Using Secure Coprocessors

Bennet Yee

May 1994

CMU-CS-94-149

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Thesis Committee:
Doug Tygar, Chair
Rick Rashid
M. Satyanarayanan
Steve White, IBM Research

Copyright © 1994 Bennet Yee

This research was sponsored in part by the Advanced Research Projects Agency under contract number F19628-93-C-0193; the Avionics Laboratories, 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; IBM; Motorola; the National Science Foundation under Presidential Young Investigator Grant CCR-8858087; TRW; and the U. S. Postal Service.

The views and conclusions in this document are those of the authors and do not necessarily represent the official policies or endorsements of any of the research sponsors.

Keywords: authentication, coprocessor, cryptography, integrity, privacy, security

Abstract

How do we build distributed systems that are secure? Cryptographic techniques can be used to secure the communications between physically separated systems, but this is not enough: we must be able to guarantee the privacy of the cryptographic keys and the integrity of the cryptographic functions, in addition to the integrity of the security kernel and access control databases we have on the machines. Physical security is a central assumption upon which secure distributed systems are built; without this foundation even the best cryptosystem or the most secure kernel will crumble. In this thesis, I address the distributed security problem by proposing the addition of a small, physically secure hardware module, a *secure coprocessor*, to standard workstations and PCs. My central axiom is that secure coprocessors are able to maintain the privacy of the data they process.

This thesis attacks the distributed security problem from multiple sides. First, I analyze the security properties of existing system components, both at the hardware and software level. Second, I demonstrate how physical security requirements may be isolated to the secure coprocessor, and showed how security properties may be bootstrapped using cryptographic techniques from this central nucleus of security within a combined hardware/software architecture. Such isolation has practical advantages: the nucleus of security-relevant modules provide additional separation of concern between functional requirements and security requirement, and the security modules are more centralized and their properties more easily scrutinized. Third, I demonstrate the feasibility of the secure coprocessor approach, and report on my implementation of this combined architecture on top of prototype hardware. Fourth, I design, analyze, implement, and measure performance of cryptographic protocols with super-exponential security for zero-knowledge authentication and key exchange. These protocols are suitable for use in security critical environments. Last, I show how secure coprocessors may be used in a fault-tolerant manner while still maintaining their strong privacy guarantees.

Contents

1 Introduction and Motivation **1**

2 Secure Coprocessor Model **5**
2.1 Physical Assumptions for Security 5
2.2 Limitations of Model 6
2.3 Potential Platforms 7
2.4 Security Partitions 8
2.5 Machine-User Authentication 10
2.6 Previous Work 11

3 Applications **13**
3.1 Host Integrity Check 13
3.1.1 Host Integrity with Secure Coprocessors 13
3.1.2 Absolute Limits 15
3.1.3 Previous Work 16
3.2 Audit Trails 19
3.3 Copy Protection 19
3.3.1 Copy Protection with Secure Coprocessors 20
3.3.2 Previous Work 22
3.4 Electronic Currency 22
3.4.1 Electronic Money Models 22
3.4.2 Previous Work 26
3.5 Secure Postage 28
3.5.1 Cryptographic Stamps 29
3.5.2 Software Postage Meters 31

4 System Architecture **35**
4.1 Abstract System Architecture 35
4.1.1 Operational Requirements 35
4.1.2 Secure Coprocessor Architecture 36
4.1.3 Crypto-paging and Sealing 37
4.1.4 Secure Coprocessor Software 37
4.1.5 Key Management 38
4.2 Concrete System Architecture 39
4.2.1 System Hardware 39
4.2.2 Host Kernel 43

4.2.3 Coprocessor Kernel 47

5 Cryptographic Algorithms/Protocols 53
5.1 Description of Algorithms 53
5.1.1 Key Exchange 54
5.1.2 Authentication 56
5.1.3 Merged Authentication and Secret Agreement 58
5.1.4 Practical Authentication and Secret Agreement 60
5.1.5 Fingerprints 61
5.2 Analysis of Algorithms 62
5.2.1 Key Exchange 62
5.2.2 Authentication 62
5.2.3 Merged Authentication and Secret Agreement 64
5.2.4 Practical Authentication and Secret Agreement 65
5.2.5 Fingerprints 66

6 Bootstrap and Maintenance 71
6.1 Simple Secure and Bootstrap 72
6.2 Flexible Secure Bootstrap and Maintenance 72
6.3 Hardware-level Maintenance 73
6.4 Tolerating Hardware Faults 74

7 Verification and Potential Failures 77
7.1 Hardware Verification 77
7.2 System Software Verification 78
7.3 Failure Modes 79
7.4 Previous Work 80

8 Performance 81
8.1 Cryptographic Algorithms 81
8.2 Crypto-Paging 83

9 Conclusion and Future Work 85

List of Figures

3.1 Copy-Protected Software Distribution 21
3.2 Postage Meter Indicia 28
3.3 PDF417 encoding of Abraham Lincoln's Gettysburg Address 30

4.1 Dyad Prototype Hardware 40
4.2 DES Engine Data Paths 41
4.3 Host Software Architecture 44

5.1 Fingerprint residue calculation 67
5.2 Fingerprint calculation (C code) 68

List of Tables

2.1 Subsystem Vulnerabilities Without Cryptographic Techniques 9
2.2 Subsystem Vulnerabilities With Cryptographic Techniques 9

8.1 Cryptographic Algorithms Run Time 82

Acknowledgements

I would like to thank Doug Tygar, without whom this thesis would not have been possible. I would also like to thank my parents, without whom I would not have been possible.

I was fortunate to have a conscientious and supportive thesis committee: thanks to Rick Rashid for his helpful advice (and his colorful metaphors); thanks to Steve White and his crew at IBM Research for their insights into secure coprocessors (and for their generous hardware grant); thanks to Satya for systems advice.

Special thanks go to Michael Rabin, whose ideas inspired my protocol work. I am also indebted to Alfred Spector, who helped Doug and I with Strongbox, the predecessor to Dyad. Steve Guattery was generous with his time in helping with the proof reading. (All rerors remaining are mine, of course.)

Thanks to Wayne Wilkerson and his staff at the U. S. Postal Service for many discussions related to cryptographic stamps.

Thanks to Symbol Technologies Inc for figure 3.3.

Chapter 1

Introduction and Motivation

> Is privacy the first roadkill on the Information Superhighway? [1] Will superhighwaymen way lay new settlers to this electronic frontier?

While these questions may be too steeped in metaphor, they raise very real concerns. The National Information Infrastructure (NII) [32] grand vision would have remote computers working harmoniously together, communicating via an "electronic superhighway," providing new informational goods and services for all.

Unfortunately, many promising NII applications demand difficult-to-achieve distributed security properties. Electronic commerce applications such as electronic stock brokerage, pay-per-use, and metered services have strict requirements for authorization and confidentiality — providing trustworthy authorization requires user authentication; providing confidentiality and privacy of communications requires end-to-end encryption. As a result of the need for encryption and authentication, our systems must be able to maintain the secrecy of the keys used for encrypting communications, the secrecy of the user-supplied authentication data (e.g., passwords), and the integrity of the authentication database against which the user-supplied authentication data is checked. Furthermore, hand in hand with the need for privacy is the need for system integrity: without the integrity of the system software that mediates access to protected objects or the integrity of the access control database, no system can provide any sort of privacy guarantee.

Can strong privacy and integrity properties be achieved on real, distributed systems?

The most common computing environments today on college campuses and workplaces are open computer clusters and workstations in offices, all connected by networks. Physical security is rarely realizable in these environments: neither computer clusters nor offices are secure against casual intruders,[2] let alone the determined expert. Even if office locks were safe, the physical media for our local networks are often but a ceiling tile away — any hacker who knows her raw bits can figure out how to tap into a local network using a PC. To make matters worse, for many security applications we must be able to protect our systems against the occasional untrustworthy user as well as intruders from the outside.

---

[1] The source of this quote is unclear; one paraphrased version appeared in print, as "If privacy isn't already the first roadkill along the information superhighway, then it's about to be" [55], and other variants of this have appeared in diverse locations.

[2] The knowledge of how to pick locks is widespread; many well-trained engineers can pick office locks [96].

Standard textbook treatments of computer security assert that physical security is a necessary precondition to achieving overall system security. While this may have been a requirement that was readily realizable for yesterday's computer centers with their large mainframes, it is clearly not a realistic expectation for today's PCs and workstations: their physical hardware is easily accessible by both authorized users and malicious attackers alike. With complete physical access, the adversaries can mount various attacks: they can copy the hard disk's contents for offline analysis; replace critical system programs with trojan horse versions; replace various hardware components to bypass logical safeguards, etc.

By making the processing power of workstations widely and easily available, we have made the entire system hardware accessible to interlopers. Without a foundation of physical security to build on, logical security guarantees crumble. How can we remedy this?

Researchers *have* realized the vulnerability of network wires and other communication media. They have brought tools from cryptography to bear on the problem of insecure communication networks, leading to a variety of key exchange and authentication protocols [25, 27, 30, 59, 67, 78, 80, 93, 98] for use with end-to-end encryption, providing privacy for network communications. Others have noted the vulnerability of workstations and their disk storage to physical attacks, and have developed a variety of secret sharing algorithms for protecting data from isolated attacks [39, 75, 86]. Tools from the field of consensus protocols can be applied as well. Unfortunately, all of these techniques, while powerful, still assume some measure of physical security, a property unavailable on conventional workstations and PCs. The gap between reality and the physical security assumption must be closed before these techniques can be implemented in a believable fashion.

Can we provide the necessary physical security to PCs and workstations without crippling their accessibility? Can real, secure electronic commerce applications be built in a networked, distributed computing environment? I argue that the answer to these questions is yes, and I have built a software/hardware system called *Dyad* that demonstrates my ideas.

In this thesis, I analyze the distributed security problem not just from the traditional cryptographic protocol viewpoint but also from the viewpoint of a hardware/software system designer. I address the need for physical security and show how we can obtain overall system security by bootstrapping from a limited amount of physical security that *is* achievable for workstation/PC platforms — by incorporating a *secure coprocessor* in a tamper-resistant module. This secure coprocessor may be realized as a circuit board on the system bus, a PCMCIA[3] card, or an integrated chip; in my Dyad system, it is realized by the Citadel prototype from IBM, a board-level secure coprocessor system.

I analyze the natural security properties inherent in secure coprocessor enhanced computers, and demonstrate how security guarantees can be strengthened by bootstrapping security using cryptographic techniques. Building on this analysis, I develop a combined software/hardware system architecture, providing a firm foundation upon which applications with stringent security requirements can be built. I describe the design of the Citadel

[3]Personal Computer Memory Card International Association

prototype secure coprocessor hardware, the Mach [2] kernel port running on top of it, the resultant system integration with the host platform, the security applications running on top of the secure coprocessor, and new, highly secure cryptographic protocols for key exchange and zero-knowledge authentication.[4]

By attacking the distributed security problem from all sides, I show that it is eminently feasible to build highly secure distributed systems, with bootstrapped security properties derived from physical security.

The next chapter discusses in detail what is meant by the term *secure coprocessor* and the basic security properties that secure coprocessors must possess. Chapter 3 outlines five applications that are impossible without the security properties provided by secure coprocessors. Chapter 4 describes the combined hardware/software system architecture of a secure coprocessor-enhanced host. I consider the basic operational requirements induced by the demands of security applications and then describe the actual system architecture as implemented in the Dyad secure coprocessor system prototype. Chapter 5 describes my new cryptographic protocols, and gives an in-depth analysis of their cryptographic strength. Chapter 6 addresses the security issues present when initializing a secure coprocessor, and presents techniques to make a secure coprocessor system fault tolerant. Additionally, I demonstrate techniques where proactive fault diagnostics may allow some classes of hardware faults to be detected and permit the replacement of a malfunctioning secure coprocessor. Chapter 7 shows how both the secure coprocessor hardware and system software may be verified, and examines the consequences of system privacy breaches. Chapter 8 gives performance figures for the cryptographic algorithms, the overhead incurred by crypto-paging, and the raw DMA transfer times for our prototype system. In chapter 9, I propose challenges for future developers of secure coprocessors.

[4]Some of this research was joint work: the design of Dyad, the secure applications, and the new protocols was done with Doug Tygar of CMU. The basic secure coprocessor model was developed with White, Palmer, and Tygar. The Citadel system was designed by Steve Weingart, Steve White, and Elaine Palmer of IBM; I debugged Citadel and redesigned parts of it.

Chapter 2

Secure Coprocessor Model

A secure coprocessor is a hardware module containing (1) a CPU, (2) bootstrap ROM, and (3) secure non-volatile memory. This hardware module is physically shielded from penetration, and the I/O interface to the module is the only way to access the internal state of the module. (Examples of packaging technology are discussed later in section 2.3.) This hardware module can store cryptographic keys without risk of release. More generally, the CPU can perform arbitrary computations (under control of the operating system); 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 secure non-volatile memory 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 the secure non-volatile memory provide the foundations needed to build distributed security systems.

With a firm security foundation available in the form of a secure coprocessor, greater security can be achieved for the host computer.

2.1. Physical Assumptions for Security

All security systems rely on a nucleus of assumptions. For example, it is often assumed that encryption systems are resistant to cryptanalysis. Similarly, I take as axiomatic that secure coprocessors provide private and tamper-proof memory and processing. These assumptions may be falsified: for example, attackers may exhaustively search cryptographic key spaces. Similarly, it may be possible to falsify my physical security axiom by expending enormous resources (possibly feasible for very large corporations or government agencies). I rely on a physical work-factor argument to justify my axiom, similar in spirit to intractability assumptions of cryptography. My secure coprocessor model does not depend on the particular technology used to satisfy the work-factor assumption. Just as cryptographic schemes may be scaled or changed to increase the resources required to penetrate a cryptographic

system, current security packaging techniques may be scaled or changed to increase the work-factor necessary to successfully bypass the secure coprocessor protections.

Chapter 3 shows how to build secure subsystems running partially on a secure coprocessor.

2.2. Limitations of Model

Confining all computation within secure coprocessors would ideally suit our security needs, but in reality we cannot — and should not — convert all of our processors into secure coprocessors. There are two main reasons: first, the inherent limitations of physical security techniques for packaging circuits; and second, the need to keep the system maintainable. Fortunately, as we shall see in chapter 3, we do not need to physically shield the entire computer. It suffices to physically protect only a portion of the computer.

If the secure coprocessor is sealed in epoxy or a similar material, heat dissipation requirements limit us to one or two printed circuit boards. 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; however, the security problems of external mass storage and networks will in all likelihood remain constant.

While it may be possible to secure package an entire multiprocessor, 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 and maintenance problems of making multiple processors and all memory secure.

Easy maintenance requires 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. Wholesale board swapping is a standard maintenance / hardware debugging technique, but defective boards are normally returned for repairs; with physical encapsulation, this will no longer be possible, thus driving up costs. Moreover, packaging considerations and the extra hardware development time imply that secure coprocessor's technology may lag behind the host system's technology — perhaps by one generation. The right balance between physically shielded and unshielded components depends on the class of intended applications. For many applications, only a small portion of the system must be protected.

What about system-level recovery after a hardware fault? If secrets are kept only within a single secure coprocessor, having to replace a faulty unit with a different one due to a will lead to data loss. After we replace a broken coprocessor with a good one, will we be able to continue running our applications? Section 6.4 gives techniques for periodic checkup testing and fault tolerant operation of secure coprocessors.

2.3. Potential Platforms

Several physically secure processors exist. This section describes some of these platforms, giving the types of attacks these systems resist, and system limitations arising from packaging technology.

The μABYSS [103] and Citadel [105] systems employ board-level protection. The systems include a standard microprocessor (Citadel uses an Intel 80386), some non-volatile (battery backed) RAM, and special sensing circuitry to detect intrusion into a protective casing around the circuit board. Additionally, Citadel includes fast (approximately 30 MBytes/sec) DES encryption hardware. The security circuitry erases non-volatile memory before attackers can penetrate far enough to disable the sensors or read memory contents.

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

Another attack is to dissolve the epoxy with solvents to expose the sensor wires. To block this attack, the epoxy is designed to be chemically "harder" than the sensor wires. Solvents will destroy at least one of the wires — and thus create an open-circuit — before the intruder can bypass the potting material and access the circuit board.

Yet another 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 memory contents. The systems contain temperature sensors which trigger erasure of secrets before the temperature drops below the critical level. (The system must have enough thermal mass to prevent rapid freezing — by being dipped into liquid nitrogen or helium, for example — and this places some limitations on the minimum size of the system. This has important implications for secure smartcard designers.)

The next step in sophistication is the high-powered laser attack. The idea is to use a high powered (ultraviolet) laser to cut through the epoxy and disable the sensing circuitry before it has a chance to react. To protect against such an attack, alumina or silica is added, causing the epoxy to absorb ultraviolet light. The generated heat creates mechanical stress, causing 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 [4, 99, 100] are special purpose encryption chips. These integrated circuit chips are reportedly

designed to destroy key information (and perhaps other important encryption parameters — the encryption algorithm, Skipjack, is supposed to be secret as well) when attempts are made to open the integrated circuit chips' packaging. Similarly, the iPower [58] encryption chip by National Semiconductor has tamper detection machinery which causes chemicals to be released to erase secure data. The quality of protection and the types of attacks which these system can withstand have not been published.

Smartcards are another approach to physically secure coprocessing [54]. A smartcard is a portable, super-small microcomputer. Sensing circuitry is less critical for many applications (e.g., authentication, storage of the user's cryptographic keys), since physical security is maintained by the virtue of its portability. Users carry their smartcards with them at all times and provide the necessary physical security. Authentication techniques for smartcards have been widely studied [1, 54]. Additionally, newer smartcard designs such as some GEMPlus or Mondex cards [35] feature limited physical security protection, providing a true (simple) secure coprocessor.

The technology envelope defined by these platforms and their implementation parameters constrains the limits of secure coprocessor algorithms. As the computation power and physical protection mechanisms for mobile computers and smartcards evolve, this envelope will grow.

2.4. Security Partitions

System components of networked hosts may be classified by their vulnerabilities to various attacks and placed within "native" security partitions. These natural security partitions contain system components that provide common security guarantees. Secure coprocessors add a new system component with fewer inherent vulnerabilities and create a new security partition; cryptographic techniques reduce some of these vulnerabilities and enhance security. For example, using a secure coprocessor to boot a system and ensure that the correct operating system is running provides privacy and integrity guarantees on memory not otherwise possible. Public workstations can employ secure coprocessors and cryptography to guarantee the privacy of disk storage and provide integrity checks.

Table 2.1 shows the vulnerabilities of various types of memory when no cryptographic techniques are used. Memory within a secure coprocessor is protected against physical access. With the proper protection mechanisms, data stored within a secure coprocessor can be neither read nor tampered with. A working secure coprocessor can ensure that the operating system was booted correctly (see section 3.1) and that the host RAM is protected against unauthorized logical access.[5] It is not, however, well protected against physical access — we can connect logic analyzers to the memory bus and listen passively

[5] I assume that the operating system provides protected address spaces. Paging is performed on either a remote disk via encrypted network communication (see section 4.1.3 below) or a local disk which is immune to all but physical attacks. To protect against physical attacks for the latter case, we may need to encrypt the data anyway or ensure that we can erase the paging data from the disk before shutting down.

Subsystem	Vulnerabilities	
	Availability	Integrity/Privacy
Secure Coprocessor	None	None
Host RAM	Online Physical Access	Online Physical Access
Secondary Store	Offline Physical Access	Offline Physical Access
Network (communication)	Online Remote Access	Online Remote Access Offline Analysis

Table 2.1 Subsystem Vulnerabilities Without Cryptographic Techniques

to memory traffic, or use an in-circuit emulator to replace the host processor and force the host to periodically disclose the host system's RAM contents. Furthermore, it is possible to use multi-ported memory to remotely monitor RAM. (While it may be impractical to do this in a way invisible to users, this line of attack can not be entirely ruled out.) Secondary storage may be more easily attacked than RAM since the data can be modified offline; to do this, however, an attacker must gain physical access to the disk. Network communication is completely vulnerable to online eavesdropping and offline analysis, as well as online message tampering. Since networks are used for remote communication, it is clear that these attacks may be performed remotely.

Subsystem	Vulnerabilities	
	Availability	Integrity/Privacy
Secure Coprocessor	None	None
Host RAM	Online Physical Access	Host Processor Data
Secondary Store	Offline Physical Access	None
Network (communication)	Online Remote Access	None

Table 2.2 Subsystem Vulnerabilities With Cryptographic Techniques

As table 2.2 illustrates, encryption can strengthen privacy guarantees. Data modification vulnerabilities still exist; however, tampering can be detected by using cryptographic

checksums as long as the checksum values are stored in tamper-proof memory. Note that the privacy level is a function of the subsystem component using the data. If host RAM data is processed by the host CPU, moving the data to the secure coprocessor for encryption is either useless or prohibitively expensive [29, 61] — the data must appear in plaintext form to the host CPU and is vulnerable to online attacks. However, if the host RAM data is serving as backing store for secure coprocessor data pages (see section 4.1.3), encryption is appropriate. Similarly, encrypting the secondary store via the host CPU protects that data against offline privacy loss but not online attacks, whereas encrypting that data within the secure coprocessor protects that data against online privacy attacks as well, as long as that data need not ever appear in plaintext form in the host memory.

For example, if we wish to send and read encrypted electronic mail, encryption and decryption can be performed by the host processor since the data must reside within both hosts for the sender to compose it and for the receiver to read it. But, the exchange of the encryption key used for the message should involve secure coprocessor computation: key exchange should use secrets that must remain within the secure coprocessor. [6]

2.5. Machine-User Authentication

How can we authenticate users to machines and vice versa? One solution is smartcards (see section 2.3) with zero knowledge protocols (see secton 5.1.2).

