not otherwise available. A secure coprocessor does not enforce the system's security policy – that is the job of the host operating system; since we know from the secure boot procedure that the correct operating system is running, we may rely on the host to enforce policy. When the system is up and running, a secure coprocessor provides the following security services to the host operating system: the host may use the secure coprocessor to verify the integrity of any data in the same manner that the secure coprocessor checks the integrity of system software; it may use the secure coprocessor to encrypt data to boost the natural security of storage media (see section above on security partitions); and it may use the secure coprocessor to establish secure, encrypted connections with remote hosts (key exchange, authentication, private key encryption, etc).[7]

Secure Coprocessor Architecture

The bootstrapping procedure described above made assumptions about the capability of a secure coprocessor. Let us refine what requirements we have on the secure coprocessor software and hardware.

When a secure coprocessor verifies that the system software is the correct version, we are assuming that a secure coprocessor has secure, tamper-proof memory

[7]Presumably the remote hosts will also contain a secure coprocessor, though everything will work fine as long as the remote hosts follow the appropriate protocols. The final design must take into consideration the possibility of remote hosts without secure coprocessors.

which remembers a description of the correct version of the system software. If we assume that proposed functions such as MD5 [44], multi-round Snefru [31], or IBM's MDC [25] are one-way hash functions, then the only requirement is that the memory is protected from writing by unauthorized individuals. Otherwise, we must use cryptographic checksums such as Karp and Rabin's technique of *fingerprinting*, which uses a family of hash functions with good error-detection capabilities. This technique requires that the memory be protected against read access as well, since both the hash value and the index selecting the particular hash function must be secret. In a similar manner, cryptographic operations such as authentication, key exchange, and secret key encryption all require that secrets be kept. Thus a secure coprocessor must have memory that is inaccessible by everybody except the secure coprocessor — enough private NVM to store the secrets, plus possibly volatile private memory for intermediate calculations in running the protocols.

There are a number of architectural tradeoffs for a secure coprocessor, the crucial dimensions being processor speed and memory size. They together determine the class of cryptographic algorithms that are practical.

The speed of the secure coprocessor may be traded off for memory in the implementation of the cryptographic algorithms. We observed in [54] that Karp-Rabin fingerprinting may be sped up by about 25% with a 256-fold table-size increase. Intermediate size tables may be used to yield intermediate speedups at a slightly higher increase in code size. Similar tradeoffs can be found for software implementations of the DES.

The amount of real memory required may be traded off for speed by employing cryptographic techniques: we need only enough private memory for an encryption key and a data cache, plus enough memory to perform the encryption if no encryption hardware is present. Depending on the throughput requirements, hardware assist for encryption may be included — where software is used to implement encryption, private memory must be provided for intermediate calculations. A secure coprocessor can securely page its private memory to either the host's physical memory (and perhaps eventually to an external disk) by first encrypting it to ensure privacy. Cryptographic checksums can provide error detection, and any error correcting encoding should be done *after* the encryption. This cryptographic paging is analogous to paging of physical pages to virtual memory on disk, except for different cost coefficients, and well-known analysis techniques can be used to tune such a system. The variance in costs will likely lead to new tradeoffs: cryptographic checksums are easier to calculate than encryption (and therefore faster modulo hardware support), so providing integrity alone is less expensive than providing privacy as well. On the other hand, if the computation can reside entirely on a secure coprocessor, both privacy and integrity can be provided for free.

Secure Coprocessor Software

With partitioned applications that must have parts loaded into a secure coprocessor to run and perhaps paging of secure coprocessor tasks, a small, simple security kernel is needed for the secure coprocessor. What makes this kernel different from other security kernels is the partitioned system structure.

Like normal workstation (host) kernels, the secure coprocessor kernel must provide separate address space if vendor and user code is to be loaded into the secure coprocessor – even if we implicitly trust vendor and user code, providing separate address spaces helps to isolate the effects of programming errors. Unlike the host's kernel, many services are not required: terminal, network, disk, and other device drivers need not be part of the secure coprocessor. Indeed, since both the network and disk drives are susceptible to tampering, requiring their drivers to reside in the secure coprocessor's kernel is overkill – network and file-system services from secure coprocessor tasks can simply be forwarded to the host kernel for processing. Normal operating system services such as printer service, electronic mail, etc. are entirely inappropriate in a secure coprocessor – these system daemons can be eliminated entirely.

The only services that are crucial to the operation of the secure coprocessor are (1) secure coprocessor resource management; (2) communications; (3) key management; and (4) encryption services. Within resource management we include task allocation and scheduling, virtual memory allocation and paging, and allocation of communication ports. Under communications we include both communication among secure coprocessor tasks and communication to host tasks; it is by communicating with host system tasks that proxy services are obtained. Under key management we include the management of secrets for authentication protocols, cryptographic keys for protecting data as well as execute-only software, and system fingerprints for verifying the integrity of system software. With the limited number of services needed, we can easily envision using a microkernel such as Mach 3.0 [22]: we need to add a communications server and include a key management service to manage non-volatile key memory. The kernel must be small for us to trust it; we have more confidence that it can be debugged and verified.

Key Management

A core portion of the secure coprocessor software is code to manage keys. Authentication, key management, fingerprints, and encryption crucially protect the integrity of the secure coprocessor software and the secrecy of private data, including the secure coprocessor kernel itself. A permanent part of a bootstrap loader, in ROM or in NVM, controls the bootstrap process of the secure coprocessor itself. Like bootstrapping the host processor, this loader verifies the secure coprocessor kernel before transferring control to it.

The system fingerprints needed for checking system integrity must reside entirely in NVM or be protected by encryption while being stored on an external storage device – the key for which must reside solely in the secure NVM. If the latter approach is chosen, new keys must be selected[8] to prevent replay attacks where old, potentially buggy secure coprocessor software is reintroduced into the system. Depending on the cryptographic assumptions made in the algorithm, the storage of the fingerprint information may require just integrity or both integrity and secrecy. For the cases of MD4, MDC, and Snefru, integrity of the integrity check information is sufficient; for the case of the Karp-Rabin fingerprint, both integrity and secrecy are required.

Other protected data held within the secure coprocessor's NVM include administrative authentication information needed to update the secure coprocessor software. We assume that a security administrator is authorized to upgrade secure coprocessor software, and that only the administrator may authenticate his identity properly to the secure coprocessor. The authentication data for this operation can be updated along with the rest of the secure coprocessor system software; in either case, the upgrade must appear transactional, that is, it must have the properties of *permanence*, where results of completed transactions are never lost; *serializability*, where there is a sequential, non-overlapping view of the transactions; and *failure atomicity*, where transactions either complete or fail such that any partial results are undone. The non-volatility of the memory gives us permanence automatically, if we assume that only catastrophic failures (or intentional sabotage) can destroy the NVM; serializability, while important for multi-threaded applications, can be

[8]One way is to use a cryptographically secure random number generator, the state of which resides entirely in NVM.

easily enforced if we permit only a single upgrade operation to be in progress at a time (this is an infrequent operation and does not require parallelism); and the failure atomicity guarantee can be provided easily as long as the non-volatile memory subsystem provides an atomic store operation. Update transactions need not be distributed nor nested; this simplifies the implementation immensely.

Machine-User Authentication

With secure coprocessors, we can perform all the necessary security functions to verify the integrity of the host system. The secure coprocessor may believe that the host system is clean, but how is the user to be convinced of this? After all, the secure coprocessor within the computer may have been replaced with a Trojan horse unit.

Smart-Cards

One solution to this is the use of smart-cards. Users can use advanced smart-cards to run an authentication procedure to verify the secure coprocessor's identity. Since secure coprocessors' identity-proofs can be based on a zero-knowledge protocol, no secret information needs to be stored in smart-cards unless smart-cards are to also aid users in authenticating themselves to systems, in which case the only secrets would be those belonging to the users. By the virtue of their portability, users can carry smart-cards at all times and thus provide the physical security needed.