Another way to verify the presense of a secure coprocessor is to ask a third-party entity — such as a physically sealed third-party computer — to check the machine's identity for the user. This service can also be provided by normal network servers machines such as file servers. Remote services must be difficult to emulate by attackers. Users will notice the absence of these services to detect that something is amiss. This necessarily implies that these remote services must be available *before* the users authenticate to the system.

The secure coprocessor must be present for the remote services to work correctly. Evidence that these services work can be conveyed to the user through a secure display that is part of the secure coprocessor. If no such display is available, care must be taken to verify that the connection to the remote, trusted third-party server is not being simulated by an attacker. To circumvent this attack, we must be able to reboot the workstation and rely on the local secure coprocessor to perform host system integrity checks.

Unlike authentication protocols reliant on central authentication servers [81, 80, 93], this machine-user authentication happens once, at boot time or session start time. Users may be confident that the workstation contains an authentic secure coprocessor if access to *any* normal remote service can be obtained. To successfully authenticate to obtain the service, attackers must either break the authentication protocol, break the physical security

[6] This is true even if public key cryptography is used. 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.

in the secure coprocessor, or bypass the physical security around the remote server. If the remote service is sufficiently complex, attackers will not be able to emulate it.

2.6. Previous Work

The secure coprocessor system model is much more sophisticated and comprehensive than that found in previous work. It fully examines the natural security boundaries between subsystems in computers and how cryptographic techniques may be used to boost the security within these subsystems. The systems of Best [8] and Kent [46] only considered the use of encryption for copy-protection, and employed physical protection for the main CPU and primary memory. White and Comerford [104] were the first to consider the use of a security coprocessor, but their system were targeted for copy-protection and for providing cryptographic services to the host. New to the secure coprocessor model is security bootstrapping and crypto-paging, important techniques for building secure distributed systems.

Chapter 3

Applications

Because secure coprocessors can *process* secrets as well as store them, they can do much more than just keep secrets confidential. I describe how to use secure coprocessors to realize exemplar secure applications: (1) host integrity verification, (2) tamper-proof audit trails, (3) copy protection, (4) electronic currency, and (5) secure postage meters. None of these are possible on physically exposed systems. These applications are discussed briefly below.

3.1. Host Integrity Check

Trojan horse software dates back to the 1960s, if not earlier. Bogus login programs are among most common, though games and fake utilities were (and 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.

In the rest of this section, I discuss how secure coprocessors addresses this problem, discuss a few alternative solutions, and point out their drawbacks.

3.1.1. Host Integrity with Secure Coprocessors

Providing trust in the integrity of a computer's system software is not so difficult if we can trust the integrity of the execution of a single program: we can *bootstrap* our trust in the integrity of host software.[7] If we are able to run a single trusted program on the system, we can use that program to verify the integrity of the rest of the system.

Getting that first trusted program running is fraught with problems, even if we ignore management and operational difficulties, especially for machines in open clusters or unlocked offices. Running an initial trusted program becomes feasible when we add a secure coprocessor — the secure coprocessor runs only trusted, unmodified software, and this software uses cryptographic techniques to verify the integrity of the host software resident on the host's disks.

[7]Bootstrapping security with secure coprocessors is completely different from the security kernels found in the Trusted Computer Base (TCB) [101] approach: secure coprocessors use cryptographic techniques to ensure the integrity of the rest of the system, and security kernels in a TCBs simply assume that the file store returns trustworthy data.

To verify integrity, a secure coprocessor maintains a tamper-proof database (kept in secure non-volatile memory) containing a list of the host's system programs along with their *cryptographic checksums*. Cryptographic checksum functions are applied to executable file. The checksums are unforgeable: given a file F and the cryptographic checksum function $crypto_cksm()$, creating a program F' such that

$$F \neq F' \text{ and } crypto_cksm(F) = crypto_cksm(F')$$

is computationally intractable. The size of the output of a one-way hash function is small relative to the input; for example, the MD5 hash function's output is 128 bits [77].

Host integrity checking is different for the cases of stand-alone workstations and networked workstations with access to distributed services such as AFS [91] or Athena [5]. While publicly accessible stand-alone workstations have fewer avenues of attack, there are also fewer options for countering attacks. I concurrently examine both cases:

Performing the necessary integrity checks with a secure coprocessor can solve the host integrity problem. Because of privacy and integrity guarantees on secure coprocessor memory and processing, we can have confidence in results from a secure coprocessor that checks the integrity of the host's state at boot-up. If the secure coprocessor is first to gain control of the system when the system is reset, it can decide whether to allow the host CPU to boot after checking the disk-resident bootstrap program, operating system kernel, and all system utilities for tampering.

The cryptographic checksums of system images must be stored in the secure coprocessor's secure non-volatile memory and be protected against modification (and sometimes, 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 (see section 4.1.3. The secure coprocessor can detect any modifications to the system objects and can check the integrity of the external storage.

Along with integrity, secure coprocessors offer privacy; this property allows the use of both *keyed* (such as Rivest's MD5 [77], Merkle's Snefru [56], Jueneman's Message Authentication Code (MAC) [44], and IBM's Manipulation Detection Code (MDC) [41]) and *keyless* (such as chained DES [102], and Karp and Rabin's family of fingerprint functions [45]) cryptographic checksum functions. All cryptographic checksum functions require integrity protection of the cryptographic checksums; keyed checksum functions additionally require privacy protection of a key.

There are no published strong intractability arguments for major keyless cryptographic checksum functions; their design appeared to be based on *ad hoc* methods. Keyed cryptographic checksum functions require certain information to be kept secret. In the keyless case, chained DES keeps encryption keys (which select particular encryption functions) secret; Karp-Rabin fingerprint functions use a secret key to select a particular hash func-

tion from a family of hash functions based on irreducible polynomials[8] over $Z_2[x]$, i.e., $f_k \in F = \{f_i : p(x) \mapsto p(x) \bmod \text{irred}_i(x)\}$. The resulting residue polynomial is the hash result. If the key polynomial is unknown by the adversary, then given input $q(x)$, there is no procedure for finding $q'(x)$ where

$$q'(x) \neq q(x), \text{ where } f_k(q) = f_k(q')$$

except by chance. The security of Karp-Rabin is equivalent to probability of two random inputs being mapped to the same residue, which is well understood [45, 68]. Chained DES is not as well understood as the Karp-Rabin functions, since very little is known about the group structure of the permutation group induced by DES encryptions.

Secure coprocessors can keep keys secret and hence can implement keyed cryptographic checksums. The Karp-Rabin fingerprint functions are particularly attractive, since they have strong theoretical underpinnings (see section 5.2.5), they are very fast and easy to implement[9], and they may be scaled for different levels of security (by using a higher degree irreducible polynomial as the modulus).

Secure coprocessors simplify the system upgrade problem. This is important when there are large numbers of machines on a network: systems can be securely upgraded remotely through the network, since the security of communication between secure coprocessors is guaranteed. Furthermore, system images are encrypted while being transferred over the network and while resident on secondary storage. This provides us with the ability to keep proprietary code protected against most attacks. As section 3.3 notes, 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.

Section 4.1.1 examines the details of host integrity check as it relates to secure coprocessor architectural requirements, and section 4.1.5 and chapter 6 discuss how system upgrades are handled by a secure coprocessor. Also relevant is the problem of how the user can know if a secure coprocessor is properly running in a system; section 2.5 discusses this.

3.1.2. Absolute Limits

So far, we have limited the attackers to using their physical access to corrupt the software of the host computer. Is the host integrity problem insoluble if we allow trojan horse *hardware*? Clearly, sufficiently sophisticated hardware emulation can fool both users and any integrity checks. There is no completely reliable way for the secure coprocessor to detect if an attacker replaced a disk controller with a "double-entry" controller providing expected data during system integrity verification but returning trojan horse data (system programs) for execution. Similarly, it is hard to detect if the host CPU is substituted with a

[8] A polynomial is said to be irreducible if it cannot be factored into polynomials of lower degree in the ring of polynomials, in this case, $Z_2[x]$.

[9] Thus the implementation is likely to be correct.

"double-entry" CPU which fails to correctly run specific pieces of code in the OS protection system. To raise the stakes, we can have the secure coprocessor do behavior and timing checks at random intervals. This makes such "double-entry" hardware emulation difficult and forces the hardware hackers to build more perfect trojan horse hardware.

3.1.3. Previous Work

Other approaches have been tried without a physical basis for security. This section describes these approaches and their shortcomings.

Authorized Programs

The host integrity problem can be partially ameliorated by guaranteeing that all programs have been inspected and approved by a trusted authority (e.g., a local system administrator or computer vendor), but this is an incomplete solution. Guarantees about the integrity of source code are not enough [95] — we also need to trust the compilers, editors, and other tools we use to manipulate the code. Even if having the trusted authority inspect the program's object code is practical, there is no guarantee that the disassembler is not also corrupted and hiding all evidence of corruption.[10]

If the object code is built from inspected source code in a clean environment and that object code is securely installed into the workstations, we still have little reason to trust the machines. Some guarantee must be provided that the software has not been modified after installation — after all, we do not know who has had access to the machine since the trusted-software installation, and the once clean software may have been corrupted.

With computers getting smaller (and more portable) and workstations often physically accessible in public computer clusters, attackers can easily bypass *any* logical safeguards to corrupt the programs on a computer's disks. Perhaps a trojan horse program has been inserted since the last time the host was inspected — how can a user tell if the operating system kernel is correct? It is not sufficient to have central authorities that guarantee the original copy or inspect the host's software periodically. The integrity of the kernel image and system utilities stored on disk must be verified each time the computer is used.

Diskless Workstations

In the case of networked "diskless" workstations, integrity verification would appear to be confined to the trusted file servers implementing a distributed file system. Any paging to implement virtual memory would go across the network to a trusted server with disk storage [28, 79, 108].

What are the difficulties with this trusted file server model? First, non-publicly readable files must be encrypted before being transferred over the network. This implies the ability

[10]This would be similar to the techniques used by "stealth" viruses on PCs, which intercept system I/O requests and return original, unmodified data to hide the existence of the virus [23].

to use secret keys to decrypt these files, and keeping such keys secret in publicly accessible workstations is impossible.

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 digitally sign the kernel image when we boot over the network — but then we must be able to store the public keys of the trusted file servers. With exposed workstations, there is no safe place to store this type of integrity information. Attackers can always modify the file servers' public keys (and network addresses) stored on the workstation, so it contacts false servers. 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 cannot reasonably be assumed to be 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 the computer's memory — attackers can obtain the encryption keys by physical means and destroy privacy and integrity as before.

Secure Boot Media

Several researchers have argued for using a secure-boot floppy containing system integrity verification code to bring machines up. This is essentially the approach taken in Tripwire [47] and similar systems.[11] Consider the assumptions involved here.

First, we must assume the host hardware has not been compromised. If the host hardware is compromised (see section 3.1.2), the "secure" boot floppy can be ignored or even modified when it is used. (Secure coprocessors, on the other hand, cannot be bypassed, especially since users will want their machine's secure coprocessor to authenticate its identity.) Next, we must fit our boot code, integrity checking code, and cryptographic checksum database onto one or two diskettes, and this code must be reasonably fast — this is a pragmatic concern, since the integrity checking procedure needs to be easy and fast so users are willing to do it every time they start using a machine.

Secure-boot floppies are widely used on home computers for virus detection. Why isn't this approach appropriate for host integrity checking? Virus scanners and host integrity checkers have similar integrity requirements — they require a clean environment. Unlike integrity checks that detect any modifications made to files, virus scanners typically scan for occurrences of suspect code fragments within files. The fragments appearing on the list of suspect code fragments are drawn from samples observed in common viruses. It is

[11] Because Tripwire checked modifications to system files while running on the host kernel, it is vulnerable to "stealth" attacks on the kernel

presumed that these code fragments will not occur in "normal" code.[12] The integrity of the code fragment list must be protected, just like the database of cryptographic checksums. Virus scanners (and general integrity checkers) can bootstrap trust by first verifying that a core set of system programs are infection-free (unmodified), and have those programs perform faster, more advanced scanning (full integrity checks) or run-time virus detection (protection OS kernel).

Although virus scanning and host integrity checking have much in common, there are some crucial differences. Virus scanners cannot detect modifications to system software — they only detect previously identified viruses. Moreover, virus scanners' lists of suspect code fragments are independent of machines' software configurations: to update a list one adds new suspect code fragments as new viruses are identified. An integrity checker, however, must maintain an exact list of the system programs that it should check, along with their cryptographic checksums. The integrity of this list is paramount to the correct operation of the integrity checker, since attackers (including viruses) can otherwise easily corrupt the cryptographic checksum database along with the target program to hide the attack.[13] Version control becomes a headache as system software is updated.

Only trusted users are allowed access to the master boot floppy and untrusted users must get a (new) copy of the boot floppy from trusted operators each time a machine is rebooted from an unknown state. Users cannot have access to the master boot floppy since it must not be altered. Read-only floppies do not help, since we assume that there may be untrustworthy users. Careless use (i.e., reuse) of boot floppies becomes another channel of attack — boot floppies can easily be made into viral vectors.

Like diskless workstations, boot floppies cannot keep secrets. Encryption does not help, since the workstation or PC 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 computing cryptographic checksums on system images. 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.

Requiring users to obtain a fresh copy of the integrity check software and data each time they need to reboot a new machine is cumbersome. If different machines have different software releases, then each machine will have a different secure boot floppy. Management will be difficult, especially if we wish to revoke usage of programs found to be buggy by eliminating their cryptographic checksum from the database to force an update.

Furthermore, using a centralized database of all the software for all versions of that software on the various machines will be a operational nightmare. Any centralized database could become a central point of attack. Destruction of this database would disable all secure bootstraps.

---

[12]Thus, virus scanners will have false positive results, when these code fragments are found inside of a virus-free program.

[13]There are PC-based integrity checkers which append simple checksums to the executable files to deter attacks; of course, this sort of "integrity check" is easily bypassed.

Both secure coprocessors and secure boot floppies can be fooled by a sufficiently faithful system emulation using a "double-entry" disk to circumvent integrity checks (see section 3.1.2), but secure coprocessors allow us to employ more powerful integrity check techniques.

3.2. Audit Trails

Audit trails must be kept secure to perform system accounting and provide data for intrusion detection. The availability of auditing and accounting logs cannot be guaranteed (since the entire machine, including the secure coprocessor, may be destroyed). The logs, however, can be made tamper evident. This is important for detecting intrusions. Experience shows that skilled attackers will attempt to forge system logs to eliminate evidence of penetration (see [94] for an interesting case study). The privacy and integrity of the system accounting logs and audit trails can be guaranteed simply by holding them inside the secure coprocessor. However, it is awkward to have to keep all logs inside the secure coprocessor since they can grow very large and resources within the secure coprocessor are likely to be tight. Fortunately, it is also unnecessary.

To provide secure logging, the secure coprocessor *crypto-seals* the data against tampering by using a cryptographic checksum function, before storing the data on 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 original data's checksums is astronomically low. This technique reduces the secure coprocessor storage requirement to memory sufficient to store necessary 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 protect the logs. Cryptographic checksums provide the basic tamper detection and are sufficient if only integrity is needed. If accounting and auditing logs may contain sensitive information, privacy can be provided using encryption. If redundancy is also desired, techniques such as secure quorum consensus [39] and secret sharing [86] may be used to distribute the data over the network to several machines without greatly expanding the space requirements.

3.3. Copy Protection

Software is often charged on a per-CPU, per-site, or per-use basis. Software licenses usually prohibit making copies for use on unlicensed machines. This injunction against copying is technically unenforceable without a secure coprocessor. If the user can execute code on a physically accessible workstation, the user can also read that code. Even if attackers cannot read the workstation memory while it is running, we are implicitly depending on

the assumption that the workstation was booted correctly — verifying this property, as discussed above, requires the use of a secure coprocessor.

3.3.1. Copy Protection with Secure Coprocessors

Secure coprocessors can protect executables from being copied and illegally used. The proprietary code to be protected — or at least some critical portion of it — is distributed and stored in encrypted form, so copying without the code decryption key is futile,[14] and this protected code runs only inside the secure coprocessor. Either public key or private key cryptography may be used to encrypt protected software. If private key cryptography is used, key management is still handled by public key cryptography. In particular, when a user pays for the use of a program, he sends the certificate of his secure coprocessor public key to the software vendor. This certificate is digitally signed by a key management center and is *prima facie* evidence that the public key is valid. The corresponding private key is stored only within the secure non-volatile memory of the secure coprocessor; thus, only the secure coprocessor will have full access to the proprietary software. Figure 3.1 diagrams this process.

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

Section 4.1.3 discusses crypto-paging in greater detail, but the basic idea is to encrypt and decrypt virtual memory contents as they are copied between secure memory and external storage. When we run out of memory space on the coprocessor, we encrypt the data before it is flushed to unsecure external storage, maintaining privacy. Since good encryption chips are fast, we can encrypt and decrypt on the fly with little performance penalty.

Splitting the code is an alternative to crypto-paging. We can divide the code 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 concurrently on the host. 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. In μABYSS [104], White and Comerford show how such a partitioning should be done to maximize the difficulty of reverse engineering the secure portion of the application.[15]

---

[14] 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.) A more interesting question arises if the secure coprocessor may fail. Section 6.4 discusses this further.

[15] I also examined a real application, gnu-emacs 19.22 [92], to show how it could be partitioned to run partially within a secure coprocessor. The X Windows display code and the basic key-press main loop should remain within the host for performance. Most of the emacs lisp interpreter (e.g., `bytecode.c`, `callint.c`, `eval.c`, `lread.c`, `marker.c`, etc) could be moved into the secure coprocessor and accessed as remote procedures. Any manipulation of host-side data — text buffer manipulation, lisp object traversal — required during remote procedure calls can be provided by a simple read-write interface (with caching) between the

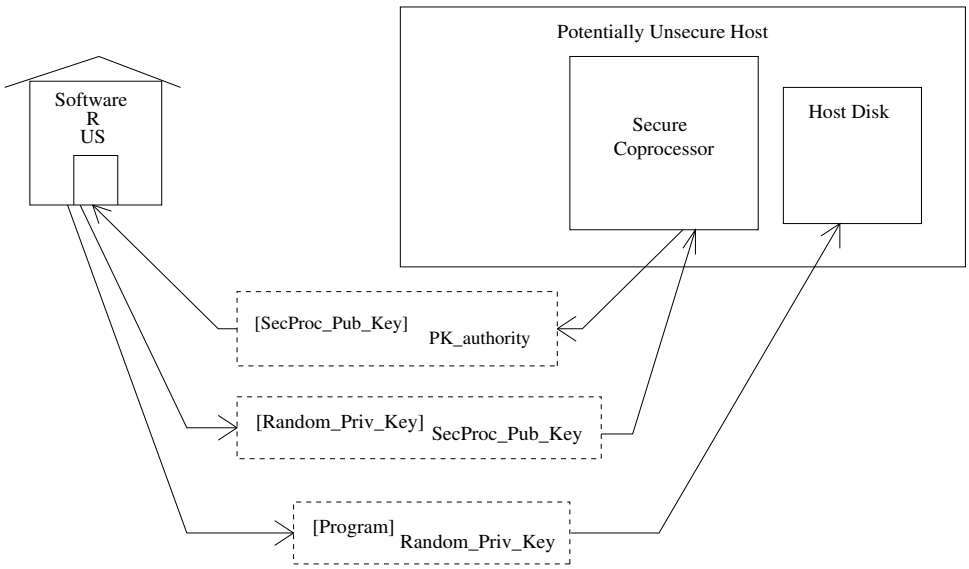

Figure 3.1 Copy-Protected Software Distribution

The software retailer encrypts the copy-protected software with a random key. This key is encrypted using the public key of the secure coprocessor within the destination host, so only the secure coprocessor may decrypt and run the copy-protected software. The software retailer knows that the public key of the secure coprocessor is good, because it is digitally signed with the public key of the secure coprocessor distributor.

Whether the proprietary code is split or not, the secure coprocessor runs a small security kernel. It provides the basic support necessary to communicate with the host or the host's I/O devices. With separate address spaces and a few communication primitives, the complexity of a security kernel can be kept low, providing greater assurance that a particular implementation is correct.

3.3.2. Previous Work

A more primitive version of the copy protection application for secure coprocessors appeared in [46, 104]; a secure-CPU approach using oblivious memory references (i.e., apparently random patterns of memory accesses) giving a poly-logarithmic slow down, appears in [29] and [61]. The secure coprocessor approach improves on these approaches by enabling the protection of large applications, permitting fault-tolerant operation (see section 6.4), and when coupled with the electronic currency application described in section 3.4, allowing novel methods of charging for use.

3.4. Electronic Currency

I have shown how to keep licensed proprietary software encrypted and allow only execute access. A natural application is to allow charging on a pay-per-use or metered basis. In addition to controlling access to the software according to the terms of a license, some mechanism must perform cost accounting, whether it tracks the number of times a program has run or tracks dollars in a user's 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 might 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.

3.4.1. Electronic Money Models

Several models can be adopted for handling electronic funds. Any implementation of these models should follow the standard transactional model, i.e., to group together operations in a *transaction* having these three properties [33, 34]:

---

coprocessor and the host, with interpreter-private data such as catch/throw frames residing entirely within the secure coprocessor. Garbage collection does become a problem, since the garbage collector must be able to determine if a Lisp object is accessible from the call stack, a portion of which is inside the coprocessor. If we chose to hide the actions of the evaluator and keep the stack within the secure coprocessor hidden, this would require that the garbage collector code (`Fgarbage_collect` and its utilities) be moved within the secure coprocessor as well.

1. *Failure atomicity*. If a transaction's work is interrupted by a failure, any partially completed results will be undone.

2. *Permanence*. If a transaction completes successfully, the result of its work will never be lost, except due to a catastrophic failure.

3. *Serializability*. Concurrent transactions may occur, but the results must be the same as if they executed serially. This means that temporary inconsistencies that occur inside a transaction are never visible to other transactions.

These transactional properties are requirements for the safe operation of any database, and they are absolutely necessary for any electronic money system.

In the following, I discuss various electronic money models, their security properties, and how they can be implemented using present day technology. (I have built an electronic currency system on top of Dyad.)

The first electronic money model is based on the cash analogy. In this mode, electronic cash has similar properties to cash:

1. Exchanges of cash can be effectively anonymous.

2. Cash cannot be created or destroyed except by national treasuries.

3. Cash transfers require no online central authority.

(Note that these properties are actually stronger than that provided by real currency — serial numbers can be recorded to trace transactions. Similarly, currency can be destroyed.)

The second electronic money model is based on the credit cards/checks analogy. Electronic funds are not transferred directly; rather, promises of payment, 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. However, by applying cryptographic techniques, anonymity can be achieved in a cashier's check-like scheme (e.g., Chaum's DigiCash model [16], which lacks transactional properties such as failure atomicity — see section 3.4.2), but the latter two requirements (conservation of cash and no online central authority) remain insurmountable. Electronic checks must be signed and validated at central authorities (banks), and checks/credit payments en route "create" temporary money. Furthermore, potential reuse of cryptographically signed checks requires that the recipient must be able to validate the check with the central authority prior to committing to a transaction.

The third electronic money model is based on the bank rendezvous analogy. This model uses a centralized authority to authenticate all transactions and is poorly suited to large distributed applications. The bank is the sole arbiter of account balance information and can implement the access controls needed to ensure privacy and integrity of the data. Electronic Funds Transfer (EFT) services use this model — there are no access restrictions on deposits into accounts, so only the person who controls the source account needs to be authenticated.

I examine these models one by one.

With electronic currency, integrity of accounting data is crucial. We can establish a secure communication channel between two secure coprocessors by using a key exchange cryptographic protocol (see section 5) and thus use cryptography to 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 [11, 22, 106] on top of the secure channel, secure coprocessors can transfer electronic funds from one account to another in a safe manner, providing privacy and ensuring that money is conserved. Most transaction protocols need stable storage for transaction logging to enable the system to roll back when a transaction aborts. On large transaction systems this typically has meant mirrored disks with uninterruptible power supplies. With the simple transactions needed for electronic currency, the per-transaction log typically is not that large, and the log can be truncated after transactions commit and further communications show all relevant parties have acknowledged the transaction. Because each secure coprocessor handles only a few users, small amounts of stable storage can satisfy logging needs. Because secure coprocessors have secure non-volatile memory, we only need to reserve some of this memory for logging. The log, accounting data, and controlling code are all protected from modification by the secure coprocessor, so account data are safe from all attacks; their only threats are 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. This is natural when one's own balances are stored on a secure coprocessor, much like the cash in one's wallets.

If the secure coprocessor has insufficient memory to hold account data for all the users, the code and accounting database may be written to host memory or disk after obtaining a cryptographic checksum (see discussion of crypto-sealing in section 4.1.3). For the accounting data, encryption may alternatively be employed since privacy is usually also desired.

Note that this type of decentralized electronic currency is *not* appropriate for smartcards unless they can be made physically secure from attacks by their owners. Smartcards are only quasi-physically secure in that their privacy guarantees stem solely from their portability. Secrets may be stored within smartcards because their users can provide the physical security necessary. Malicious users, however, can violate smartcard integrity and insert false data.[16]

Secure coprocessor mediated electronic currency transfer is analogous to rights transfer (not to be confused with rights copying) in a capability-based protection system [107].

[16]Newer smartcards such as GEMPlus or Mondex cards [35] feature limited physical security protection, though the types of attacks these cards can withstand have not been published.

Using the electronic money — e.g., spending it 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 if we assume that users can enter their passwords into a workstation without fear of their password being 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 somehow exchanged. If communications are in plaintext, attackers may simply record a transfer of a promise of payment and replay it to temporarily create cash. While security systems such as Kerberos [93], if properly implemented [6], can help to authenticate entities and create session keys, they use a centralized server and have problems similar to those in the bank rendezvous model. While we can implement the credit card/check model using secure coprocessors, the inherent weaknesses of this model keep us from taking full advantage of the security properties provided by secure coprocessors; if we use the full power of the secure coprocessor model to properly authenticate users and verify their ability to pay (perhaps by locking funds into escrow), the resulting system would be equivalent to the cash 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. Banks do not scale well to large user bases. When a bank system grows from 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 a bank's 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 cash 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 saved onto 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 balances. The security administrators who initialize the secure coprocessor software will presumably have access to the decryption keys for this purpose. Careful procedural security *must* be used here, both for protection of the decryption key and for checking 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. Fortunately, by using multiple secure coprocessors (see section 6.4), full secure fault tolerance may be achieved. The degree of redundancy and

the frequency of backups depend on the reliability guarantees desired; in reliable systems, secure coprocessors may continually run self-tests when idle and warn of impending failures (in addition to periodic maintenance checkups and replication). Section 6.3 discusses how such self-tests may be done while retaining all security properties.

The trusted electronic currency manager running in the secure coprocessor uses distributed transactions to transfer money and other electronic tokens. Transaction messages are encrypted by the secure coprocessor's basic communication layer, providing privacy and integrity of communications. Traffic analysis is beyond the scope of this work and is not addressed.

Electronic tokens are created and destroyed by a few trusted programs. For pay-per-use applications, the token is created by the vendor's sales program and destroyed by executing the application — the exact time of destruction of the token is a vendor design decision, since runs of application programs are not, in general, transactional in nature.

Because certain privileged applications may create or destroy tokens, each token type has a pair of access control lists for token creation and token destruction. These access control lists may contain zero-knowledge authentication identities [36] or application IDs: trusted applications may run on physically secure hardware (e.g., in a guarded machine room), or in a secure coprocessor. In the former case, they should have access to the corresponding zero-knowledge authenticators and should be able to establish a secure channel with other electronic currency servers to create and destroy tokens; in the latter case, the program runs (partially) in a secure coprocessor, and its program text is protected from modification.

Zero-knowledge authenticators (section 5.1.4) running in the secure coprocessor permit the use of more powerful server machines, sidestepping limits (e.g., communication bandwidth or CPU speeds) imposed on secure coprocessor design by the need for secure packaging. These server machines must be deployed within a physically secure facility and special methods must be used to ensure security [101]. Server machines installed in a secure facility, could be secure as a normal secure coprocessor; however, they need not run the secure coprocessor kernel, nor would they have access to all secret keys normally installed into a secure coprocessor.

3.4.2. Previous Work

An alternative to the secure coprocessor managed electronic currency is Chaum's DigiCash protocol [12, 16]. In such systems, anonymity is paramount, and cryptographic techniques are used to preserve the secrecy of the users' identities. No physically secure hardware is used, except in the *observers* refinement to prevent double spending of electronic money (rather than detecting it after the fact).[17]

[17]The observers model employs a physically secure hardware module to detect and prevent double spending. Chaum's protocol limits information flow to the observer, so that the user need not trust it to maintain privacy; however, it must be trusted to not destroy money. Secure coprocessors achieve the same goals with greater flexibility.

Chaum-style electronic currency schemes are characterized by two key protocols. The first is a *blind signature protocol* between a user and a central bank. During a withdrawal, the user obtains a cryptographically signed check that is probabilistically proven to contain an encoding of the user's identity. The user keeps the values used in constructing the check secret; they are used later in the spending protocol.

The second protocol is a randomized interactive protocol between a user and a merchant. The user sends the blind-signed check to the merchant and interactively proves that the check was constructed appropriately out of the secret values and reveals some, but not all, of those secrets. The merchant "deposits" to the central bank the blind-signed number and the protocol log as proof of payment. This interactive spending protocol has a flavor similar to zero-knowledge protocols in that the answers to the merchant's queries, if answered for both values of the random coin flips, reveal the user's identity. When double spending occurs, the central bank gets two logs for the same check, and from this identifies the double spender.

There are a number of problems with this approach. First, any system that provides complete anonymity is currently illegal in the United States, since any monetary transfer exceeding $10,000 must be reported to the government [19], employee payments must be reported similarly for tax purposes [18], stock transfers must be reported to the Securities and Exchange Commission, etc. Second, in a real internetworked environment, network addresses are required to establish and maintain a communication channel, barring the use of trusted anonymous forwarders — and such forwarding agents are still subject to traffic analysis. Providing real anonymity in the high level protocol is useless without taking network realities into account. Third, Chaum's cryptographic protocols do not handle failures, and any systems based on them cannot simultaneously have transactional properties and also maintain anonymity and security. A transaction abort in the blind signature protocol either leaves the user with a debited account and no electronic check or a free check. A transaction abort in the spending protocol either permits the user to falsify electronic cash if the random coin flips are reused when the transaction is reattempted (e.g., the network partition heals), or reveals identifying information to the merchant if new random coin flips are generated when the transaction is reattempted.

Clearly, to provide a realistic distributed electronic currency system, transactional properties must be provided. Unfortunately, the safety provided by transactions and the anonymity provided by cryptographic techniques appear to be inherently at odds with each other, and the tradeoffs made by Chaum-style electronic cash systems for anonymity instead of safety are inappropriate for real systems.

Another electronic money system is the Internet Billing Server [88]. This system implements the credit card model of electronic currency. A central server acts as a credit provider for users who can place a spending limit on each authorized transaction, and it provides billing services to the service providers. No anonymity is achieved: the central server has a complete record of every user's purchases and the records for the current billing period is sent to users as part of their bill. Some scaling may be achieved through replication,

but in this case providing hard credit limits require either distributed transactions, or every user must be assigned to a particular server, making the system non-fault tolerant.

Other approaches include anonymous credit cards [52] or anonymous message forwarders to protect against traffic analysis, at the cost of adding centralized servers back to the system.

3.5. Secure Postage

While cryptographic methods have long been associated with mail (dating back to the use by Julius Caesar described in his book *The Gallic Wars* [15]), they have generally been used to protect the contents of a message, or in rare cases, the address on an envelope (protecting against traffic analysis). In this section, we examine the use of cryptographic techniques to protect the *stamp* on an envelope.

The US Postal Service, with almost 40,000 autonomous post office facilities, handles an aggregate total of over 165 billion pieces of mail annually [84]. Most mail is metered or printed. (Figure 3.2 shows an example of a postage meter indicia.) Traditional postage meters must be presented to a branch post office to be loaded with postage. The postage credit is stored in a register sealed in the machine. As each letter is stamped, the amount is deducted from the machine's credit register. Postal meters are subject to at least four types of attack: (1) the postage meter recorded credit may be tampered with, allowing the user to steal postage; (2) the postage meter stamp may be forged or copied; (3) a valid postage meter may be used by an unauthorized person; and (4) a postage meter may be stolen.[18]

With modern facilities for barcoding machine readable digital information, it would be easy to replace old-fashioned human readable indicia by indicia which are either entirely or partially machine readable. These indicia could encode a digitally signed message which

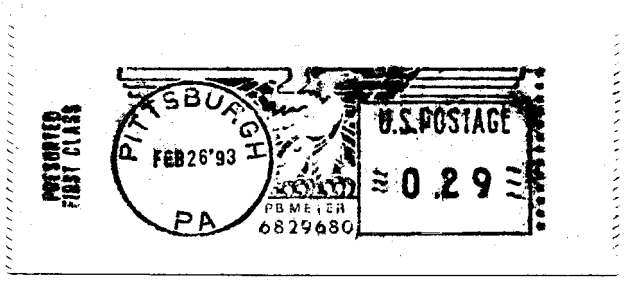

Figure 3.2 Postage Meter Indicia
Today's metered letters have a simple imprint that can be easily forged.

[18] 82,000 franking machines in the U. S. are currently reported as lost or stolen [85].

would guarantee authenticity. If this digital information included unique data about the letter (such as the date mailed, zip codes of the originator and recipient, etc.), the digitally signed stamp could protect against forged or copied stamps. A rough outline of how such a system might work was detailed by Pastor [63].

Unfortunately, a digitally signed stamp may be vulnerable to additional types of attack:

1. If cryptographic systems are misused, the system may be directly attacked.
2. Even if cryptographic techniques are used correctly, if the adversary has physical access to the postage meter, he may be able to tamper with the credit register.
3. Even if the credit is tamper-proof, a postage meter may be opened and examined to discover cryptographic keys, allowing the adversary to build new bogus postage meters.
4. The protection scheme may depend on a highly available network connecting post office facilities in a large distributed database. Since 40,000 autonomous post office facilities exist, such a network would suffer from frequent failures and partitions, creating windows of vulnerability (with 165 billion pieces of mail each year, a database to check the validity of digitally signed metered stamps appears infeasible.)

I outline a protocol for protecting electronic meter stamps, and demonstrate that the use of a secure coprocessor can address all of the above concerns. With the use of cryptography and secure coprocessors, both postage meters and their indicia can be made fully secure and tamper-proof.

3.5.1. Cryptographic Stamps

A cryptographic postage stamp is an indicia that can demonstrate to the postal authorities that postage has been paid. Unlike the usual stamps purchased at a post office, these are printed by a conventional output device, such as a laser printer, directly onto an envelope or a package. Because such printed indicia can be copied, cryptographic and procedural techniques must be employed to minimize the probability of forgery.

We use cryptography to provide a crucial property: the stamp depends on the address. A malicious user may copy a cryptographic stamp, but any attempts to *modify* it or the envelope address will be detected. To achieve this goal, we encrypt (or cryptographically checksum) as part of the stamp information relevant to the delivery of the particular piece of mail — e.g., the return address and the destination address, the postage amount, and class of mail, etc, as well as other identifying information, such as the serial number of the postage meter, a serial number for the stamp, and the date/time (a *timestamp*). The information, including the cryptographic signature or checksum, is put into a barcode. The barcode must be easily printable by commodity or after-market laser printers, it must be easily scanned and re-digitized at a post office, and it must have sufficient information density to encode all the bits of the stamp on the envelope within a reasonable amount of

space. Appropriate technologies include Code49 [62], Code16K [43], and PDF417 [42, 65, 66]. Symbol Technologies' PDF417, in particular, is capable of encoding at a density of 400 bytes per square inch, which is sufficient for the size of cryptographic stamps needed to provide the necessary security in the foreseeable future. Figure 3.3 shows the amount of information that can be encoded.

Six lines of 40 full ASCII characters for each address, four bytes each for hierarchical authorization number, the postage meter serial number, the stamp sequence number, the postage/class, and the time, totals to under 500 bytes of data. (Using PDF417, 500 bytes takes 1.24 square inches.)

The cryptographic signature within the indicia prevents many forms of replay attacks. Malicious users will not find it useful to copy the stamps, since the cryptographic signature prevents them from modifying the stamp to change the destination addresses, etc, so the copied stamps may only be used to send more mail to the same destination address. If duplicate detection is used (see below) then even this threat vanishes. The timestamps and serial numbers also limit the scope of the attack by restricting the lifetime of copies and permitting law enforcement to trace the source of the attack.

Because cryptographic stamps also includes source information, the postage meter serial number, and the return address, duplicated stamps can also be detected in a distributed manner. Replays are detected by logging recent, unexpired indicia from processed mail. If the post office finds a piece of mail with a duplicate stamp, they will know that some form of forgery has occurred. We will examine the practicality of replay detection later in section 3.5.2.

While databases at regional offices can deter replay attacks, we need some way to protect the cryptographic keys within the postage meters as well — attackers who gain

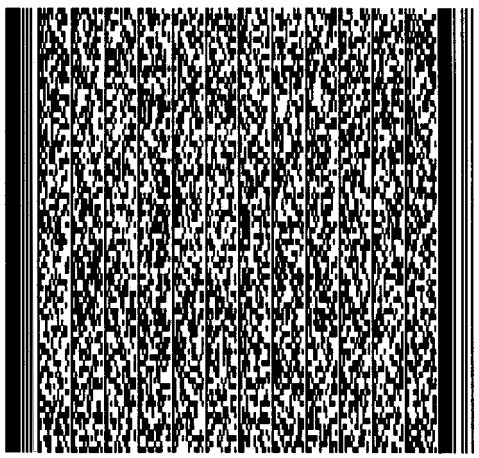

Figure 3.3 PDF417 encoding of Abraham Lincoln's Gettysburg Address

access to the keys can use them to fraudulently sign cryptographic stamps. To prevent malicious users from accessing cryptographic keys requires physically protected memory and secure processing of the cryptographic keys. (If a machine does not perform secret computations using cryptographic keys, an adversary can place logic analyzer probes to observe address/data buses and obtain key values. Alternatively, the adversary may replace the memory subsystem with dual ported memory, and just read the keys as they are used.) Even password protected, physically secure memory (such as that that provided by some dongles used with PC software) is insufficient — the software must contain the passwords required to access that protected memory, and if attackers don't know how to disassemble the software to obtain the passwords, they can read it off of the wires of the parallel port as the software sends the passwords to enable access.

Private processing of cryptographic keys is a necessary condition for cryptography. Not only is this a necessary requirement to run real cryptographic protocols, it is also a necessary requirement for keeping track of the credit amount remaining in a electronic postage meter register. Protected computation is also required to establish secure channels of communication for remote (telephone or network) credit update — the electronic postage meter must communicate with the post office when the user buys more postage, and cryptographic protocols must be run over the communication lines to prevent foul play. Secure communication channels require cryptography, and we need a safe place to keep cryptographic keys and to perform secure computation.

To achieve private, tamper-proof computation, a processor with secure non-volatile memory for key storage, and perhaps some normal RAM as scratch space (to hold intermediates in the calculations) must also be made physically secure. These properties are easily provided by secure coprocessors.

3.5.2. Software Postage Meters

By using secure coprocessors in a PC-based system, we can build secure postage meter software. A PC-based electronic postage meter system would include a secure coprocessor, a PC (the coprocessor host), a laser printer, a modem, and optionally an optical character recognition (OCR) scanner and/or a network interface. Like ordinary postage meters, our PC-based postage meter system would operate in an office environment as a shared resource, much like laser printers.

The basic idea is simple: the software obtains the destination and return addresses and the weight and delivery class from the user — either directly from the word processor running on the user's PC[19], by reading directly from the envelope and using OCR software, or by direct keyboard input — and requests a cryptographic stamp from the secure coprocessor. The secure coprocessor decrements its credit register, and generates a digitally signed message containing the value of the stamp, all of the addressing information, the

[19]The word processing software can even provide good weight estimates since it knows the number of pages in the letter.

date, the ID of the secure coprocessor, and other serial numbers. This message (a bit vector) is sent to the PC, which encodes it and prints a machine readable indicia on the laser printer Advanced 2-D bar coding technology such as PDF417 mentioned in section 3.5.1 may be employed.

Postage Meter Currency Model

Postage credits held within an electronic postage meter are simpler than general electronic currency because of their restricted usage. Postage credits must be purchased from a post office, and credits can only buy one type of item: cryptographic stamps (or be transferred to another electronic postage meter).

We can take advantage of these restrictions to the currency model to achieve solutions simpler than those considered in section 3.4. Furthermore, because pieces of mail produced by a particular secure coprocessor are likely to be mailed in the same locality, the replay detection can be done with much lower overhead than otherwise, as described below.

Reloading a Meter

Only post offices may reload postage meters. Unlike their older mechanical brethren, electronic postage meter equipment need not be carried to the local post office when the amount of credit inside runs low — the local post office can simply provide a phone number to "recharge" electronic postage meters by modem, paying by credit card numbers or direct electronic funds transfer. The USPS meter authenticates the secure coprocessor and uploads funds. Meters' communications must be protected by cryptography; otherwise a malicious user may record the control signals used to update credit balances and replay that message. Encryption also protect businesses' credit card or EFT account numbers from being used by malicious eavesdroppers.

Detecting Replays

With a kilobyte of data per stamp, it would seem at first that replay detection is infeasible because of size of the database required. However, we can exploit the distributed nature of mail delivery and sorting.

The US Postal Service sorts mail twice. First, mail is sorted by destination zip code at a site near the source. Then, the mail is delivered (in large batches) to a site associated with the destination zip code, where the mail is again sorted, this time by carrier route. Every piece of mail destined for the same address passes through the same secondary sorting site, making it a natural place for detecting replays.

Detecting replays locally is feasible with today's technology. Using the 1992 figures of 165 billion pieces of mail per year handled at 600 regional sorting sites, with the simplifying assumption that the volume of mail is evenly distributed among these regional offices, we can obtain an estimate of the storage resources required. Assuming that cryptographic

stamps expire six months after printing,[20] an average regional office will see approximately 130,000,000 stamps out of a national total of 80,000,000,000 stamps. If we store one kilobyte of information per stamp (doubling the above estimate) and assume that the entire current mail volume uses cryptographic stamps, this would require only 130 gigabytes of disk storage per facility for logging, well within the capacity of a single disk array system. The stamps database can be viewed as a sparse boolean matrix indexed in one dimension by postage meter serial number and in the second dimension by stamp sequence number for that postage meter. Hashing this matrix into a 256 megabyte hashtable results in a 6% chance of collision.

To make replay detection even easier, we exploit the physical locality property: pieces of mail stamped by a single postage meter are likely to enter the mail processing system at the same primary sorting site. Therefore, cryptographic stamps from the same postage meter are very likely to be canceled at the same regional office, and we can detect replays there. If any cryptographically stamped piece of mail is sent from a different mail cancellation site, network connections can be used for real-time remote access of cancellation databases, or batch processing media such as computer tapes may be used. In the case of real-time cancellation, the network bandwidth required depends on the probability of the occurrence of such multi-cancellation-site processing, and on how quickly we need to detect replays. The canceled stamps database at each regional office need not be large — each postage meter can simply write a counter value in its stamps. We need only fast access to a bit vector of recently used, unexpired stamp counter values. These bit vectors are indexed by the postage meter's serial number and can be compressed by run-length encoding or other techniques. Only when a replay is detected might we need access to the full routing information.