Remote Services

Another way to verify that a secure coprocessor is present is to ask a third-party entity — such as a physically sealed third-party computer — to check for the user. Often, this service can also be provided by normal network servers machines such as file-servers. The remote services must be difficult to emulate by attackers. Users may rely on noticing the absence of these services to detect that something is amiss with the secure coprocessor. This necessarily implies that these remote services must be available *before* the users authenticate to the system.

Unlike authentication protocols reliant on accessing central authentication servers, this authentication happens once, at boot time. The identity being proven is that of the secure coprocessor – users may be confident that the workstation contains an authentic secure coprocessor if access to *any* normal remote service can be obtained. This is because in order to successfully authenticate to obtain the service, attackers must either break the authentication protocol, break the physical security in the secure coprocessor, or bypass the physical security around the remote server. As long as the remote service is sufficiently complex, attackers will not be able to emulate it.

Relationship With Previous Work

Partitioning security is not new. The method of embodying physical security in a secure coprocessor, however, *is* new, and it has been made possible only recently due to advances in packaging technology [62]. Certainly, the need for physical security is widely described in standard textbooks. For example, one book states that “physical security controls (locked rooms, guards, and the like) are an integral part of the security solution for a central computing facility.”[18]

We can trace several analogs to this approach of partitioning security in previous work. The logical partitioning of security in the literature [58, 61] of dividing the system into a “Trusted Computing Base” (TCB) and applications in some sense heralds this idea — the security partition was firmly drawn between the user and the machine; it not only included the logical security of the operating system part of the TCB, but also the physical security of the TCB hardware installation (machine rooms, etc).

Systems such as Kerberos [53] move that security partition for distributed systems toward including just one trusted server behind locked doors. This approach, however, still has serious security problems: client machines are often physically exposed and users are provided with no real assurances of their logical integrity, and the centralized server approach offers attackers a central point of attack — the system catastrophically fails when the central server is compromised [5]. Certainly, it does not offer much in terms of providing fault tolerance with distributed computing.

More recently, the partitioning in Strongbox [54] more clearly points the way toward minimizing the number of assumptions about trusted components in a secure system and clearly defining the security partition boundaries and security assumptions. In that system, the base security system was divided into trusted servers which, assuming protected address spaces, allowed security to be bootstrapped to application servers and clients. Unfortunately, while the system has better degradation properties, it could deliver system integrity assurances only by assuming trusted-operator-assisted bootstrapping. Table 3 shows the various types of systems and their basic assumptions as well as typical cryptographic assumptions.

System Type	Basic Assumptions	Cryptographic Assumptions
Conventional Non-distributed e.g., Unix password	Physical Security of Central Mainframe	DES cannot be inverted
Conventional Distributed e.g., Kerberos	Physical Security of Authentication Server	DES cannot be inverted
Self-Securing Distributed e.g., Strongbox	Physical Security of a Quorum of White Pages Servers	DES cannot be inverted; factoring is hard
Secure Coprocessors e.g., Dyad	Physics (Tampering destroys cryptographic data)	DES cannot be inverted; factoring is hard

Table 3: Basic Assumptions of Security Systems

The secure coprocessor approach minimizes the basic assumptions and can address all of the problems with the approaches cited above. By implementing cryptographic protocols within a secure coprocessor, we can be assured that they will execute correctly and that the secrets required by the various protocols are indeed kept secret. By using the secure coprocessor to verify the integrity of the rest of the system, we can give users greater assurance that the system has not been compromised and that the system has securely bootstrapped.

In addition to the work mentioned above, there are many other relevant works on security related issues: [3, 56, 57, 63] discusses issues in the design and implementation of physically secure system components. Research on cryptosystems and cryptographic protocols which are important tools for secure network communication can be found in [2, 5, 7, 11, 15, 16, 17, 19, 20, 21, 24, 29, 34, 35, 37, 42, 45, 47, 48, 49, 53]. More general information on some of the number theoretic tools behind many of these protocols may be found in [33, 36, 40, 51]. The tools for checking data integrity are described in [27, 28, 38, 41].

Research on protection systems and general distributed system security may be found in [39, 43, 46].

[9] provides a logic for analyzing authentication protocols, and [23] extends the formalism.

General security/cryptography information can be found in [14], the new proposed federal criteria for computer security [61], and in older standards such as the “Orange book” [58] and the “Red book” [59]. General information on cryptography can be found in [13, 32].

Glossary of Terms and Acronyms

At the suggestion of the editors of this anthology, the authors have included a small glossary of some technical terms which may be unfamiliar to readers who are not computer scientists. We have also included a list of acronyms used in the text. Readers who are interested in reading more broadly in operating systems may wish to read [50]; those who are interested in cryptography may wish to read [13, 32].

Glossary

authentication The process by which identity (or other credentials) of a user or computer are verified. *Authentication protocols* are series of stylized exchanges which proves identity. The simplest example is that of the *password*, which is a secret shared between the two parties involved; password-based schemes are cryptographically weak because any eavesdropper that overhears the password may subsequently impersonate as one of the parties.

More sophisticated authentication protocols use techniques from *cryptography* to eliminate the problem with eavesdroppers. In these systems, the password is used as a *key* to parameterized the *cryptographic function*. Some of these protocols depend on the strength of the particular cryptographic function involved and may leak a little bit of information about the keys used each time the protocol is run. A more powerful class of authentication protocols known as *zero knowledge authentication* provably do not leak any information. Zero knowledge protocols are an important special case of zero knowledge proofs, which have a number of important applications in computer science.

authentication protocol See *authentication*.

bootstrap The process of initializing a computer.

checksum A small output value computed using some known checksum function from input data. The checksum function is chosen so that changes in the input will likely result in a different output checksum. Checksums are intended to guard against the corruption of the original data, typically due to noisy communication channels or bad storage media.

checksum, cryptographic A checksum computed using a function where it is infeasible (other than by exhaustively searching through all possible input) to find another input which would have the same output value. Cryptographic checksums may be used to guard against malicious as well as unintentional modification of the data, since the correct checksum may be delivered by a (more expensive) tamper-proof means, and the recipient of the data may recompute the checksum to verify that the data has not been corrupted.

Cryptographic checksums are generally computed by applying a conjectured[9] *one-way hash function* to the input. One-way hash functions have the property that it is computationally infeasible (except by exhaustive search) to invert — that is, given only the output value, find an input to the function that would give that output.

Another approach to cryptographic checksums is based on parameterizing the checksum function by a secret key. The Karp-Rabin *fingerprint* system uses secret keys to checksum data, forcing attackers to guess the secret key correctly; the secret key and resultant checksum must be delivered via a secure communication channel which guarantees against both tampering and eavesdropping.

ciphertext See *cryptography*.

client-server model A model of distributed computing where *client* software on one computer makes requests to *server* software on the same or a different computer. Examples of the client-server model include distributed file systems, electronic mail, and file transfer.

client See *client-server model*.

cryptography The enciphering and deciphering of messages in secret codes. The enciphered messages are known as *ciphertext*; the original (or decoded) messages are known as *plaintext*. *Cryptographic functions* are used to transform between plaintext and ciphertext, and *keys* are used to parameterize the cryptographic function used (or alternatively, *select* the cryptographic function from a set of functions). The security of cryptographic systems depend on the secrecy of the keys. Generally cryptographic systems may

[9]Theorists have not been able to prove any particular function to be one-way. However, there are several functions that seem to work in practice.

be classified into two different kinds: *private key* (or symmetric) systems; and *public key* (or asymmetric) systems.

In private key systems, a single value or *key* is used to parametrically encode and decode messages. Thus, both the sender and the receiver of enciphered messages must know the same key.