The average figure of 130,000,000 stamps tracked by a regional office can now be represented as a dense bit vector, since only local postage meters need to be tracked. A fast bit-vector representation would require 1300 megabits of storage plus indexing overheads, or just 17 megabytes plus overhead — an amount of storage that can easily fit into an average PC. While additional space may be required for indexing to improve throughput and for replicated stable storage, the amount of memory required is quite small.

[20]The U. S. Postal Service claims to deliver more than 90% of all first class mail in three days, and more than 99% in seven days. Six months would appear to be a generous bound for mail delivery.

Chapter 4

System Architecture

I have implemented Dyad, a prototype secure coprocessor system. The Dyad architecture is based on operational requirements arising from the security applications in chapter 3. However, the hardware modules on which Dyad is built present additional limitations on the actual implementation. This chapter starts off with Dyad's abstract system architecture based on the operational requirements of a security system during system initialization and during normal, steady state operation. Next, I detail the capabilities of our hardware platform, and describe the architecture of the actual implementation.

4.1. Abstract System Architecture

Chapter 3's security applications place requirements and constraints on system structure. From these application requirements I arrive at an operational view of how secure coprocessor systems should be organized.

4.1.1. Operational Requirements

I begin by examining how a secure coprocessor interacts with the host during system boot and then proceed with a description of system services that a secure coprocessor provide to the host operating system and user software.

To be sure that a system is securely booted, the bootstrap process must involve secure hardware. Depending on the host hardware (e.g., whether a secure coprocessor could halt the boot process in case of an anomaly) we may need secure boot ROM. Either the system's address space is configured so the secure coprocessor provides the boot vector and the boot code directly; or the boot ROM is a piece of secure hardware. In either case, a secure coprocessor verifies system software (operating system kernel, system related user-level software) by checking the softwares' signatures against known values. To check that the version of the software present in external, unsecure, 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 unsecure network: if an attacker can completely emulate the interaction that the secure coprocessor has with a normal host system, it is impossible for the secure coprocessor to detect this. With secure coprocessor/host interaction, we can make very few assumptions about the host (it can not keep cryptographic keys). The best that we can do is to assume that the cost of completely emulating the host at boot

time is prohibitively expensive. (Section 3.1.2 discusses the theoretical limitations to this approach.)

The secure coprocessor ensures that the system securely boots; after booting, a secure coprocessor aids the host operating system by providing security functions. A secure coprocessor does not enforce the host system's security policy — this is the job of the host operating system. Since we know from the secure boot procedure that a correct operating system is running, we may rely on the host to enforce policy. When the host system is up and running, a secure coprocessor provides various security services to the host operating system:

- integrity verification of any stored data (by secure checksums);
- data encryption to boost storage media natural security (see section 2.4); and
- encrypted communication channels (key exchange, authentication, private key encryption, etc).[21]

4.1.2. Secure Coprocessor Architecture

The boot procedure described above made assumptions about secure coprocessor capabilities. Let us refine the requirements for secure coprocessor software and hardware.

To verify that the system software is the correct version, the secure coprocessor must have secure memory to store checksums or other data. If keyless cryptography checksums such as MD5 [77], multi-round Snefru [56], or IBM's MDC [41] are one-way hash functions, then the only requirement is that the memory be protected from unauthorized writes. Otherwise, we must use keyed cryptographic checksums such as Karp and Rabin's technique of *fingerprinting* (see [45] and section 5.1.5). The latter approach requires that memory also be protected against read access, since both the hash value and the key must be secret. Similarly, cryptographic operations such as authentication, key exchange, and secret key encryption all require secrets to be kept. Thus a secure coprocessor must have memory inaccessible by all entities except the secure coprocessor itself — enough private non-volatile memory to store the secrets, plus private (possibly volatile) memory for intermediate calculations in running protocols.

How much private non-volatile and volatile scratch memory is enough? How fast must the secure coprocessor be to have good performance with cryptographic algorithms? 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.

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

4.1.3. Crypto-paging and Sealing

Crypto-paging is another technique for trading off memory for speed. A secure coprocessor encrypts its virtual memory contents before paging it out to the host's physical memory (and perhaps eventually to an external disk), ensuring privacy. 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. To ensure integrity, virtual memory contents may be *crypto-sealed* by computing cryptographic checksums prior to paging out and verifying them when paging in.

Crypto-paging and sealing are analogous to paging of physical pages to virtual memory on disk, except for different cost coefficients. Well-known analysis techniques can be used to tune such a system [49, 108]. The cost variance will likely lead to new tradeoffs: computing cryptographic checksums is faster to calculate than encryption, 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.

Crypto-paging is a special case of a more general speed/memory trade off for secure coprocessors. I observed in [97, 98] that Karp-Rabin fingerprinting can be sped up by about 25% on an IBM RT/APC with a 256-fold table-size increase; when implemented in assembler on an i386SX the speedup is greater (about 80%; see chapter 8). Intermediate-size tables yield intermediate speedups at a slightly higher increase in code size. Similar tradeoffs can be found for software implementations of DES.

4.1.4. Secure Coprocessor Software

A small, simple security kernel is needed for the secure coprocessor. What makes Dyad's 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 isolate the effects of programming errors. Unlike the host's kernel, many services are not required: terminal, network, disk, and most 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 be forwarded to the host kernel for processing. Normal operating system daemons such as printer service, electronic mail, etc. are entirely inappropriate in a secure coprocessor.

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. *Resource management* includes task allocation and scheduling, virtual memory allocation and paging, and allocation of communication ports. *Communications* 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.

Key management includes management of authentication secrets, cryptographic keys, and system fingerprints of executables and data. With the limited number of services needed, we can easily envision using a microkernel such as Mach 3.0 [31], the NT executive [20], or QNX [40]. We only need to add a communications server and include a key management service to manage secure non-volatile key memory. If the kernel is small, we have more confidence that it can be debugged and verified. (In Dyad, we ported Mach 3.0 to run within the Citadel secure coprocessor.)

4.1.5. Key Management

Key management is a core portion of the secure coprocessor software. Authentication, key management, fingerprints, and encryption protect the integrity of the secure coprocessor software and the secrecy of private data. The bootstrap loader, in ROM or in secure non-volatile memory, controls the bootstrap process of the secure coprocessor itself. In the same way that the host-side bootstrapping process verifies the host-side kernel and system software, this loader verifies the secure coprocessor kernel before transferring control to it.

The system fingerprints needed for checking system integrity reside entirely in secure non-volatile memory or are protected by encryption while in external storage. (Decryption keys reside solely in secure non-volatile memory.) If the latter approach is chosen, new private keys must be selected for every new release of system software[22] to prevent replay attacks where old, buggy, secure coprocessor software is reintroduced into the system. Depending on the algorithm, storage of the fingerprint information requires only integrity or both integrity and secrecy. For keyless cryptographic checksums (MD4, MDC, and Snefru), integrity is sufficient; for keyed cryptographic checksums (Karp-Rabin fingerprint), both integrity and secrecy are required.

Other protected data held in secure non-volatile memory include administrative authentication information needed to update the secure coprocessor software. We assume that a security administrator is authorized to upgrade secure coprocessor software. The authentication data for the administrator 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 [26, 33, 34]. Non-volatile memory gives us permanence automatically; serializability, while important for multi-threaded applications, can be enforced by permitting only a single upgrade operation at a time (this is an infrequent operation and does not require concurrency); and the failure atomicity guarantee can be provided as long as the secure non-volatile memory subsystem provides an atomic store operation. Update transactions need not be distributed nor nested; this simplifies the implementation.

[22] One way is to use a cryptographically secure pseudo-random number generator [9, 10] with its internal state entirely in secure non-volatile memory.

4.2. Concrete System Architecture

My Dyad prototype secure coprocessor system is realized from several system components. To a large extent, it satisfies the system hardware requirements induced by the abstract architecture discussed in the previous section. At the highest level, the Dyad prototype is a host workstation with special modifications that allows it to talk to a secure coprocessor, and the secure coprocessor itself. The prototype host system hardware is a IBM PS/2 Model 80. The prototype secure coprocessor subsystem is a Citadel coprocessor board [105]. The secure coprocessor is attached to the PS/2's microchannel system bus via a Data Translation adapter card. The interfaces between these hardware components and limitations of these components influence or constrain some aspects of the system software architecture.

Both hardware subsytems run the CMU Mach 3.0 microkernel [31]: the host has special device drivers to support communication with the coprocessor through the Data Translation card, and the coprocessor kernel has special drivers and platform-specific assembly language interface code in addition to the machine independent code. On the host side there is additional software for providing interface support for the secure coprocessor.

The remainder of this section describes the hardware, the host-side system software, the coprocessor-side system code, and the application interface.

4.2.1. System Hardware

The PS/2 host contains an Intel i386 CPU running at 16 MHz, 16 megabytes of RAM, and a microchannel system bus. The Citadel coprocessor contains an Intel i386SX CPU, also running at 16 MHz; one megabyte of "scratch" volatile RAM; 64 kilobytes of battery-backed secure RAM; bootstrap EPROM with a simple monitor program; and 64 kilobytes of second-stage bootstrap EEPROM; an IBM produced DES chip with a theoretical throughput of 30 megabytes per second.[23] All of this is privacy-protected by intrusion detection hardware.

Because the Citadel coprocessor is prototype hardware, it has not been integrated into a standard microchannel card. Instead, the coprocessor board is physically external to the host and is logically attached to the host's system bus via a Data Translation microchannel interface card within the host. (See figure 4.1.) The Data Translation card contains the bus drivers, microchannel protocol chips, and bidirectional "command" port I/O, plus some simple logic for generating host-side interrupts. The microchannel chips handle arbitration for two independent DMA channels which simultaneous input and output to the DES engine.

[23]The coprocessor board's design limits the maximum throughput to 16 Mbyte/sec — an external hardware state machine controls the DES chip's operation, and a separate 32 MHz crystal independently clocks this state machine. If the control software used zero time, this 16 Mbyte/sec figure would represent the maximum attainable encryption throughput for the Citadel board.

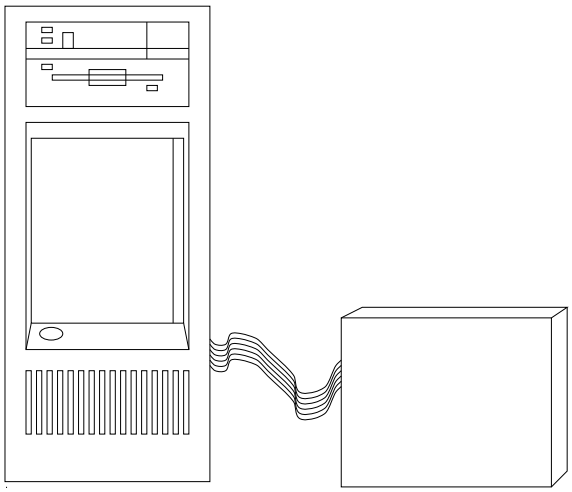

Figure 4.1 Dyad Prototype Hardware

DES Engine

The DES engine on the coprocessor board includes input and output FIFO buffer chips for the DES chip I/O. Because the DES chip runs on a separate clock, these FIFOs permit fast, asynchronous data transfer with hardware interlocks. The data source and sink for the FIFOs may be programmed via multiplexors to be one of six sources/sinks: the host (via DMA transfers), the coprocessor, and an external bus interface. The external bus interface is unused in the present configuration; in the future, it may be connected to network interfaces or disk controllers. Furthermore, the DES engine can be configured to work in "cipher bypass" (CBP) mode, where data is routed around the DES chip. This permits the use of the DMA channels to transfer bulk data between the coprocessor and the host without encryption. Figure 4.2 shows the DES engine data paths.

The Citadel DES engine's I/O multiplexors and the DES chip's encryption/decryption mode are configured via a control port accessible on the coprocessor bus. When the host is the data source, the DES engine expects that the host has configured its DMA channel to transfer data to the input FIFOs, and when the coprocessor system bus is the data source, the coprocessor itself will write to the input FIFO via processor I/O instructions. Similarly, when the host is the data sink, the DES engine expects the host will DMA-transfer data from the output FIFOs to its memory; when the coprocessor is the sink, processor I/O instructions are used to read out data from the output FIFO.

The Data Translation card provides two 16-bit wide DMA channels, giving simultaneous access to the input and output ends of the hardware DES engine, thus allowing the host to request host-memory to host-memory DES encryption/decryption operations. This form of "filter" I/O operation does not fit the usual Unix/Mach style read/write model; we will see below that the driver software handles this as a special case.

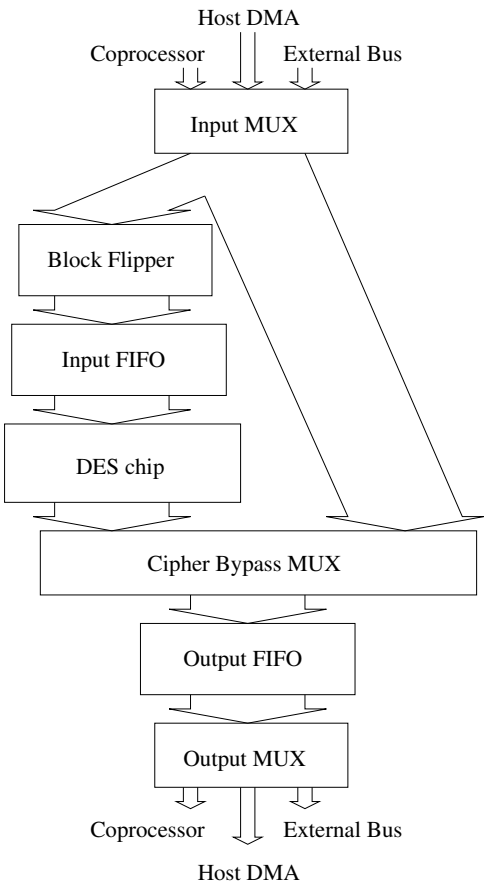

Figure 4.2 DES Engine Data Paths

The DES engine runs asynchronously; the input and output FIFOs allow the data sources and sinks to move data quickly without needing to poll or spend too much time processing interrupts. The cipher-bypass multiplexor allows the use of the buffers and DMA control logic circuits without engaging the DES chip, allowing dual usage of the DMA hardware.

Command Ports

The Citadel-side Dyad kernel uses the DES data path for bulk communication as well as encryption; for lower bandwidth communication and controlling the DES engine data path, the kernel uses bidirectional command ports provided by the Data Translation card.

The command ports are 16 bits wide and show up in the I/O address space of both the host and the coprocessor; status bits in a separate status register show whether the current word is unread, and interrupts may be generated as a result of certain state transitions of the command ports. On the host side, individually maskable interrupts may be generated whenever a new value is written to the host from the coprocessor, telling the host to read that value; or whenever the previous value from the host to the coprocessor was read by the coprocessor, telling the host that it may write the next value. Unfortunately, on the coprocessor side only one interrupt for command port I/O exists — the coprocessor receives a (maskable) interrupt when a new value arrives on the port, but is not informed when it may send the next value to the host. As we will see in section 4.2.3, this causes some problems with performance in the Dyad kernel implementation.

Hardware Limitations

There are several Citadel hardware design limitations which degrade system performance. Because the command port does not generate an interrupt when data may be sent from the Citadel to the host, the command port throughput is lower than it would be otherwise. The coprocessor kernel software polls a status register occasionally to send data to the host, and this polling frequency limits the bandwidth. Furthermore, the use of the command port is used to send control messages for setting up the DES/DMA data path for high speed transfers. This adds extra latency to these transfers.

The data path between the IBM designed DES chip and the I/O FIFOs are 16-bit-wide words. Unfortunately, because of a design error in the DES chip, the 16-bit units within a block of ciphertext/cleartext must be provided to the chip in reverse order. To work around this in hardware, extra "byte flipper" latches are included to reverse words within blocks. There is very little penalty in terms of throughput; however, it does add extra latency to every transfer, since 6 extra bytes of data must be written to the input FIFOs in order to flush out the previous block of data. This cannot always be done by extending the sizes of the transfers, since overflowing input buffers in an DMA transfer causes system faults when transferring at the end of physical memory. For DMA transfers, since the DMA controller needs to be reset/released at the end of the transfer in any case, the extra 6 bytes are written via software at the DMA completion interrupt.

The DMA completion interrupt occurs when the DMA controller transfers the requested number of words. Unfortunately, the DMA controller cannot always be reset at this time, since for host-to-coprocessor transfers this means only that the input FIFO to the DES engine is full, and resetting the DMA controller at this point would confuse the DES engine. Similarly, the coprocessor must initialize its DES engine before the host can program the

DMA controller. Both the DES-engine-completion event and the DES-engine-initialization event cause a status bit to change, and the host must poll the status register to detect this.

Another design flaw in the IBM DES chip causes alternate decryptions to output garbage. The driver software compensates by performing a dummy decryption of a small, internal buffer after every normal decryption. This imposes extra latency overhead. Some of the overhead of the dummy decryption is hidden from host-side applications by performing it *after* the real decryption, since the host-side DMA transfer for the real decryption will complete by this point and the dummy decryption may overlap with host-side driver execution (releasing DMA channel etc).

Yet another limitation is not a design flaw *per se*: the Data Translation card interface does not provide the coprocessor with the ability to become a "bus master" on the host's system bus — i.e., the coprocessor may not take over the microchannel, driving the address lines and read/write memory. Furthermore, the system bus interface provided by this card does not provide ABIOS device boot ROM space, which contains code that the host processor runs at host boot-up. Because the coprocessor cannot control the host system to perform host integrity checks, this prohibits the prototype system's coprocessor from performing secure bootstrap of the host and from periodically checking the behavioral of the host system. This should be repaired in a revised version of the board.

These hardware idiosyncrasies force some extra complexity in coprocessor kernel software, and make it impossible (currently) to implement secure bootstrapping of the host. Fortunately, most of this extra complexity only imposes a slight overall performance degradation in the system software, though the DMA transfer rates are much lower than they could be otherwise.

4.2.2. Host Kernel

The system software on the host contains only one Dyad-specific module: the driver needed to use the Data Translation card to talk to the Citadel board. The host kernel driver is separated into two parts: two low-level drivers in the Mach microkernel and a higher-level driver in the Unix server. The low-level drivers handle interrupts and simple device data transfers to the command port and the DMA channels; the high-level driver provides an integrated view of the coprocessor as a Unix device, emulating an older driver that I wrote for the Mach 2.5 integrated kernel. Figure 4.3 shows the structure of the host-side system.

Microkernel Drivers

The microkernel drivers handle low-level data transfer, with separate drivers for the command port and DMA. The command port I/O is viewed as a serial device, since every 16-bit word being transferred is usually accompanied by an interrupt. The DMA I/O is viewed as a block transfer device, since large chunks of data are transferred at a time.

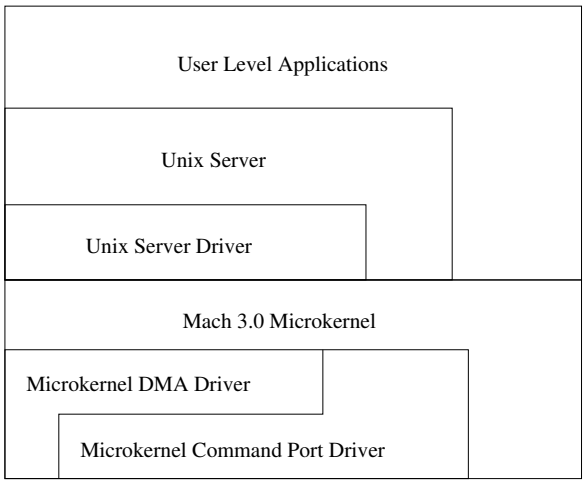

Figure 4.3 Host Software Architecture

Command Port The two microkernel drivers are not independent — the command port driver provides hooks for the DMA driver's internal use; the command port is used by the DMA driver to synchronize DMA transfers with the coprocessor. When the DMA driver needs to use the command port, it checks that all other pending data queued for the coprocessor has been sent prior to taking it over. If new I/O requests arrive while the command port driver is being used by the DMA driver, they are enqueued separately in the command port driver until the DMA driver is done, so that any messages sent by the DMA driver will not be interrupted.

Because of the interactions between the DMA driver and the command port driver, command port device operations must guarantee that a successful return from the device remote procedure call really means that the data was transferred. Unlike serial line drivers currently in Mach 3.0, my port I/O driver does not simply enqueue data from `device_write()` requests into a circular buffer, return immediately with `D_SUCCESS`, and send the data later; rather, it keeps the write requests in queues and generates a reply message only after the data has been actually sent to the coprocessor. Similarly, data from the coprocessor are read only if there have been `device_read()` requests have been enqueued, and when the DMA driver takes over the command port, the DMA driver may jump this read queue to obtain replies. Typically, the Unix server will have only one `device_read()` request pending at any given time for the command port.

DMA driver The microkernel DMA driver translates device I/O requests into DMA transfers to or from the coprocessor DES engine. The DES hardware is integral to every DMA transfer, and must be programmed by the coprocessor with the appropriate transfer count, DES chip mode, and data source or sink. Prior to a transfer, the DMA driver uses the

command port to inform the coprocessor of the size and type of the transfer. The encryption key and initial vector used for DES operations are also set by the coprocessor, and it is assumed that the host processor has arranged with the coprocessor to use the appropriate key and initialization vector.

Associated with the driver is state which is set by calling `device_set_status()`. This state determines whether the driver should be operating in "filter" mode and whether the driver expects that the DES engine within the coprocessor will perform an encryption DMA transfer or a decryption transfer. This driver state must be consistent with the state of the coprocessor kernel.