In contrast, public key systems have are parameterized by pairs of keys, one for encryption and the other for decryption. Knowing one of the two keys in a pair does not help in determining the other. Thus, the encryption key may be widely published while the decryption key is kept secret; anyone may send encrypted messages that can only be read by the owner of the secret key.

Alternatively, the decryption key may be widely published while the encryption key is kept secret; this is used in *digital* or *cryptographic signature* schemes where the owner of the encryption key uses the secret key to encrypt, or *sign* a digital document. The result is a *digital signature*, which may be decrypted by anyone to verify that it corresponds to the original digital document. Since the secret key is known by the owner of that key, the cryptographic signature serves as evidence that the data has been processed by that person.

Cryptographic systems are attacked in two main ways. The first is that of *exhaustive search*, where the attacker tries all possible keys in an attempt to decipher messages or derive the key used. The second class is that of *short cut* attacks where properties of the cryptographic system are used to speed up the search.

Cryptographic attacks may be further classified by the amount of information available to the attacker. A *ciphertext-only attack* is one where the only information available to attacker are the ciphertext. A *known-plaintext attack* is one where the attacker has available corresponding pairs of plaintext and ciphertext. A *chosen-plaintext attack* is one where the attacker may chose plaintext messages and obtain the corresponding ciphertext in an attempt to decrypt other messages or derive the key. A *chosen-ciphertext* attack is one where the attacker may chose some ciphertext messages and obtain their corresponding plaintext in an attempt to derive the key used.

cryptographic checksum See checksum, cryptographic.

cryptographic signature See *cryptography*.

cryptography, private key See *cryptography*.

cryptography, public key See *cryptography*.

daemon a program which runs unattended, providing system services to users and application programs; examples includes electronic mail transport, printer spooling, and remote login service.

Data Encryption Standard A U.S. federal data encryption standard adopted in 1976, recommended by the NSA that the Federal Reserve Board use DES for electronic funds transfer applications. [60]

digital signature See *cryptography*.

executable binary A file containing an executable program; typically includes standardized headers.

fingerprint See *checksum, cryptographic*.

kernel The program which is the core of an operating system, providing the lowest level services upon which all other programs are built. There are two major schools of designing kernels. The traditional method of *monolithic kernels* places all basic system services within the kernel. A more modular kernel design approach, that of *microkernels*, provide many of the system services by a set of separate system servers rather than the kernel itself.

known-plaintext attack See *cryptography*.

microkernel See *kernel*.

monolithic kernel See *kernel*.

National Security Agency The U.S. agency responsible for military cryptography and signals intelligence; recently more active in civilian cryptography.

nonvolatile memory Memory that retains its contents even after power is removed.

one-way hash function See *cryptography*.

operating system The collection of programs which provide the interface between the hardware and the user and application programs.

paging See *virtual memory*.

plaintext See *cryptography*.

private key cryptography See *cryptography*.

public key cryptography See *cryptography*.

server See *client-server model*.

virtual memory A technique of providing the appearance of having more primary memory than actually exists by moving primary memory (RAM) contents to/from secondary storage (disk) as needed. The transfer of sections of this virtual memory between physical memory and secondary storage is called *paging*.

zero knowledge protocol See *authentication*.

Acronyms

CPU Central Processing Unit

DES Data Encryption Standard

I/O Input/Output

IC Integrated circuit

NSA National Security Agency

NVM Non-volatile memory

PC Personal Computer.

RAM Random Access Memory.

ROM Read Only Memory.

VM Virtual memory

References

[1] M. Abadi, M. Burrows, C. Kaufman, and B. Lampson. Authentication and delegation with smart-cards. Technical Report 67, DEC Systems Research Center, October 1990.

[2] W. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr. RSA and Rabin functions: Certain parts are as hard as the whole. *SIAM Journal on Computing*, 17(2):194–209, April 1988.

[3] R. G. Andersen. The destiny of DES. *Datamation*, 33(5), March 1987.