The type of DMA transfer with the DES engine depends on whether or not a DMA driver read/write operation is in filter mode. The DES engine's I/O FIFOs may be configured to both source and sink data via the DMA transfers. In non-filter mode the driver simply translates `device_write()` operations into DMA transfers from the supplied data buffer to the DES engine's input FIFO, with the coprocessor bus as the data sink. Similarly, `device_read()` operations are translated into DMA transfers from the DES engine's output FIFO, with the coprocessor writing to the DES engine's input.

When the DMA driver is in filter mode, `device_write()` and `device_read()` operations must come in identically-sized pairs. The DMA driver assumes that the coprocessor will program the DES engine's input and output FIFOs to use DMA transfers to or from the host, and on a `device_write()` operation the driver will internally allocate an I/O buffer to hold the filter result. These I/O buffers are then enqueued in the driver, with the data being transferred into the result of the matching `device_read()` operation when it comes along.

Whether a DMA transfer results in an encryption or a decryption by the DES engine is important to the DMA driver because the DES engine performs encryption/decryption in Ciphertext Block Chaining (CBC) mode [57]: the previous block of ciphertext is fed back into the DES chip as part of the encryption/decryption operation. Since out-of-line data in Mach IPC messages are remapped pages of virtual memory, the DMA driver has no control over their location in the physical address space, and these pages are likely to be physically non-contiguous. Because DMA transfers operate on *wired-down* physical buffers (virtual memory marked as nonpageable), the DMA driver typically cannot DMA-transfer more than one physical page at a time. This implies that the driver must check, on a new transfer, that the last block of ciphertext from the previous transfer is available to the coprocessor to maintain the CBC feedback.

While there are three transfer modes (encrypt, decrypt, crypto-bypass) and multiple data sources and sinks (host or coprocessor), only some of these require special action. These are the cases where the host DMA driver has possession of the last block of cipher text (1) when the operation is a non-filter `device_read()` and the DES mode is encryption, (2) when the operation is a non-filter `device_write()` and the DES mode is decryption, and (3) the operation is an filter (paired `device_write()` and `device_read()`) and the DES mode is either encryption or decryption.

At first glance it may appear that it would be possible to simplify the problem of non-physically-contiguous pages by running multiple DMA transfers to the DES engine without informing the DES engine that the encryption (or decryption) transfer will be performed in pieces. Unfortunately, this doesn't work, since the DMA controller on the host relies on the peripheral to generate a DMA completion interrupt, and the Citadel interface generates DMA completion based on the DES engine's count register reaching zero. If we tried to program the DES engine with the full size of the I/O request but performed smaller, partial DMA transfers, the host would not know when a DMA transfer is completed except by polling the host-side DMA controller's transfer count register.

Because user data buffers must be partitioned into DMA transfers of physically contiguous pages, the DMA driver also sends a DMA start control message (via the command port) to the coprocessor kernel specifying the size of each of the current transfer. This control message is sent after the DMA driver has initialized the host-side DMA controller; when the coprocessor receives a DMA start message, it then initializes the DES engine, which causes the DMA transfer to proceed.

Unix Server Driver

Like most Unix server drivers, the Unix server driver for communicating with the secure coprocessor is simpler than the microkernel driver. The Unix server driver for coprocessor communication provides a more integrated view of the coprocessor than do the microkernel-level drivers. It achieves this by using both of the underlying Mach devices via standard Mach device remote procedure call primitives.

Unix level system calls `open()`, `close()`, `read()`, `write()`, and `ioctl()` are translated by the Unix server driver into equivalent Mach driver `device_open()`, `device_close()`,`device_read()`,`device_write()`,`device_set_status()`, and `device_get_status()` remote procedure calls. The `read()` and `write()` system calls are translated into `device_read()` and `device_write()` to the DMA driver in the microkernel for bulk data transfer. The Unix server driver provide special `ioctl()` requests to send short control messages via the command port; these are translated to the appropriate device messages to the Mach-level coprocessor command port driver. These control messages are used to negotiate the type and contents of bulk DMA data transfers, for low level control operations with the EPROM boot monitor, and for emulation of a console device for the coprocessor kernel.

Coprocessor Interface

I wrote a user-level program, `cit`, to run on the host to download the coprocessor microkernel into the coprocessor, provide a simple emulated console display, and provide mass storage access for the coprocessor kernel once it boots up. The program uses the Unix driver to communicate via the command port to the EPROM-resident monitor, and thus perform simple diagnostics and download code.

The `cit` program also provides the functionality of the display half of a console driver — it maintains and updates a memory map of the console display contents and passes keyboard input through the command port to the microkernel running on the coprocessor. At the same time, it uses the DMA driver to provide a simulated disk drive to the coprocessor microkernel, with DMA control I/O being multiplexed with the console I/O (and the lower level, automatic DMA control I/O) multiplexed over the command port I/O channel.

Any host-side user-level process that wishes to use the coprocessor's DES engine must request those services using interprocess communication with `cit`. In turn, `cit` will make the appropriate requests (via the command port) to the coprocessor kernel to configure the DES engine appropriately.

4.2.3. Coprocessor Kernel

The coprocessor runs a Mach 3.0 microkernel that is downloaded by `cit`. The basic Mach 3.0 kernel for the AT-bus i386 required significant changes to its low-level interface code, in addition to new device drivers. This section outlines my changes.

Low Level Machine Interface

When Mach 3.0 is loaded into the Citadel coprocessor, the initial environment provided to it by the bootstrap loader differs from that provided in standard i386 AT-bus systems. In standard PC systems, the second level bootstrap loader switches the processor to 32-bit code and data segments before transferring control to the kernel's entry point, `pstart`,[24] in addition to setting machine specific parameters such as conventional/extended memory sizes. The coprocessor PROM monitor downloading the kernel runs with 16-bit data and code segments, so I had to add new assembler-language code to switch the processor from 16-bit code/data segments to 32-bit segments at the kernel's startup.

A more important difference in the low-level environment is that the interrupt subsystem is completely different — the Citadel coprocessor does not include a peripheral interrupt controller, and interrupt priorities are hard-wired into the system inside a programmable logic device. Interrupts may be individually masked. The system provides seven interrupts:

1. clock (1 kHz),
2. command port input available,
3. DES done,
4. DES input FIFO full,
5. DES output FIFO empty,
6. DES input FIFO not full,

[24] The kernel expects to be using physical addresses at this point, thus the name.

7. DES output FIFO not empty.

Note that there is no interrupt to indicate that the command port is writable (i.e., the previous data element has been read by the host). The coprocessor kernel must poll a status port to send data to the host; this testing is done at every interrupt (in `interrupt.s`), the maximum additional latency is one millisecond per transferred character.

The DES input FIFO full and the DES output FIFO empty interrupts were intended to allow high throughput coprocessor encryption: a thread could write into the DES input FIFO at very high speeds, and switch to reading from the DES output FIFO when an input FIFO full interrupt occurs; similarly, when an output FIFO empty interrupt occurs, the thread may switch back to writing to the DES input FIFO.

Console

The microkernel console driver multiplexes its I/O with I/O from the DMA driver and a serial-line-style communications driver `com` through the command port. The command port I/O channel uses the lower 8 bits of the 16-bit wide port for the console and `com` driver I/O; high order bits are set in special command words to switch the command port channel between console and `com` driver modes. Low-level DMA negotiation data are sent with special bit patterns in the high-order bytes, allowing them to interrupt the multiplexed serial-line datastreams at any time without confusion.

The console subsystem does not provide a separate keyboard driver — the host-side `cit` program sends ASCII values to the coprocessor. Special escape sequences are provided to signal the console driver for access to the kernel debugger.

Microkernel DES Engine Driver

The DES engine control code within the coprocessor microkernel is not directly accessible as a driver. Instead, it is an internal device providing multiplexed services to the host emulated disk driver (`hd`) and the DES services driver (`ds`).

Each DES request is packaged in a structure specifying the DMA transfer mode (if any — a request may also be entirely local to the coprocessor), encryption key and initialization vector, transfer size, etc. Encoded with each DMA request is also a *client-id* sent with a DMA request descriptor to the host via the command port. The requests are read by `cit` and acknowledged before initiating the DMA transfer.

Host Emulated Disk

The microkernel contains a host emulated disk driver (`hd`) which uses the DMA multiplexing driver to transfer data blocks to/from the host. The entire disk image provided by `cit` in the host is encrypted, and my code uses encryption/decryption DMA transfers to access it. The default pager using this emulated disk suffices for providing coprocessor-based applications with truly private virtual memory. (Alternatively, crypto-paging could be performed to a single encrypted partition, and the remainder of the disk could stay unencrypted.)

The use of crypto-paging to protect the privacy of virtual memory can be inefficient if the emulated disk block sizes are smaller than the size of a physical page on the host, since the DMA negotiation and setup overhead would be incurred for partial page transfers. For efficiency reasons, the emulated disk's blocksize must be a multiple of the virtual memory page size in the host. Currently, both the VM page size and the emulated disk block size are 4 Kbytes.

A simple extension to the encrypted emulated disk would provide multiple disk images on the host, permitting one or more of them to be used for data sharing with the host (but not for simultaneous access). In a similar fashion, an emulated network interface may be provided to the secure coprocessor, allowing the use of NFS [79] and other network services. In the case of NFS, meta-data (directory information) would not be encrypted.

DES Service Interface

The DES engine interface `ds` provides another multiplexed service, the DES service driver interface. The `ds` interface provides the coprocessor applications access to DES operations — including host-to-host filter mode operations performed on the behalf of host-side applications.

Coprocessor-side applications typically make DES service requests to a crypto-server (`crypt_srv`) which is responsible for scheduling access to the DES engine for both the coprocessor-side applications and the host-side applications that make requests through `cit`. The `crypt_srv` server runs inside the coprocessor, and is the sole client of the `ds` driver. While the scheduling decisions and the simple protocols required to implement them could be performed entirely within the drivers, having the crypto-server implement scheduling policy outside of the kernels leads to gains in overall flexibility.

The `ds` driver provides device-level access to the DES engine, with each `device` remote procedure call request being serially serviced The various modes of operation of the DES engine are set via `device_set_status()` remote procedure call requests; `device_read()` and `device_write()` remote procedure calls turn into DES operations involving local coprocessor-resident data. Another special `device_set_status()` remote procedure call initiates host-only filter operations.

Secure Memory Interface

Dyad's model of secure coprocessors depends on the availability of privacy-protected persistent memory. Such protected memory can hold encryption keys, since privacy is guaranteed in a very strong sense. Similarly, cryptographic checksums are stored in protected memory — integrity of data is well protected by the system, since only the coprocessor system software may modify (or reveal) protected memory contents.

Hardware Secure Memory The Citadel coprocessor system provides 64 kilobytes of battery-backed memory (secure RAM / non-volatile memory) protected by intrusion detection circuitry. The circuitry erases memory if any attempt at physical access is detected,

ensuring the privacy of the memory contents. Additionally, 64 kilobytes of EEPROM is available for persistent (but not necessarily private) storage. Since any attempt at penetration results in erasure of critical keys required to load the coprocessor system software, altering EEPROM contents results in catastrophic failure of the Dyad system.

EEPROM contents may be made private by encrypting the EEPROM contents with a key kept in secure RAM.

Secure Memory Service The Dyad secure coprocessor kernel exports secure RAM and EEPROM raw access to user applications by permitting applications to map these memories into their address space via the `mmap` primitive on the `iopl` device.

We employ a special secure memory server `sec_ram` to provide coprocessor applications with controlled access to secure RAM and EEPROM via a remote procedure call interface allowing clients to read/write their own regions of secure memory. (Alternatively, all coprocessor applications could directly map the secure memory into their own address space.)

Encapsulating secure memory access using a special secure memory server means that errors in the user-level applications within the secure coprocessor are unlikely to corrupt secure memory contents of another application. Furthermore, my `sec_ram` server provides the system with the ability to dynamically allocate secure memory among various coprocessor-side clients; memory compaction to reduce or eliminate fragmentation is also feasible. Additionally, the memory server can implement the common code required to make atomic block updates (16-bit word updates of the secure RAM are assumed to be atomic, since the i386SX uses a 16-bit data bus to write to the secure RAM). Similarly, the `sec_ram` server can mask the complexity of the hardware EEPROM update protocol for the user.[25]

The disadvantage of the `sec_ram` approach is speed, since secure memory accesses would run several hundred times slower than direct access, depending on the size of memory accesses over which the remote procedure call overhead is amortized.

Cryptographic keys are kept in the secure memory by the `sec_ram` server for the various coprocessor applications. Note that applications must have unique IDs for allocating and accessing secure memory from the `sec_ram` server. These IDs are also persistent quantities, since all runs of the same application should access the same data private to the application. Because applications have no access to any persistent memory (other than their own instructions) before contacting the `sec_ram` server, and external non-encrypted storage is vulnerable, there is a bootstrapping problem. We can solve this problem by binding the secure memory access ID with the application at compile time, since coprocessor application binaries are guaranteed their integrity by cryptographic checksum verification.

[25]Making EEPROM updates atomic is harder, since we do not have atomic writes. An entire sixty-four byte page of the Xicor EEPROM used by Citadel must updated in a single step, and each page mode update requires up to 10 mS for the write cycle to complete. Secure RAM can be used to provide a directory into the EEPROM and preserve the appearance of atomic updates of the 64-byte pages.

There is no need to protect the privacy of these ID values, since they only refer to secure memory regions. A drawback is that static allocation of the IDs implies that external ID granting authorities must exist. Because these IDs do not have to be contiguous, the granting authorities may be distributed (much as physical Ethernet addresses are currently allocated). This aspect of application installation ties in with system bootstrapping and maintenance, discussed in chapter 6.

Chapter 5

Cryptographic Algorithms/Protocols

This chapter discusses and analyzes the key algorithms used in Dyad.[26] The notation used is standard from number theory and algebra (groups, rings, and fields).

In addition to the zero-knowledge authentication and key exchange algorithms below, Dyad uses public key signatures and public key encryption [78] (e.g., for copy-protected software distribution). In lieu of the zero-knowledge authentication and key exchange algorithm presented here, RSA or Diffie-Hellman key exchange [25] could be used instead. RSA and Diffie-Hellman have weaker theoretical underpinnings; for example, RSA is known to leak information (the Jacobi symbol) [51], and our zero-knowledge authentication scheme provably does not. Similarly, in lieu of Karp-Rabin fingerprinting, other cryptographic checksum algorithms such as Rivest's MD5 [77], Merkle's Snefru [56], Jueneman's Message Authentication Code (MAC) [44], IBM's Manipulation Detection Code (MDC) [41], or chained DES [102] could be used. Primes needed in the key exchange algorithm, the authentication algorithm, and the two merged key exchange/authentication algorithms may be generated using known probabilistic algorithms such as Rabin's [70].

There are two main sections in this chapter. Section 5.1 describes all of the algorithms in detail. A programmer should be able to reimplement the protocols from this part alone. Section 5.2 revisits the algorithms and provides an analysis of their cryptographic properties.

5.1. Description of Algorithms

Before the description of my algorithms, I define some terms that will be used throughout this section.

A number M is said to be a *Blum modulus* when $M = P \cdot Q$, and P, Q are primes of the form $4k + 3$. Moduli of this form are said to have the *Blum* property. Blum moduli have special number theoretic properties that I will use in my protocols.

A value is said to be a *nonce* value if it is randomly selected from a set S and is used once in a run of a protocol. The nonce values that we will use are usually selected from a ring $\mathbb{Z}_M^*$, where M is a Blum modulus.[27]

[26] This chapter is a slightly revised version of my paper [98]. These algorithms first appeared in the Strongbox system.

[27] $\mathbb{Z}_n^*$ denotes integers modulo n relatively prime to n considered as a group with multiplication as the group operator.

5.1.1. Key Exchange

End-to-end encryption of communication channels is mandatory when channel security is suspect. To do this efficiently, I use private-key encryption coupled with a public-key encryption algorithm used for key exchange. I first describe the public-key algorithm.

What properties do we need in a public-key encryption algorithm? Certainly, we want assurances that inverting the ciphertext without knowing the key is difficult. To show that inverting the ciphertext is difficult, often we show that breaking a cryptosystem is equivalent to solving some other problem that we believe to be hard. For example, Rabin showed that his encryption algorithm is equivalent to factoring large composite numbers, which number theorists believe to be intractable [67]. Unfortunately, Rabin's system is brittle, i.e., if the user's program (or other hardware/software agents working on the user's behalf) can be made to decrypt ciphertext chosen by an attacker, it would be easy for the attacker to subvert the system, divulging the secret keys. The RSA encryption algorithm [78], while believed to be strong, has not been proven secure. Chor [17] showed that if an attacker can guess a single bit of the plaintext when given the ciphertext with an accuracy of more than $1/2 + \epsilon$, then the attacker can invert the entire message. Depending on your point of view, this could be interpreted to mean either that RSA is strong in that not a single bit of the plaintext is leaked, or that RSA is weak in that all it takes is one chink in its armor to break it. The public-key cryptosystem used in Dyad is based on the problem of deciding quadratic residuosity, another well-known number theoretic problem that is believed to be intractable.

When a connection is established between a client and a server, the two exchange a secret, randomly generated DES key using a public key system. Because private key encryption is much cheaper, we use the DES key to encrypt all other traffic between the client and the server.

The public key system works as follows: All entities in the system publish via a white pages server their moduli, $\{M_i\}$, where M_i is a Blum moduli. The factorization of M_i, of course, is known only to the entity corresponding to M_i and is kept secret.

Observe that Blum moduli have the property that the multiplicative group $\mathbb{Z}^*_{M_i}$ has -1 as a quadratic non-residue. To see this, let $L(a, p)$ denote the Legendre symbol, which is defined as

$$L(a,p) = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue, i.e., if } \exists\, x : x^2 \equiv a \pmod{p} \\ -1 & \text{otherwise} \end{cases}$$

where p is prime and $a \in \mathbb{Z}^*_p$. Now, we are going to use two important identities involving the Legendre symbol:[28]

$$L(-1,p) \;=\; -1^{(p-1)/2} \tag{5.1}$$

$$L(m \cdot n, p) \;=\; L(m,p) \cdot L(n,p) \tag{5.2}$$

[28] See [60] for a list of identities involving the Legendre symbol.

When $p = 4k + 3$, from (5.1) we have $L(-1, p) = -1^{2k+1} = -1$, so -1 is a quadratic non-residue in $\mathbb{Z}_p^*$. This suffices to show that -1 is a quadratic non-residue in $\mathbb{Z}_{M_i}^*$, since if there is a root r such that $r^2 = -1 \bmod M_i$, then $(r \bmod p)$ must be a square root of -1 in Z_p^* as well, where p is a prime factor of M_i.

The property that -1 is a quadratic non-residue makes it easy to randomly generate random quadratic residues and non-residues: simply chose a random[29] $r \in \mathbb{Z}_{M_i}^*$ and compute $r^2 \bmod \mathbb{Z}_{M_i}$. If we want a quadratic residue, use $r^2 \bmod M_i$; if we want a quadratic non-residue, use $-r^2 \bmod M_i$.

Therefore, given $n = p \cdot q$ where both p and q are primes of the form $4k + 3$, it is easy to generate random quadratic residues and quadratic non-residues. Next, note another property of quadratic residues that will enable us to decode messages. The important property of the Legendre symbol is that it can be efficiently computed using a algorithm similar to the Euclidean gcd algorithm. Note that this likewise holds for the generalization of the Legendre symbol, the Jacobi symbol, defined by $J(n, m) = \prod_i L(n, p_i)$ where $m = \prod_i p_i$, where the p_i's are the prime factors of m. The value of the Jacobi symbol can be efficiently calculated *without* knowing the factorization of the numbers.

The following approach was described in [30]. Suppose a client wants to establish a connection to the server corresponding to M_i. The client first randomly choses a DES key k, which will be sent to the server using the public key system. The client then decomposes the message into a sequence of single bits, $b_0, b_1, \ldots, b_m$. Now, for each bit of the message b_j, the client computes $x_j \equiv -1^{b_j} r_j^2 \pmod{M_i}$ where r_j are random numbers (nonce values). The receiver i can compute $b_j = L(x_j, P_i)$ to decode the bit stream since he knows the factorization of M_i. Note that while the Jacobi symbol, the generalization of the Legendre symbol, can be quickly computed without knowing the factorization of M_i, it does not aid the attacker. We see from

$$\begin{aligned} J(-r^2, M_i) &= J(-1, M_i) J(r^2, M_i) \\ &= J(-1, P_i) J(-1, Q_i) J(r^2, M_i) \\ &= -1 \cdot -1 \cdot J(r^2, M_i) \\ &= J(r^2, M_i) \\ &= 1 \end{aligned}$$

that quadratic non-residues formed as residues modulo M_i of $-r^2$ will also have 1 as the value of the Jacobi symbol.[30]

---

[29] We can actually just chose $r \in \mathbb{Z}_{M_i}$ and not bother to check that $r \in \mathbb{Z}_{M_i}^*$. If $r \notin \mathbb{Z}_{M_i}^*$, this means that $GCD(M_i, r) \neq 1$ and we've just found a factor of M_i. Since factoring is assumed to be difficult, this is an highly improbable event.

[30] Some cryptographic protocols, such as RSA, leak information through the Jacobi symbol. In RSA, plaintext and corresponding ciphertext always have the same value for their Jacobi symbols. To see this, consider the Legendre symbol: if $L(x, p) = 1$, then there exists a residue r such that $r^2 = x \bmod p$. But $x^e = (r^e)^2 \bmod p$, so r^e is a quadratic residue of x^e. If $L(x, p) = -1$, then $L(x^e, p) = -1$ as well, since e is odd. Because $J(x, pq) = L(x, p)L(x, q)$, $J(x, pq) = J(x^e, pq)$ holds. This information leak can be significant in some applications where only a limited number of messages or message formats are used, since attackers can easily gather statistical information on the distribution of messages.

When the receiver has decoded the bit sequence b_j and reconstructed the message m_i, he installs m_i as the key for DES encryption of the communication channel. From this point on, DES is used to encrypt all coprocessor managed remote procedure call traffic between the client and the server.

5.1.2. Authentication

Whether or not communication channels are secure against eavesdropping or tampering, some form of authentication is needed to verify the identity of the communicating parties. Even if the physical network links are secure, we still need to use authentication: to look up the communication ports of remote servers, we must ask a network name server on a remote, untrusted machine. Since we make no assumptions about the trustworthiness of the network name servers, even the identity of a remote host is suspect. In addition to the existing network name service, the secure coprocessor uses a White Pages server that maintains authentication information (in addition to key exchange moduli when applicable) and is itself an authenticated agent — the White Pages services have digitally signed authentication information associated with them, and so no directory lookup is required for them. The digital signature is generated by a central, trusted authority. For the purposes of this discussion, the role of the White Pages server is to serve as a repository of trusted *authentication puzzles*. Authentication is based on having the authenticator prove that it can solve the published puzzle without revealing the solution.

The best available protocols for authentication all rely on a crucial observation made by Rabin [67]: if one can extract square roots modulo n where $n = p \cdot q$, p and q primes, then one can factor n. This theorem has led the way to practical *zero-knowledge authentication protocols*. Two important examples of practical zero-knowledge protocols include an unpublished protocol first developed in 1987 by Rabin [74], and a protocol developed by Feige, Fiat, and Shamir (the FFS protocol) [27]. Between the FFS and Rabin's protocols, Rabin's method is much stronger because it provides a super-exponential security factor. In contrast to Needham and Schroeder's authentication protocol [59], both of these zero-knowledge authentication protocols require no central authentication server and thus there is no single point of failure that would cripple the entire system. The Dyad system uses a modified version of Rabin's authentication protocol. Like Rabin's protocol, my protocol is decentralized and has a super-exponential security factor.

What do we mean when we say the authentication is *zero-knowledge*? By this we mean that the entire authentication session may be open — an eavesdropper may listen to the entire authentication exchange, but will gain no information at all that would enable him to later masquerade as the authenticator.

Let's see how authentication works. After establishing a secure communication channel with the remote entity, an agent queries the white pages server for its corresponding party's authentication puzzle. Authentication puzzles are randomly generated when a new authenticated entity is created and can be solved only by their owners, who know their secret solutions. However, the remote entity does not exhibit a solution to its puzzle, but

rather is asked to show a solution to a randomized version of its puzzle. My puzzles are again based on quadratic residuosity — this time not on deciding residuosity but on actually finding square roots.

Whenever a new entity is created, an authentication puzzle/solution pair is created for it in an initial, once-only preparatory step — the puzzle is published in the local White Pages server, and the solution is given to the new task. The secure coprocessor creates a new puzzle for every new user of that coprocessor, and the White Pages directory is provided by the secure coprocessor which guarantees its integrity from tampering.

The authentication puzzle consists of a modulus $M_i = p_i \cdot q_i$ and the vector

$$\vec{V_i} = (v_{i,1}, v_{i,2}, \ldots, v_{i,n-1}, v_{i,n})$$

where p_i and q_i are primes, and each $v_{i,j}$ is a quadratic residue in $\mathbb{Z}^*_{M_i}$. The authentication modulus is distinct from the key exchange modulus; in the authentication algorithm, it is not necessary for anyone to know the factors p_i and q_i, and in fact a single modulus can be used for all authentication puzzles. The secret solution is the vector

$$\vec{S_i} = (s_{i,1}, s_{i,2}, \ldots, s_{i,n-1}, s_{i,n})$$

where $s_{i,j}$ are roots of the equations $x^2 \equiv 1/v_{i,j} \pmod{M_i}$. Generating a new solution/puzzle pair is simple: we choose random $s_{i,j} \in \mathbb{Z}_{M_i}$ to form the solution vector, and then element-wise square and invert $\vec{S_i}$ modulo M_i to form the puzzle $\vec{V}$.

Suppose a challenger $\mathcal{C}$ wants to authenticate $\mathcal{A}$'s identity. $\mathcal{C}$ first randomly choses a boolean vector $\vec{E} \in \{0,1\}^n$:

$$\vec{E} = (e_1, e_2, \ldots, e_{n-1}, e_n)$$

where $\vec{E} \circ \vec{E} = \left\lfloor \frac{n}{2} \right\rfloor$, and $\phi \in S_n$ a permutation.[31] We can represent ϕ as a number φ from 0 to $n! - 1$ which represents elements of S_n under a canonical numbering.[32]

The pair $(\vec{E}, \phi)$ is the *challenge* that $\mathcal{C}$ will use to query $\mathcal{A}$. Now, $\mathcal{C}$ encodes $\vec{E}$ and φ as follows:

$$\vec{C} = (c_1, c_2, \ldots, c_{n+\lceil \log(n!) \rceil})$$

where

$$c_i = \begin{cases} -1^{e_i} t_i^2 \bmod M_{pub} & \text{if } 1 \leq i \leq n \\ -1^{\varphi_i} t_i^2 \bmod M_{pub} & \text{otherwise} \end{cases}$$

where φ_i denotes the i^{th} bit of φ and t_i are nonce values from $\mathbb{Z}^*_{M_{pub}}$, and M_{pub} is the Blum modulus that is used by all entities in this initial round, i.e. $M_{pub} = P_{pub}Q_{pub}$, where

[31] $\vec{E} \circ \vec{E}$ denotes the dot product of $\vec{E}$ with itself. S_n denotes the symmetric group of n elements.

[32] Note that this numbering provides a way to randomly choose ϕ: since φ requires $\log(n!)$ bits to represent, we can simply generate $\lceil \log(n!) \rceil$ random bits and use it as a number from 0 to $2^{\lceil \log(n!) \rceil} - 1$. If the number is greater than $n! - 1$, we try again. This procedure terminates in an expected two tries, so on average we expend $2 \lceil \log(n!) \rceil$ random bits. Other approaches are given in [24, 48].

$P_{pub} \equiv Q_{pub} \equiv 3 \pmod 4$. The values of P_{pub} and Q_{pub} are secret and may be forgotten after M_{pub} was generated.

$\mathcal{C}$ sends the encoded challenge $\vec{C}$ to $\mathcal{A}$.

When $\mathcal{A}$ receives $\vec{C}$, $\mathcal{A}$ computes the nonce vector

$$\vec{R} = (r_1, r_2, \ldots, r_{n-1}, r_n)$$

where r_j are randomly chosen from $\mathbb{Z}^*_{M_i}$, and the vector

$$\vec{X} = (x_1, x_2, \ldots, x_{n-1}, x_n)$$

where $x_j \equiv r_j^2 \pmod{M_i}$. The authenticator sends $\vec{X}$, called the *puzzle randomizer*, to the challenger $\mathcal{C}$, keeping the value of $\vec{R}$ secret. As we will see in section 5.2.2, $\vec{X}$ is used to randomize the puzzle in order to keep the solution from being revealed.

$\mathcal{C}$ responds to the puzzle randomizer with $\vec{T} = (t_1, t_2, \ldots, t_{n-1}, t_n)$ of nonce values used to compute $\vec{C}$. Using $\vec{T}$, $\mathcal{A}$ reconstructs $(\vec{E}, \phi)$.

In response to the decoded challenge, $\mathcal{A}$ replies with

$$\vec{Y} = (y_1, y_2, \ldots, y_{n-1}, y_n)$$

where $y_j \equiv r_{\phi(j)} \cdot s_{i,j}^{e_j} \pmod{M_i}$. $\vec{Y}$ is the *response*. To verify, the challenger checks that $\forall\, j : x_{\phi(j)} \equiv y_j^2 \cdot v_{i,j}^{e_j} \pmod{M_i}$.

5.1.3. Merged Authentication and Secret Agreement

Instead of running key exchange and authentication as separate steps, I have a merged protocol that performs secret agreement and authentication at the same time. The protocol performs *secret agreement* rather than key exchange: after the protocol completes, both parties will share a secret, but neither party in the protocol can control the final value of this secret. This merged protocol has the advantage of eliminating a remote procedure call, but requires that the authentication security parameter n (the puzzle size) be at least $2m$, where m is the number of bits in a session key. We do not use this protocol in our current version of the system since we need a much weaker level of security than the $n = 2m$ level. Our merged protocol goes as follows:

As in the normal key exchange protocol, each entity i in the system calculates a Blum modulus $M_i = P_i Q_i$, with P_i and Q_i primes of the form $4k+3$. Entity i keeps the values of P_i and Q_i secret and publishes M_i. Entity i also generates a random puzzle by first generating the desired solution vector

$$\vec{S_i} = (s_{i,1}, s_{i,2}, \cdots, s_{i,n})$$

where the elements of $\vec{S_i}$ are computed by $s_{i,j} = z_{i,j}^2$, where $z_{i,j}$ is a random number from $\mathbb{Z}^*_{M_i}$. Then, i publishes the puzzle vector

$$\vec{V_i} = (v_{i,1}, v_{i,2}, \cdots, v_{i,n})$$

with $v_{i,j} = 1/s_{i,j}^2$. With both M_i and V_i are published, i is ready to authenticate and exchange keys.

When the challenger $\mathcal{C}$ wishes to verify $\mathcal{A}$'s identity and obtain a session key from $\mathcal{A}$, $\mathcal{C}$ first chooses a challenge $(\vec{E}, \phi)$ as before, with $\vec{E} \in \{0,1\}^n$ such that $\vec{E} \circ \vec{E} = \left\lfloor \frac{n}{2} \right\rfloor$, and permutation $\phi \in S_n$. Just as in the previous authentication protocol, $\mathcal{C}$ encodes $\vec{E}$ and ϕ

$$\vec{C} = (c_1, c_2, \ldots, c_{n+\lceil \log(n!) \rceil})$$

where

$$c_j = \begin{cases} -1^{e_j} t_j^2 \bmod M_{pub} & \text{if } 1 \leq i \leq n \\ -1^{\varphi_j} t_j^2 \bmod M_{pub} & \text{otherwise} \end{cases}$$

where φ is the canonical numbering of $\phi \in S_n$, φ_j denotes the j^{th} bit of φ, t_j is a nonce value from $\mathbb{Z}^*_{M_{pub}}$, and M_{pub} is a Blum modulus. $\mathcal{C}$ sends $\mathcal{A}$ the encoded challenge $\vec{C}$. Let $\vec{T}$ denote the vector of nonce values used to generate $\vec{C}$.

$\mathcal{A}$ computes a puzzle randomizer $\vec{X}$ by first computing a pre-randomizer $\vec{R}$, which will be used to transmit the key bits. $\mathcal{A}$ computes $\vec{R}$

$$\vec{R} = (r_1, r_2, \ldots, r_{n-1}, r_n)$$

by randomly choosing the nonce vector

$$\vec{W} = (w_1, w_2, \ldots, w_{n-1}, w_n)$$

The values w_j are chosen from $\mathbb{Z}^*_{M_a M_c}$, where M_a is the published modulus of $\mathcal{A}$ and M_c is the published modulus of $\mathcal{C}$. The value of $\vec{R}$ is obtained by setting $r_j = -1^{b_j} \cdot w_j^2 \bmod \mathbb{Z}_{M_a M_c}$, where b_j is a random bit. Some of the these bits b_j will form the secret transferred. Next, $\mathcal{A}$ computes the puzzle randomizer $\vec{X}$ from $\vec{R}$ as before, setting $x_j = r_j^2 \bmod \mathbb{Z}_{M_a M_c}$, and sends $\vec{X}$ to $\mathcal{C}$.

Now, $\mathcal{C}$ reveals the challenge $(\vec{E}, \phi)$ by sending $\mathcal{A}$ the vector $\vec{T}$; in response, $\mathcal{A}$ sends $\vec{Y}$ with

$$y_j = r_{\phi(j)} \cdot s_{a,j}^{e_j} \bmod (M_a M_c^{1-e_j})$$

To verify $\mathcal{A}$'s identity, $\mathcal{C}$ checks that

$$\forall\, j \colon x_{\phi(j)} = y_j^2 v_{a,j}^{e_j} \bmod M_a$$

There are $\left\lceil \frac{n}{2} \right\rceil$ usable key bits transferred, and they correspond to those b_j for which $e_j = 0$. To extract b_j, $\mathcal{C}$ computes the Legendre symbol $L(y_j, P_c)$ to determine whether y_j is a quadratic residue. If y_j is a quadratic residue, then $b_j = 0$; otherwise, $b_j = 1$.

5.1.4. Practical Authentication and Secret Agreement

In this section, I present another protocol for simultaneous authentication and secret agreement requiring two rounds of interaction but fewer random bits. Furthermore, the message sizes are smaller, thus making this protocol more practical. This protocol strikes the best balance between performance and security, and I have implemented it for Dyad.

Each agent $\mathcal{A}$ who wishes to participate in the protocol generates a modulus M_a with secret prime factors P_a and Q_a. Each agent also generates a vector of secret numbers

$$\vec{S_a} = (s_{a,1}, s_{a,2}, \cdots, s_{a,n})$$

where $s_{a,i} \in \mathbb{Z}^*_{M_a}$. From this $\vec{S_a}$, $\mathcal{A}$ computes

$$\vec{V_a} = (v_{a,1}, v_{a,2}, \cdots, v_{a,n})$$

where $v_{a,i} = 1/s^4_{a,i} \bmod M_a$. Published for all to use is a modulus M_{pub}; the two prime factors of M_{pub}, P_{pub} and Q_{pub}, are forgotten as in the previous protocol.

Now, suppose a challenger $\mathcal{C}$ wishes to verify the identity of an authenticator $\mathcal{A}$. Assume the parties have published their moduli M_c and M_a, respectively, and that $\mathcal{C}$'s puzzle vector $\vec{V}$ has also been published. First, $\mathcal{C}$ chooses a bit vector

$$\vec{E} = (e_1, e_2, \cdots, e_n)$$

where $\vec{E} \circ \vec{E} = \left\lfloor \frac{n}{2} \right\rfloor$, and a permutation $\phi \in S_n$. The pair $(\vec{E}, \phi)$ is the challenge that $\mathcal{C}$ will use later in authentication. Let $\zeta = \binom{n}{\lfloor \frac{n}{2} \rfloor}$, the number of possible vectors $\vec{E}$. Encode both $\vec{E}$ and ϕ as numbers using mappings $f : \{\vec{E}\} \leftrightarrow \mathbb{Z}_\zeta$ and $g : S_n \leftrightarrow \mathbb{Z}_{n!}$. Let $E = g(\phi) \cdot \zeta + f(\vec{E})$, the combined encoding for the two parts of the challenge,[33] and let $C = E^2 \bmod M_{pub}$. The value C is used to commit the value of $\mathcal{C}$'s challenge to $\mathcal{A}$, preventing $\mathcal{C}$ from changing it after learning the puzzle randomizer. $\mathcal{C}$ sends C to $\mathcal{A}$.

In response, $\mathcal{A}$ generates a puzzle randomizer by choosing

$$\vec{R} = (r_1, r_2, \cdots, r_n)$$

where each r_i is a nonce value chosen from $\mathbb{Z}_{M_a M_c}$. $\mathcal{A}$ creates the puzzle randomizer vector $\vec{X}$ from this by setting

$$\vec{X} = (x_1, x_2, \cdots, x_n)$$

where $x_i = r_i^4 \bmod M_a M_c$. $\mathcal{A}$ sends $\vec{X}$ to $\mathcal{C}$. $\mathcal{C}$ will have to recover some of the values of $\vec{R}$ in order for the protocol to work. These values will become the agreed upon secret used as private keys. $\mathcal{C}$ will recover exactly those r_i where $e_i = 0$. There are exactly $\left\lceil \frac{n}{2} \right\rceil$ such values. Let those i such that $e_i = 0$ be the set I.

[33] If $|E| \neq |M_{pub}|$, extra random pad bits may be necessary.

When $\mathcal{C}$ receives the puzzle randomizer, $\mathcal{C}$ replies by revealing the challenge by sending E to $\mathcal{A}$.

$\mathcal{A}$ verifies that this E encodes the challenge that corresponds to the challenge commitment value C by checking that $C = E^2 \bmod M_{pub}$. If the encoding is correct, $\mathcal{C}$ extracts the challenge tuple $(\vec{E}, \phi)$, and computes

$$\vec{Y} = (y_1, y_2, \cdots, y_n)$$

where $y_i = r_{\phi(i)}^2 s_i^{2e_i} \bmod M_a^{e_i} M_c^{1-e_i}$.

Now $\mathcal{A}$ composes a special vector $\vec{W}$. The ith entry of this vector will be the pair

$$(w_i, h_{u_i}(w_i))$$

where $i \in I$, $u_i = r_{\phi(i)}$, w_i is a nonce value, and h_k is an element of a family F of cryptographic hash functions. $\mathcal{A}$ sends $\vec{Y}$ and $\vec{W}$ to $\mathcal{C}$.

$\mathcal{C}$ verifies that

$$\forall\, i\colon y_i^2 v_i^{e_i} = x_{\phi(i)} \bmod M_a^{e_i} M_c^{1-e_i}$$

If each y_i passes this test, $\mathcal{C}$ then examines the values of y_i for which $e_i = 0$: since

$$y_i = r_{\phi(i)}^2 \bmod M_c$$

and $\mathcal{C}$ knows the factorization of M_c, $\mathcal{C}$ can extract the four square roots of $y_i \bmod M_c$, one of which was the original $r_{\phi(i)}$ chosen by $\mathcal{C}$.[34] To choose the proper root of y_i, $\mathcal{C}$ uses the ith element of $\vec{W}$. $\mathcal{C}$ can try all four square roots of $y_i \bmod M_c$ and see which one gives the value that matches the value sent by $\mathcal{A}$. This assumes that F is immune from known plaintext attacks. (One class of functions that could be used as F is a family of encryption functions.)

5.1.5. Fingerprints

Next, I describe the Karp-Rabin fingerprinting algorithm, which is crucial to Dyad's ability to detect attackers or security problems in the underlying system. The key idea is this: associated with each file — in particular, every trusted program generated by trusted editors/compilers/assemblers/linkers/etc. — is a *fingerprint* which, like a normal checksum, detects modifications to the data. Unlike normal checksums, however, fingerprints are parameterized by an irreducible polynomial[35] and the likelihood of an attacker forging a fingerprint without knowing the irreducible polynomial is exponentially small in the degree of the polynomial.

---

[34] Standard algorithms for modular square root computation are given in [3, 7].

[35] A polynomial $p(x) \in F[x]$ (F a field) is said to be *irreducible* if $\nexists f(x) \in F[x]$: $f(x) \mid p(x), 0 < \deg f < \deg p$, i.e., the only divisors are p and nonzero elements of F (the units of $F[x]$). This is analogous to primality for integers.

Dyad chooses random irreducible polynomials p from $\mathbb{Z}_2[x]$ of degree 31 by the algorithm due to Rabin [45, 68, 71].

Here is one way to visualize the fingerprinting operation: We take the irreducible polynomial $p(x)$, arrange the coefficients from left to right in decreasing order, i.e., with the x^{31} term of $p(x)$ at the leftmost position, and scan through the input bit stream from left to right. If the bit in the input opposite the x^{31} term is set, we exclusive-or $p(x)$ into the bit stream. As we scan down the bit stream all coefficients to the left of the current position of the x^{31} term of $p(x)$ will be zeros. When we reach the end of the bit stream, i.e., the x^0 term of $p(x)$ is opposite the last bit of the input stream, we will have computed $f(x) \bmod p(x) = \varphi(f(x))$.

5.2. Analysis of Algorithms

5.2.1. Key Exchange

The correspondence between the problem of deciding quadratic residuosity and the protocol is direct. For a detailed analysis, see [30].

5.2.2. Authentication

What are the chances that a system breaker $\mathcal{B}$ could break the first (unmerged) authentication scheme? As we stated before, we assume that the modulus M_i is sufficiently large so that factoring it is impractical. Now, consider what $\mathcal{B}$ must do to pose as $\mathcal{A}$.

Let us first look at a simpler authentication system to gain intuition. Let the puzzle and the secret solution be v and s where $v = 1/s^2$; let the puzzle randomizer be $x = r^2$ (r known only to the authenticator); let the challenge be $e \in \{0, 1\}$; and let the response be $y = r \cdot s^e$. All calculations are done modulo M.

We claim that if $\mathcal{B}$ could slip through our authentication procedure with more than probability $\frac{1}{2}$, then $\mathcal{B}$ could extract the square roots and thus factor M, violating our basic assumption. To wit, in order for $\mathcal{B}$ to reliably pass the authentication procedure, it must be able to handle the case where e is either 1 or 0, and thus it would need to know both r and $r \cdot s$. This means that he would be able to compute the square root of v, which we know from Rabin [67] is equivalent to factoring.

What must $\mathcal{B}$ do in the full version of the authentication? In order to pass the challenge, $\mathcal{B}$ must know the value of $\vec{E}$. In addition, $\mathcal{B}$ must know part of ϕ. In particular, $\mathcal{B}$ does not have to guess all of ϕ but only those values selected by the 1 entries in $\vec{E}$.