[4] E. Balkovich, S. R. Lerman, and R. P. Parmelee. Computing in higher education: The Athena experience. *Communications of the ACM*, 28(11):1214–1224, November 1985.

[5] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. Submitted to *Computer Communication Review*, 1990.

[6] Robert M. Best. Preventing software piracy with crypto-microprocessors. In *Proceedings of IEEE Spring COMPCON 80*, page 466, February 1980.

[7] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, November 1984.

[8] Andrea J. Borr. Transaction monitoring in Encompass (TM): Reliable distributed transaction processing. In *Proceedings of the Very Large Database Conference*, pages 155–165, September 1981.

[9] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. In *Proceedings of the Twelfth ACM Symposium on Operation Systems Principles*, 1989.

[10] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, October 1985.

[11] Ben-Zion Chor. *Two Issues in Public Key Cryptography: RSA Bit Security and a New Knapsack Type System*. ACM Distinguished Dissertations. MIT Press, 1986.

[12] C. J. Date. *An Introduction to Database Systems Volume 2*. The System Programming Series. Addison-Wesley, Reading, MA, 1983.

[13] Donald Watts Davies and W. L. Price. *Security for Computer Networks: an Introduction to Data Security in Teleprocessing and Electronic Funds Transfer, 2nd Edition*. Wiley, 1989.

[14] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[15] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-26(6):644–654, November 1976.

[16] Uriel Feige, Amos Fiat, and Adi Shamir. Zero knowledge proofs of identity. In *Proceedings of the 19th ACM Symp. on Theory of Computing*, pages 210–217, May 1987.

[17] U. Fiege and A. Shamir. Witness indistinguishable and witness hiding protocols. In *Proceedings of the 22nd ACM Symp. on Theory of Computing*, pages 416–426, May 1990.

[18] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Co, New York, 1988.

[19] S. Goldwasser and M. Sipser. Arthur Merlin games versus zero interactive proof systems. In *Proceedings of the 17th ACM Symp. on Theory of Computing*, pages 59–68, May 1985.

[20] Shafi Goldwasser and Silvio Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, 1982.

[21] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, May 1985.

[22] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[23] Nevin Heintze and J. D. Tygar. A critique of Burrows', Abadi's, and Needham's *a logic of authentication*. To Appear.

[24] Maurice P. Herlihy and J. D. Tygar. How to make replicated data secure. In *Advances in Cryptology, CRYPTO-87*. Springer-Verlag, August 1987. To appear in *Journal of Cryptology*.

[25] IBM Corporation. *Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*, sc40-1675-1 edition.

[26] R. R. Jueneman, S. M. Matyas, and C. H. Meyer. Message authentication codes. *IEEE Communications Magazine*, 23(9):29–40, September 1985.

[27] Richard M. Karp. 1985 Turing award lecture: Combinatorics, complexity, and randomness. *Communications of the ACM*, 29(2):98–109, February 1986.

[28] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. Technical Report TR-31-81, Aiken Laboratory, Harvard University, December 1981.

[29] Michael Luby and Charles Rackoff. Pseudo-random permutation generators and cryptographic composition. In *Proceedings of the 18th ACM Symp. on Theory of Computing*, pages 356–363, May 1986.

[30] J. McCrindle. *Smart Cards*. Springer Verlag, 1990.

[31] R. Merkle. A software one-way function. Technical report, Xerox PARC, March 1990.

[32] C. Meyer and S. Matyas. *Cryptography*. Wiley, 1982.

[33] G. L. Miller. Riemann's hypothesis and a test for primality. *Journal of Computing and Systems Science*, 13:300–317, 1976.

[34] R. M. Needham. Using cryptography for authentication. In Sape Mullender, editor, *Distributed Systems*. ACM Press and Addison-Wesley Publishing Company, New York, New York, 1989.

[35] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978. Also Xerox Research Report, CSL-78-4, Xerox Research Center, Palo Alto, CA.