Thus, while

$$\left|\{(\vec{E}, \phi) : \vec{E} \in \{0,1\}^n, \vec{E} \circ \vec{E} = \left\lfloor \tfrac{n}{2} \right\rfloor, \phi \in S_n\}\right| = \binom{n}{n/2} n!,$$

our the security factor (the inverse of the probability of breaking the system) is slightly smaller. Our authentication system provides, for puzzles of n numbers, a probability of an

attacker breaking the authentication system of

$$\begin{aligned} P &= \frac{1}{\binom{n}{n/2} n!/\frac{n}{2}!} \\ &= \frac{(n/2)!^3}{n!^2} \\ &\approx \frac{\left(\frac{2\pi n}{2}\right)^{\frac{3}{2}}\left(\frac{n}{2e}\right)^{\frac{3n}{2}}}{(2\pi n)\left(\frac{n}{e}\right)^{2n}} \\ &= \frac{\sqrt{2\pi n}\, e^{\frac{n}{2}}}{2^{\frac{3}{2}(n+1)} n^{\frac{n}{2}}} \\ &= \frac{\sqrt{\pi}\, e^{\frac{n}{2}}}{2^{\frac{3}{2}n+1} n^{\frac{n-1}{2}}} \end{aligned}$$

(using the Stirling's approximation of $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$) which shows that P is clearly super-exponentially small. By using longer vectors or multiple vectors (iterating) the security factor can be made arbitrarily high. Note that since the security factor is super-exponential in n, the puzzle size, and only multiplicative when the protocol is iterated, increasing puzzle size is usually preferable: If n', the new size of the puzzle, is $2n$, then the probability of successfully breaking the system becomes

$$\begin{aligned} P' &\approx \frac{\sqrt{\pi}\, e^n}{2^{3n+1}(2n)^{n-\frac{1}{2}}} \\ &= \frac{\sqrt{\pi 2n}\, e^n}{2^{3n+1}(2n)^n} \\ &= \frac{\sqrt{2\pi n}\, e^n}{2^{\frac{3n}{2}+1} 2^{\frac{3n}{2}} 2^n n^n} \\ &= \frac{2\sqrt{2}(\pi n) e^n}{2^{\frac{3n}{2}+1} 2^{\frac{3n}{2}+1} 2^n \sqrt{\pi n}\, n^n} \\ &= \frac{P^2}{2^{n-\frac{3}{2}}\sqrt{\pi n}} \end{aligned}$$

On the other hand, if we simply run the protocol twice, we would only obtain $P' = P^2$. Iterating does have one advantage: it makes the selection of the security factor $(1/P)$ flexible. Using iteration makes it easy for applications at different security levels to negotiate the desired security of the connection.

How did we arrive at the expression for P? $1/P$ simply measures the number of equiprobable random states visible to the attacker. First, note that $\binom{n}{n/2}$ is the number of different $\vec{E}$ where $\vec{E} \circ \vec{E} = \left\lfloor \frac{n}{2} \right\rfloor$ (i.e., the number of 1 bits in $\vec{E}$ is $\left\lfloor \frac{n}{2} \right\rfloor$). The $n!/(n-i)!$ term gives the number of ways of chosing i objects from n without replacement, which is what the projection, as specified by the on (i.e., 1) values in $\vec{E}$, of the permutation ϕ gives us.

Why do we restrict $\vec{E}$ to have $\left\lfloor \frac{n}{2} \right\rfloor$ on bits? If $j = \vec{E} \circ \vec{E}$ could be any value, then there would be $\sum_{k=0}^{n} \binom{n}{k} \frac{n!}{(n-k)!}$ different states visible to $\mathcal{B}$ *not all of which would be equiprobable* if $\vec{E}$ and ϕ are chosen uniformly from $\{0,1\}^n$ and S_n. In particular, it can be seen that the state corresponding to $j = 0$ is most probable. This weakens the security factor of our

algorithm. In the limit case where $\vec{E}$ is the zero vector, our algorithm no longer provides super-exponential security.

Note that my protocol provides super-exponential security only if the moduli remain unfactored. Since there is an exponential time algorithm for factoring, it is always possible to break the system in the minimum of the time for factoring and the super-exponential bound. Thus we can scale our protocol in a variety of ways.

The authentication protocol not only provides super-exponential security when the moduli cannot be factored, but is also zero knowledge. The encoded challenge vector, $\vec{C}$, performs *bit commitment* [13, 14, 21, 72, 73], forcing $\mathcal{C}$ to choose the challenge values prior to $\mathcal{A}$ choosing the puzzle randomizer. This means that $\vec{E}$ and ϕ can not be a function of $\vec{X}$, and thus the challenger's side of the protocol can be simulated by an entity that does not have knowledge of any of the secrets. Any entity S can simulate both sides of the protocol — S can choose random $\vec{E}$, ϕ, and, knowing their values, construct vectors $\vec{X}'$ and $\vec{Y}'$ that will pass the verification step:

$$\begin{array}{ll} y_j = r_{\phi(j)}, x_j = r_j^2 & \text{if } e_j = 0 \\ y_j = r_{\phi(j)}, x_j = r_j^2 \cdot v_{i,\phi^{-1}(j)} & \text{if } e_j = 1 \end{array}$$

Note that my model differs slightly from the usual model for zero knowledge interactive proofs because both the prover and the verifier are assumed to be polynomial time (and that factoring and quadratic residuosity are not in polynomial time); if the prover were infinitely powerful, as in the usual model, the prover could simply factor the moduli used in the bit commitment phase of our protocol. Other bit commitment protocols may be used instead; e.g., we could use a protocol based on the discrete log problem [83] requiring more multiplications but use fewer random bits.

5.2.3. Merged Authentication and Secret Agreement

Like the first authentication algorithm, the merged authentication and key exchange algorithm reveals no information if factoring and deciding quadratic residuosity are intractable.

How does the merged algorithm differ from the original algorithm? I use $M_a M_c$ as the modulus for the nonce vectors, and I use quartic residues instead of quadratic residues for the puzzle randomization vector $\vec{X}$.

No information is leaked. An analysis similar to that done above establishes this fact. When $e_j = 1$, we know that

$$\begin{aligned} y_j &= r_{\phi(j)} \cdot s_{a,j} \bmod M_a \\ &= -1^{b_j} \cdot w_{\phi(j)}^2 \cdot z_{a,j}^2 \bmod M_a \\ &= -1^{b_j} \cdot (w_{\phi(j)} z_{A,j})^2 \bmod M_a \end{aligned}$$

so y_j looks like the square of a random number, possibly negated, in $\mathbb{Z}_{M_a}^*$. The challenger $\mathcal{C}$ or an eavesdropper could have generated this without $\mathcal{A}$'s help. (Note that the reason that this value is computed modulo M_a is because $s_{a,j}$ is the residue modulo M_a of a random

square; if we computed y_j modulo $M_a M_c$, we would have no guarantees as to whether $s_{a,j}$ would be a quadratic residue.)

When $e_j = 0$, we have

$$\begin{aligned} y_j &= r_{\phi(j)} \bmod M_a M_c \\ &= -1^{b_j} \cdot w_{\phi(j)}^2 \bmod M_a M_c \end{aligned}$$

This is just the square of a random value, possibly negated, in $\mathbb{Z}_{M_a M_c}$. The challenger $\mathcal{C}$ or any eavesdropper could have generated this without $\mathcal{A}$'s help as well.

This proves that one atomic round of the authentication leaks no information. As with the vanilla authentication, the vectors $\vec{C}$ and $\vec{T}$ provide bit commitment, forcing the challenge $(\vec{E}, \phi)$ to be independent of $\vec{X}$, thus running the atomic rounds in parallel rather than in serial has no impact on the proof of zero knowledge.

Might some system breaker $\mathcal{B}$ compromise the authentication? To do so, $\mathcal{B}$ must guess the values of $\vec{E}$ and ϕ just as in the vanilla authentication protocol. As before, the probability of somebody breaking the authentication is super-exponentially small. (See section 5.2.2)

The bits of the session key (b_j) are transferred only when $e_j = 0$. When $e_j = 1$, $\mathcal{C}$ cannot determine the quadratic residuosity of the element y_j since we assume that determining quadratic residuosity is intractable without the factorization of M_a. When $e_j = 0$, on the other hand, $\mathcal{C}$ can easily determine the quadratic residuosity of y_j by simply evaluating the Legendre symbol $L(y_j, P_c)$.

5.2.4. Practical Authentication and Secret Agreement

Assuming that factoring is intractable, the third protocol (my "practical" protocol) is also zero knowledge. In particular, breaking this protocol is equivalent to factoring: any system breaker $\mathcal{B}$ who has a strategy that allows $\mathcal{B}$ to masquerade as $\mathcal{A}$ can trivially adapt the strategy to factor the various moduli in the system.

Let us examine how this authentication/secret agreement protocol differs from the previous one. Instead of using the quadratic residuosity decision problem to do bit commitment, this protocol uses the Rabin function, removing the requirement that the moduli have the Blum property. Since neither $\mathcal{A}$ nor $\mathcal{C}$ can factor, neither of them can extract the square root of an arbitrary number mod M_{pub}. In particular, $\mathcal{A}$ has no way of getting the encoding E from the commitment value C; the only way $\mathcal{A}$ finds out the value of C (and thus the value of $(\phi, \vec{E})$) is for $\mathcal{C}$ to reveal C. The challenge commitment works as before.

The analysis for the authentication properties are identical to that for the previous protocols, so I omit that here. (See section 5.2.2.) What about the zero-knowledge property?

When $e_j = 1$, we know that

$$\begin{aligned} y_j &= r_{\phi(j)}^2 \cdot s_{a,j}^2 \bmod M_a \\ &= (r_{\phi(j)} \cdot s_{a,j})^2 \bmod M_a \end{aligned}$$

so y_j looks like the square of a random number in $\mathbb{Z}_{M_a}^*$. The challenger $\mathcal{C}$ or an eavesdropper could have generated this without $\mathcal{A}$'s help. Note that the reason that this value is computed

modulo M_a is because $s_{a,j}$ is the residue modulo M_a of a random square; if we computed y_j modulo $M_a M_c$, we would have no guarantees as to whether $s_{a,j}$ would be a quadratic residue.

When $e_j = 0$, we have

$$y_j \;=\; r^2_{\phi(j)} \bmod M_c$$

This is just the square of a random value in $\mathbb{Z}^*_{M_c}$. The challenger $\mathcal{C}$ or any eavesdropper could have generate this without $\mathcal{A}$'s help as well.

In both cases, a simulator $\mathcal{S}$ who pretends to be $\mathcal{A}$ and is able to control the coin flips of $\mathcal{C}$ can easily produce a run of the protocol where the message traffic is indistinguishable from that of an actual run. Since $\mathcal{S}$ can simulate the protocol without the secret known only to $\mathcal{A}$, the protocol is zero knowledge.

This protocol is much more efficient than the previous one, since it sends a factor of $|M_c|$ more secret bits than the previous algorithm; this efficiency is somewhat offset by the fact that root extraction must be performed by the receiver, and extracting square roots is more expensive than computing the Legendre symbol.

5.2.5. Fingerprints

Before we analyze the performance of the fingerprint algorithm, we will fix some notation. We let p (or $p(x)$) refer to an irreducible polynomial of degree m (where m is prime). We use the symbol $\twoheadrightarrow$ to denote surjective mappings, and $\widetilde{F}$ to denote the algebraic closure of the field F.

How good is the fingerprint algorithm? Choosing random irreducible polynomials is equivalent to chosing random homomorphisms $\varphi\colon \mathbb{Z}_2[x] \twoheadrightarrow GF(2^m)$, where the kernel of φ is the ring generated by the irreducible polynomial p. To be precise, φ associates the indeterminate x with u, a root of the irreducible polynomial in the field $\widetilde{\mathbb{Z}_2}$, i.e., $\varphi\colon \mathbb{Z}_2[x] \twoheadrightarrow \mathbb{Z}_2(u) \cong GF(2^m)$. There are exactly $(2^m - 2)/m$ such homomorphisms. To compute the fingerprint of a file, consider the contents of the file as a large polynomial in $\mathbb{Z}_2[x]$: take the data as a string of bits $b_n, b_{n-1}, \ldots, b_1, b_0$, and construct the polynomial $f(x) = \sum_{i=0}^{n} b_i x^i$. The fingerprint is exactly $\varphi(f(x))$.

Now, f can have at most $\left\lfloor \frac{n}{m} \right\rfloor$ divisors of degree m. Any two distinct polynomials f_1 and f_2 will have the same residue if $f_1 - f_2 \equiv 0 \bmod p$. The number of polynomial divisors of $f_1 - f_2$ is at most n/m, so the probability that a random irreducible polynomial gives the same residue for f_1 and f_2 is $\frac{n/m}{(2^m-2)/m} = n/(2^m - 2)$. For a page of memory containing 4 kilobytes of data ($n = 2^{15}$, or 32 kilobits), and setting m to be 31, this probability is less than 0.002%.

This 0.002% probability measures the odds that an adversary's replacement 4 kilobyte page of a file would have a residue that matches that of the original — because the adversary has no knowledge of the particular homomorphism used, there is no better strategy than guessing a polynomial (i.e., the data in the replacement page). The probability that the adversary could guess the homomorphism is $31/(2^3 1 - 2)$ or less than 0.0000015%, which

is much less likely. Hence we can see that the fingerprint algorithm is an excellent choice as a cryptographic checksum.

The naive implementation of this algorithm is quite fast, but it is possible to achieve even faster algorithms by precomputation. Given a fixed p, and a set of small polynomials, we construct a table T of residues of those polynomials. I initially describe the algorithm for arbitrary sized p; afterwards, I describe optimizations specific to $m = \deg p = 31$.

Let T be the table of residues of all polynomials of the form $g(x) \cdot x^m$, where g varies over polynomials of degree less than k. In other words, T gives us the function $\varphi(g(x) \cdot x^{\deg p})$ where $\deg g(x) < k$. Using T allows us to examine k bits at a time from the input stream instead of one at a time. View $f(x)$ now as

$$f(x) = \sum_{i=0}^{\left\lceil \frac{n}{k} \right\rceil} a_i(x) x^{i \cdot k}$$

where $\deg a_i(x) < k$. The algorithm to compute the residue $r(x) = f(x) \bmod p(x)$ becomes the code shown in Figure 5.1.

```
r(x) = 0;
for (i = ⌈n/k⌉; i ≥ 0; --i) {
        r'(x) = r(x) · x^k + a_i(x);
        r(x) = r'(x) mod p(x);
}
```

Figure 5.1 Fingerprint residue calculation. The operation $r'(x) \bmod p(x)$ is performed by decomposing r' into $g(x) \cdot x^m + h(x)$, where $\deg g < k$ and $\deg h < m$, finding $r''(x) = g(x) \cdot x^m \bmod p(x)$ from T, and setting $r(x) = r''(x) + h(x)$.

If we fix the value $m = \deg p = 31$, we can realize further size-specific optimizations. We can represent p exactly in a 32-bit word. Furthermore, since word at a time operations work on 32 bits at a time, by packing the coefficients as bits in a word we can perform some basic operations on the polynomials as bit shifts and exclusive-ors: multiplication by x^k is a left-shift by k bits; addition or subtraction of two polynomials is just exclusive-or. Of course, since we are dealing now with fixed-size machine registers, we must take care not to overflow.

In Dyad, I have two versions of the fingerprinting code, one for $k = 8$ and the other for $k = 16$, both of which use irreducible polynomials of degree 31. To read the input stream a full 32-bit word at a time, I modified the algorithm slightly: instead of T being a table of $\varphi(g(x) \cdot x^{\deg p})$, T contains $\varphi(g(x) \cdot x^{32})$; the code above is modified correspondingly. While the residues $\varphi(g(x) \cdot x^{32})$ require only 31 bits to represent, T is represented as a table of machine words with 2^k entries. The program can uniquely index into the table by evaluating $g(x)$ at the point $x = 2$ (this index is just the coefficient bits of g, which are already stored in a machine word as an integer). If we run the code loop to perform this operation, we will

get a 32-bit result, which represents a polynomial of degree at most 31. Hence the result of the loop, $r(x)$, is either the residue $R(x) = f(x) \bmod p(x)$ or $R(x) + p(x)$, and the following simple computation fixes up the result:

$$\varphi(f(x)) = \begin{cases} r(u) & \text{if } \deg r(x) < 31 \\ (r-p)(u) & \text{otherwise} \end{cases}$$

A particularly elegant implementation is achieved when we set k to be 8 or 16. The code in Figure 5.2 illustrates the algorithm for $k = 16$.

```
fp_mem(a,nwords,p,table)
unsigned long *a, p, *table;
int           nwords;
{
        unsigned long   r, rlo, rhi, a_i;
        int             i;

        r = 0;
        for = (i = 0; i < nwords; i+{}+) {
                a_i = a[i];
                rhi = r >> 16;
                rlo = (r << 16) ^ (a_i >> 16);
                r = rlo ^ table[rhi];
                rhi = r >> 16;
                rlo = (r << 16) ^ (a_i & ((1 << 16)-1));
                r = rlo ^ table[rhi];
        }
        if (r >= 1 << 31) r ^= p;
        return r;
}
```

Figure 5.2 Fingerprint calculation (C code).
This C code shows how using a precomputed table of partial residues can speed up fingerprint calculations. Unlike the actual code within Dyad, it omits loop unrolling, forces memory to be aligned, and may perform unnecessary memory references.

For the case where $k = 16$, initializing T will be time consuming if we use the simple brute force method. Instead of calculating each of the 2^{16} entries directly, we first compute the table T' for $k = 8$, size 256, and then T is bootstrapped from T' in the obvious manner: for each entry in T, we simply use its index $g(x)$, decompose it into $g(x) = g_{hi}(x) \cdot x^8 + g_{lo}(x)$ where $\deg g_{hi} < 8$ and $\deg g_{lo} < 8$, and compute $T'[T'_{hi}(g_{hi}) \oplus g_{lo}] \oplus T'_{lo}(g_{hi}) \cdot x^8$ as the table entry.

If a higher security level is required, multiple fingerprints can be taken on the same data, or polynomials of higher degree may be used. The speedup techniques extend well to handle $\deg p(x) = 61$, the next prime[36] close to a multiple of word size, though the number of working registers required (if implemented on a 32-bit machine) doubles. Our current

[36]While the algorithm for finding irreducible polynomials does not require that the degree be prime, using polynomials of prime degree makes counting irreducibles simpler.

implementation is largely limited by the main memory bandwidth on the Citadel CPU's bus for reading the input data and the table size. Note that the table for $k = 8$ can easily fit in most modern CPU memory caches. If we use main memory to store intermediate results, performance dramatically degrades.

Chapter 6

Bootstrap and Maintenance

On the face of it, securely initializing and bootstrapping a secure coprocessor's system software can be very simple: burn *all* the code into the embedded ROM so the coprocessor will always run secure code. Unfortunately, this strategy is unrealistic.

Practical requirements complicate the secure initialization and bootstrap of secure software running in a secure coprocessor:

- maintenance and revocation updates of the trusted software by the secure coprocessor system software vendor (or a trusted authority);
- installation of optional software by local system administrators;
- efficiency of secure bootstrap; and
- security.

Two aspects of bootstrapping go hand in hand: secure bootstrapping, and bootstrapping security. The former deals with verifying code integrity so untrusted code will not be executed with any privileges[37], and the latter deals with increasing security guarantees provided by the system related to bootstrapping, using basic security properties of lower system levels as a basis [97, 98].

The process of secure bootstrapping must provide means of proving the trustworthiness and correct initialization of the final system to the end user. Additionally, depending on the users' degree of trust in the secure coprocessor hardware vendors / system software vendors, we may need to prove to the user (or site security administrator) that the coprocessor hardware (having passed through the system software vendor for initialization) is legitimate. This chapter addresses bootstrapping; the next chapter addresses the verification of system software and hardware.

Digital signatures and cryptographic checksums are basic tools we use to attack secure initialization, bootstrapping, and maintenance. These tools are applied by each layer of bootstrapping code to verify the integrity and authenticity of the next higher layer, ensuring that only trusted code is booted.

[37] The secure coprocessor, when booted, runs a secure form of the Mach microkernel. If administered correctly, untrusted user-level code may be loaded and run after booting.

6.1. Simple Secure and Bootstrap

As a thought experiment, consider the simplest instantiation of secure bootstrapping: the bootstrap ROM for the secure coprocessor contains digital signature checking code. At boot time, this digital signature code verifies that the host-supplied kernel image is from a trusted authority. The trusted authority's public key may be kept in ROM rather than secure RAM, since only integrity and not secrecy is required.[38] The security kernel uses an encrypted file system image supplied by the host to load system servers and applications (the decryption key is kept in secure RAM). This preserves privacy and integrity guarantees for the rest of the operating system and the applications, thus securely bootstrapping to a fully running system.

There are several things wrong with the above scenario. it is inflexible: it allows only centralized updates of system software and data; it requires (computationally expensive) digital signature verification for the kernel; it does not permit revocation of old microkernel images (which may have security bugs); and it does not permit resetting of the coprocessor. Fortunately, by providing a layered security bootstrap, all these flaws can be fixed.

6.2. Flexible Secure Bootstrap and Maintenance

By necessity, secure bootstrapping starts with code embedded in the coprocessor's ROM. This code must be simple — because such embedded code cannot be fixed, its correctness must be certain. This code must be public — an attacker can gain access to it by destroying a secure coprocessor's physical encapsulation. To allow more complex boot code to be used, the boot process proceeds in stages, where the primary boot code in ROM loads in a secondary boot loader from an external source.

Dyad assumes a write-only model of installing the secondary boot code. The secondary boot code, along with any private data it needs, is stored in secure RAM after the secure RAM is cleared. There is no need to trust secondary boot code since no secrets are stored in the secure coprocessor at initialization time — furthermore, users wishing to perform behavioral testing of the secure coprocessor hardware may load their own code at this point to validate the hardware.