[36] I. Niven and H. S. Zuckerman. *An Introduction to the Theory of Numbers*. Wiley, 1960.

[37] Michael Rabin. Digitized signatures and public-key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1979.

[38] Michael Rabin. Fingerprinting by random polynomials. Technical Report TR-81-15, Center for Research in Computing Technology, Aiken Laboratory, Harvard University, May 1981.

[39] Michael Rabin and J. D. Tygar. An integrated toolkit for operating system security (revised version). Technical Report TR-05-87R, Center for Research in Computing Technology, Aiken Laboratory, Harvard University, August 1988.

[40] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12:128–138, 1980.

[41] Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM Journal on Computing*, 9:273–280, 1980.

[42] Michael O. Rabin. Efficient dispersal of information for security and fault tolerance. Technical Report TR-02-87, Aiken Laboratory, Harvard University, April 1987.

[43] B. Randell and J. Dobson. Reliability and security issues in distributed computing systems. In *Proceedings of the Fifth IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 113–118, January 1985.

[44] R. Rivest and S. Dusse. The MD5 message-digest algorithm. Manuscript, July 1991.

[45] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[46] M. Satyanarayanan. Integrating security in a large distributed environment. *ACM Transactions on Computer Systems*, 7(3):247–280, August 1989.

[47] A. W. Schrift and A. Shamir. The discrete log is very discreet. In *Proceedings of the 22nd ACM Symp. on Theory of Computing*, pages 405–415, May 1990.

[48] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–614, November 1979.

[49] Adi Shamir and Eli Biham. Differential cryptanalysis of DES-like cryptosystems. In *Advances in Cryptology, CRYPTO-90*. Springer-Verlag, August 1990.

[50] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Concepts, 3rd Edition*. Addison-Wesley, 1991.

[51] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6:84–85, March 1977.

[52] Alfred Z. Spector and Michael L. Kazar. Wide area file service and the AFS experimental system. *Unix Review*, 7(3), March 1989.

[53] J. G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Conference Proceedings*, pages 191–200, Winter 1988.

[54] J. D. Tygar and Bennet S. Yee. Strongbox: A system for self securing programs. In Richard F. Rashid, editor, *CMU Computer Science: 25th Anniversary Commemorative*. Addison-Wesley, 1991.

[55] J. D. Tygar and Bennet S. Yee. Cryptography: It's not just for *electronic* mail anymore. Technical Report CMU-CS-93-107, Carnegie Mellon University, March 1993.

[56] U. S. National Institute of Standards and Technology. Capstone chip technology press release, April 1993.

[57] U. S. National Institute of Standards and Technology. Clipper chip technology press release, April 1993.

[58] U.S. Department of Defense, Computer Security Center. Trusted computer system evaluation criteria, December 1985.

[59] U.S. Department of Defense, Computer Security Center. Trusted network interpretation, July 1987.

[60] U.S. National Bureau of Standards. Federal information processing standards publication 46: Data encryption standard, January 1977.

[61] U.S. National Institute of Standards and Technology and National Security Agency. Federal criteria for information technology security, December 1992. Draft.

[62] Steve H. Weingart. Physical security for the μABYSS system. In *Proceedings of the IEEE Computer Society Conference on Security and Privacy*, pages 52–58, 1987.

[63] Steve R. White and Liam Comerford. ABYSS: A trusted architecture for software protection. In *Proceedings of the IEEE Computer Society Conference on Security and Privacy*, pages 38–51, 1987.

[64] Steve R. White, Steve H. Weingart, William C. Arnold, and Elaine R. Palmer. Introduction to the Citadel Architecture: Security in Physically Exposed Environments. Technical Report RC16672, Distributed Security Systems Group, IBM Thomas J. Watson Research Center, March 1991. Version 1.3.

[65] Jeannette Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietzke, Richard Lerner, and Su-Yuen Ling. The Avalon language: A tutorial introduction. In Jeffery L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors, *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.