The secondary boot code loaded by a trusted secure coprocessor software vendor is loaded with a secret allowing secondary boot code to authenticate its identity. This secret is loaded at the same time as the secondary boot code, and is privacy protected: (1) the tamper detection circuitry will erase the secure RAM if any physical intrusion is detected; (2) the primary bootstrap loader will erase secure RAM prior to loading other secondary bootstrap code; and (3) the secondary bootstrap code reveals not even partial information about its authentication secret, since it uses a zero knowledge authentication. In addition to the authentication secret, the secondary boot code is provided with cryptographic checksums

[38]The ROM in the coprocessor cannot provide secrecy, since an attacker can sacrifice a secure coprocessor to discover ROM contents (which are likely to be uniform across all secure coprocessors.)

of the coprocessor kernel and coprocessor system programs, permitting validation of the next higher layer of code.

To limit the amount of secure RAM used, Dyad stores just the authentication secrets and a cryptographic checksum of the secondary boot code, with actual secondary bootstrap code being read from the host's disk or other external memory at boot time.[39]

This method of initializing the secure coprocessor permits loading of both secure coprocessor vendor authentication data as well as verification data for secondary boot code, yet prevents reinitialization from leaking sensitive data.

The primary boot code is permitted only two operations: installing the secondary boot code along with its authentication secrets; and loading, validating, and running the secondary boot.[40] The secondary boot code authenticates its identity — and thus the identity of the secure coprocessor software vendor — to the user. It also validates and boots the secure coprocessor kernel.

Secondary boot code in secure RAM can permit multiple versions of secure coprocessor kernels, since it can store several cryptographic checksums, each corresponding to a different coprocessor kernel. This permits the system administrators to back out the coprocessor kernel if bugs are ever discovered. Because these cryptographic checksums are kept in secure RAM, the coprocessor kernel may update them as newer kernels are released.

6.3. Hardware-level Maintenance

So far, I have discussed only software maintenance. Because secure coprocessors contain critical data, we need to also support hardware maintenance related functions. We may want secure coprocessors to perform self-tests while otherwise idle, and generate warnings if any transient errors are detected (e.g., correctable memory ECC errors, encryption hardware self-test errors, etc), as well as permit periodic checkup maintenance testing requiring suspension of the coprocessors' normal operations.

Such maintenance access to the internals of a secure coprocessor, while only logical and not physical, requires complete access to the secure coprocessor's state. Self-tests necessarily may require destructive writes to secure RAM; even though such self-tests are vendor supplied, we would like to prevent self-test code from accessing private or integrity-protected user data. This poses a dilemma: the secure coprocessor state seemingly cannot be backed up, since this permits replay attacks for applications such as electronic currency.[41] Secrets stored in secure RAM must remain private.

We can securely back up secure coprocessor state for maintenance testing and also transfer the state of one secure coprocessor to a replacement secure coprocessor. The trick

[39] Alternatively, we can use tamper-protected EEPROM to store the secondary boot loader to optimize for speed. See section 4.2.3 for a discussion of its security properties.

[40] If we store the secondary boot loader in protected EEPROM, we can omit the loading/validation steps.

[41] The attackers back up the state of their secure coprocessor, spend some electronic currency, and restore the previous state. See section 3.4.

is to use atomic transactions: state information is transactionally transferred from the source secure coprocessor to a target secure coprocessor. Most of the secure RAM of source the secure coprocessor is erased as a result of the transactional transfer. The only secure RAM contents not erased are the unique authentication and public key. This is required if the secure coprocessor is to be reused, since new code could not be loaded otherwise.

Dyad uses a simplified version of the traditional two-phase commit protocol [33, 53], since only two parties are involved and the write locks can be implicit.[42] The secure coprocessor transfer commit protocol requires an acknowledgement message from the target coprocessor after the source secure coprocessor (the transaction coordinator) sends the "commit" (or "abort") message, since the source secure coprocessor log (held in the secure RAM) will be forcibly truncated as a result of the transfer.

Note that the target secure coprocessor does not have to actually store all the source state information in its secure RAM: if all secure coprocessors have the same capacity, it will not have enough secure RAM. Fortunately, the state information only needs to be *logically* transferred to the target coprocessor — the target secure coprocessor can simply encrypt the state data, write it to disk, and save just the key in its secure RAM. As a optimization, the encryption and storage of the state data can be performed entirely by the source secure coprocessor; only the key needs to be transactionally transferred to the back up secure coprocessor.

After the state transfer is completed and secure RAM erased, testing may proceed. The secondary bootstrap code may now load in whatever vendor-supplied self-test code is needed, since this self-test code will not have any access to secret or integrity-protected user data. When the testing is done, we can restart the secure coprocessor (or a new one) and transactionally reload the original secure RAM state. Because state is always transferred and never copied, such back ups are not subject to replay attacks, and the testing provides users with assurance against hardware faults.

6.4. Tolerating Hardware Faults

At first glance, it would appear that by keeping secrets only in secure coprocessors, we face the risk of losing those secrets when secure coprocessor has a hardware failure. Fortunately, by applying a modified quorum consensus technique [37, 38], we can make a secure coprocessor system fault tolerant. We assume a failstop model [82].

An example of such a configuration would use three secure coprocessors in a group, all of which maintain the same secure data. Every update transaction involves two of the three with a secure timestamp [90, 89], so the secure data should remain identical between

[42] In the first phase, the transaction coordinator asks whether all entities involved in the transaction agrees that they are able to commit and have logged the appropriate data to stable storage. After a party has agreed that it is willing to commit, all values involved in the transaction are inaccessible until the coordinator declares a "commit" or "abort." The coordinator broadcasts "commit" or "abort" during the second phase, and transactional modifications to values become permanent or vanish.

transactions. Communication among the three coprocessors are encrypted. When a secure coprocessor fails, a new one is added (replacing the broken one) by being initialized from the most up-to-date of the remaining two secure coprocessors, simultaneously updating the group's group membership list. This update is performed transactionally, using a state transfer mechanism like the method described in section 6.3. If two or more coprocessors simultaneously fail, however, the data is unrecoverable. (Otherwise an attacker could separate a working trio of secure coprocessors into three groups of isolated coprocessors and use that to duplicate currency.) After regenerating to a triad of secure coprocessors, the failed coprocessor will be shunned by the regenerated group if it becomes operational again: attackers cannot create a new quorum by faking coprocessor failures.

In general, the number of failures F that can be tolerated can be made arbitrarily large by using more secure coprocessors in a group. Let there be N secure coprocessors in a group. Writes to secure data are considered successful if W secure coprocessors updates their copy of the secure data, and reads from secure data are considered to have obtained valid data only if R secure coprocessors in the group respond with (time stamped) data. Dyad allows failure-recovery restores to new secure coprocessors to proceed only if there are at least ρ working secure coprocessors, where F, R, N, W, and ρ satisfy the equations

$$R + W > N + F \tag{6.1}$$

$$\rho > \frac{N}{2} \tag{6.2}$$

$$\rho \geq R \tag{6.3}$$

Equation 6.1 is the standard requirement for the number of readers and writers to overlap (pigeon hole principle) from quorum consensus. Equation 6.2 requires that at least half of the coprocessors are available for regenerating the missing (and presumed dead) members of a group — preventing smaller partitions from being used to clone money or other access capabilities. Equation 6.3 ensure that the subset of our secure coprocessor group from which regenerate missing ones will contain at least one coprocessor containing the correct data, which can be propagated to the other coprocessors as part of the recovery/regeneration process, preserving the reader/writer overlap invariance for the regenerated coprocessor group. As part of the regeneration transaction, group membership is updated to contain only the secure coprocessors in the regeneration transaction.

This technique is a simple modification of quorum consensus for fault tolerance and security under the secure coprocessor framework. By combining secret sharing [76, 86] with quorum consensus [39], replication space requirements can be reduced.

Another approach would be to adapt striping to secure coprocessors, distributing data and error correction bits among several secure coprocessors. (Also see information on RAID [64].) This requires that every logical write to the secure data result in an atomic set of writes to secure coprocessors within the group, with data transmission among secure coprocessors encrypted. Recovery of data due to a failed secure coprocessor would operate in the same fashion as in classic striped systems, with the replacement coprocessor initialized via a transactional state transfer so it will possess the encryption keys necessary to communicate with its peers.

Using multiple secure coprocessors dramatically reduces the likelihood of critical data being lost due to hardware failures. This enables the use of secure coprocessor technology for large scale and high reliability applications. I also eliminate the possibility that a single hardware failure would preclude properly licensed programs from running.

Chapter 7

Verification and Potential Failures

Security critical systems are not just vulnerable to hardware-level attacks and simple hardware faults; the delivered hardware might have been substituted with bogus, trojan-horse hardware, and the system software may contain bugs. This chapter explains how users can verify secure coprocessor hardware, and shows how the secure coprocessor system design helps isolate the effects of software faults and check software. Additionally, this chapter analyzes the consequences of potential failures in the system and identifies the degree of trust that must be placed on hardware and system software vendors

7.1. Hardware Verification

The self-tests that I considered in section 6.3 are vendor-provided executables. Suppose we wish to verify that the secure coprocessor or system software vendor is not supplying us with bogus secure coprocessor hardware. Can some form of testing be performed?

By modifying the self-test procedure, we can perform limited statistical checking of secure coprocessor hardware. To verify that the hardware originated from the proper hardware vendor, the local system administrators or security officers may reset a fraction of the secure coprocessors and load in hardware verification software in lieu of a secondary bootstrap loader. This permits arbitrary secure coprocessor hardware testing code to be loaded. While sophisticated bogus hardware could be made to operate identically to a real secure coprocessor under most conditions, this software probing can, coupled with gross hardware verification (e.g., verifying the X-ray image of the circuit board and other physical controls), provide us with strong assurances that the secure secure coprocessor hardware is genuine.

Note that this testing is quasi-destructive, since the authentication secrets stored by the coprocessor system software vendor are lost. These coprocessors may, however, be returned to the system software vendor to be reinitialized with a new set of authentication secrets. Additional destructive testing of secure coprocessors may be performed on a spot-check basis for greater assurances of the authenticity of the secure coprocessor hardware.

7.2. System Software Verification

Having the secure coprocessor security kernel provide logical security (basic peer-to-peer authentication, encrypted communication channels, and private address spaces) is central to being able to run secure applications within a secure coprocessor. While any absolute proof of correctness of security kernels is outside of the scope of this thesis and such proofs will not be feasible for a long time (if ever), we must have some assurance of the security of secure coprocessor system software.

In the Dyad system, the Mach 3.0 kernel runs in the secure coprocessor. It is a small security kernel with capability-based interprocess communication, configured with only a few device drivers necessary for communicating with the host system. Because the kernel code is cryptographically fingerprinted by the system software vendor and not encrypted, the code may be independently inspected. Though failstop bugs in the coprocessor kernel would not permit disclosure of secrets, it remains to be shown whether the system design can minimize the amount of damage caused by other kinds of kernel bugs.

The system design isolates security-critical portions of the coprocessor security kernel, reduces the impact of bugs, and makes analysis easier.

I assume that the kernel provides private address spaces using the underlying virtual memory hardware, a very stable technology. I also assume that the secure applications do not intentionally reveal their own secrets, whether explicitly or through covert channels. Furthermore, I assume that bugs in one part of the kernel do not have far-reaching effects, e.g., permit user-level code to arbitrarily modify another part of the kernel.

Dyad uses *security checkpoints* to minimize the impact of bugs in the rest of the system. These are the security critical portions of the kernel that must bear close inspection. Fortunately, there are only a few modules controlling I/O between the host system and the secure coprocessor (the port and DMA drivers) and access to the secure RAM (the `iopl` interface for accessing the secure RAM and the `sec_ram` server — see section 4.2.3). These security-critical modules are well isolated, and provide an opportunity for carefully controlling and checking data flow, simplifying the code inspection task.

The example of crypto-paging illustrates how testing is simplified. Instead of looking at the code for the default pager, we simply make sure that encryption is turned on whenever we use the DMA interface on the default pager's behalf. Similarly, for access control to the secure RAM, the `iopl` interface allows only a privileged client (the `sec_mem` server) to map in the secure RAM into the client's address space, and the `sec_mem` server provides access control among the secure applications. The secure RAM's physical address range is not otherwise known to the kernel, and the virtual memory subsystem could not accidentally provide access to it unless the memory mapping entries are copied from the `sec_mem` server's address map. If we do not want to trust the virtual memory subsystem to prevent unauthorized access, we could provide a trivial device driver performing physical I/O only to the address range of the secure RAM with exclusive use by the `sec_mem` server. The `sec_mem` server code, of course, must also be carefully scrutinized to only

give access to appropriate portions of the secure RAM as part of the cryptographic loading of a secure application's code.

Because Dyad has simple secure memory interfaces and host interface, it is possible to focus on the security properties of the code implementing these interfaces. Rigorously checking this code decreases the likelihood that bugs in the Mach kernel could cause secret data disclosure. While this does not replace rigorous correctness proofs of the kernel code, we can increase our confidence that kernel bugs will not cause catastrophic disclosure of secrets.

7.3. Failure Modes

An ideal distributed security system would never fail, but any serious design must take failures into account. In this section, I discuss the potential failure modes of the Dyad system and examine the risks involved.

I identify the potential sources of security breaches and consider their impacts. There are several secrets crucial to the correct operation of the overall system and their disclosure would have a severe impact on the system. Some of these reside only within software manufacturers' facilities, and others are also kept in secure coprocessors in the field.

The most critical secret in the system is the secure-coprocessor software private key. This key is created at the system software manufacturing facilities and produces a digitally signed certificate for every new coprocessor, each certifying the public key and authentication puzzle as belonging to a secure coprocessor identity created by that manufacturer. The corresponding private key and secret authentication puzzle solution are loaded into the secure memory as part of the system software installation, along with the certificate.

Disclosure of the system software manufacturer's signature key permits attackers to create fake secure coprocessors, and these *un*secure coprocessors or software emulations can totally compromise the system.

In a similar fashion, if attackers possess the secret key and authentication puzzle solution of a secure coprocessor, they can obtain any application-specific secrets associated with secure applications subsequently installed on that coprocessor.[43] Furthermore, attackers will also be able to expose secrets stored in other secure coprocessors they manage, since they can use an unsecure coprocessor as a transactional state transfer target.

Coprocessor-specific secrets are only vulnerable to exposure between the time of generation and the time of installation; by my main axiom, it is impossible to to obtain secrets after they are installed in a secure coprocessor. Additional security can optionally be obtained by requiring authorization (perhaps from the system software vendor) before engaging in transactional state transfers.

One particularly security sensitive application is electronic currency, and it is important to discuss how disclosures of critical secrets will compromise the system. The critical

[43] If the attacker had logged the previous installation of secure applications, those application-specific secrets (and the privacy of the texts of programs themselves) are also endangered.

data is the electronic currency application authentication puzzle solution. Disclosure of this information permits creation of electronic cash, if access to the secure channel between secure coprocessors can be achieved. Since having access to the individual secure coprocessor secrets implies access to the application secrets, one method of increasing the work required to attack the system is to have the electronic currency application use secure channels provided by the secure coprocessor kernel (perhaps with doubly encrypting using application-level keys as well). The kernel performs coprocessor-to-coprocessor key exchange using the individual secure coprocessor secrets. This forces attackers to obtain access to individual secure coprocessor secrets rather than just the application secrets.

Further application-specific limits can limit the amount of the damage. In the case of electronic currency, the electronic currency application can limit the total amount of electronic currency that may be stored within a secure coprocessor. This limitation reduces the risk of losing money as a result of catastrophic hardware failure, and also reduces the rate at which fake electronic currency may be introduced into the system if secrets are compromised. Additional limits may be added to restrict the rate at which electronic funds can be transferred, though this only serves as a tourniquet and cannot solve the problems of compromised secret keys.

Similar problems occur if the underlying cryptographic system is broken. The intractability of factoring large moduli is basic to both the authentication and public key systems. If a modulus used in a cryptographic algorithm is factored, secrets would be similarly revealed. This problem is endemic to cryptographic applications in general.

7.4. Previous Work

Previous work on system isolation include fences [69] which introduced the idea of using cryptographic checks to find system errors. Trusted computing bases form an important part of the "Orange Book" Trusted Computer System Evaluation Criteria [101]. Trusted computing bases rely on a strict security boundary between the secure environment and the unsecure environment — all the computer hardware and software, including the terminals, are considered secure, and the users are not. The system software implements the appropriate access control, often mandatory, to enforce policy.

Chapter 8

Performance

This chapter discusses Dyad's performance. First, I examine the implementations of my authentication and fingerprinting algorithms. Next, I look at the overhead of crypto-paging relative to simple paging.

8.1. Cryptographic Algorithms

This section gives timing figures for my implementation of the authentication algorithm and fingerprinting algorithm described in chapter 5. Because the Citadel unit is a research prototype and its processor will be updated to newer, faster RISC processor, my timing figures are for several processors: an i386SX processor running at 16 MHz; an i486DX2/66 processor; a MIPS R3000 processor; a Power processor for an IBM RS6000/950; and projected figures for a 601 PowerPC. Table 8.1 shows running times for the basic authentication algorithm and the processing rates for the fingerprint algorithm on these processors.

The Citadel processor requires 3.45 seconds to perform zero knowledge two-way authentication (see section 5.1.4) to achieve a security factor of 3.18×10^{28}, using a 150 decimal digit modulus.[44] To perform the authentication, Citadel and the host processor (which provides networking for the secure coprocessor) must exchange 4 messages. The overhead for sending a message between Citadel and the host processor is approximately 0.96 S; much of this overhead should disappear if the device drivers in the host and in the Citadel-side Dyad kernel did not need to poll hardware status. (See section 4.2.1 for a discussion of the source of this overhead.) We anticipate dramatic improvement in authentication time in the next generation of the Dyad hardware base.

The fastest fingerprinting implementation (see section 5.1.5) running on the Citadel's i386SX fingerprints at 410 Kbytes/sec. This assembler-coded routine uses a 65536-entry table (2^{16}) of precomputed partial residues. This code run at the maximum possible memory bandwidth: for comparison, a tight assembler loop loading source memory into a register reads memory at a rate of less than 1.1 Mbytes/sec on the Citadel, and the fingerprint table look-up code reads two additional memory words per word of input data. Because the i386SX has no on-chip cache and the Citadel board provides no external cache memory,

[44] Factoring such a modulus should require approximately 20000 MIPS-years of CPU time using contemporary (May 1994) factoring techniques [50, 87].

Algorithm	i386SX (16 MHz)	MIPS R3000 (20 MHz)	i486DX2/66	RS6000/950	PowerPC 601 (66 MHz)
Authentication (mS)	3450	249	167	114	86.0 est.
Fingerprint (MB/S)	0.410	1.14	1.42	3.99	2.70 est.

Table 8.1 Cryptographic Algorithms Run Time

Because the Citadel prototype coprocessor is a research prototype, its processor, a i386SX running at 16 MHz, is likely to be upgraded to a newer, faster processor when secure coprocessors become commercial products. To obtain these run times for non-Citadel processors, I ran the portable C-language implementations of these algorithms on test data on commercially available PCs and workstations (a DECstation 5000/200, an Elite 486 PC, and an IBM RS6000/950); the times for the PowerPC 601 is extrapolated from its SPECint ratings.

some of the memory bandwidth is expended fetching instructions. A more space-efficient assembler language implementation uses a much smaller 256-entry table and fingerprints at 226 Kbytes/sec, or about 55% of the speed of the first implementation. On Citadel, the most tightly tuned C language implementations of the fingerprint algorithm achieve only 224 Kbytes/sec and 204 Kbytes/sec for large and small tables respectively, largely because of the inability of the compiler to avoid register spills into memory and to optimally use (and in some cases, even generate) some i386 instructions.

(The residue table initialization algorithm is described in section 5.2.5. For the large table, the time required is approximately 1.23 S; the time for the small table is negligible (42 mS). Note that this is a one-time charge.)

My experiments recommend that the smaller assembler coded version be used for most cases. The large table version is useful where the same irreducible polynomial is used for a large amount of data (perhaps when checking disk contents); the small version wins when the irreducible polynomial is changed often, or where there are tight real-memory requirements (such as in the Citadel prototype). When cache memory is added to future generations of Citadel, the smaller-table version will gain in performance relative to the larger-table version because the table of partial residues should easily fit within the cache.

Smaller code size is desirable for security code. When the code is smaller, the system is easier to verify and less likely to contain bugs. The key exchange routines consists of 80 lines of C code. The authentication routines consists of 75 lines of C code. Both the key exchange and the authentication code are written on top of a library of routines for calculating with arbitrarily large integers. The fingerprinting code consists of 211 lines of

C code and 160 lines of i386 assembler. My total core routines are relatively small: 366 lines of C code and 160 lines of assembler.

8.2. Crypto-Paging

The overhead for crypto-paging is unmeasurable, since both crypto-paging and normal paging activity go through the hardware DES machinery and DMA channels. Overhead only incurs when the encryption keys are set. This happens every time a page is written out to host memory, where a (small) encrypted system disk image resides.

Additionally, the host system imposes limits on the number of pages that can be transferred, since we cannot guarantee that the disk image will reside in physically contiguous memory. This means that if paging was not encrypted, the number of bytes copied per DMA transfer is most likely to be a single virtual memory page (4K) anyway, and the currently high per-DMA-transfer overhead (0.96 S) cannot be amortized over many pages of memory.

Chapter 9

Conclusion and Future Work

The problem of providing security properties for widely distributed systems is a difficult one that must be solved before applications with strong privacy and integrity demands, such as electronic commerce applications, can be safely deployed. All cryptographic protocols require secrets to be kept, and all access control software assume the ability to maintain the integrity of the access control database. These assumptions must be satisfied in any serious secure distributed system; providing these security properties in a rigorous and complete way is impossible without some form of physically secure hardware.

In this thesis I have shown that it is possible to provide very strong security guarantees without putting the entire computer in a locked room. By adding secure coprocessors to normal workstations or PCs, overall security may be bootstrapped from a core set of security properties guaranteed by secure coprocessor hardware. Cryptographic techniques to check integrity and to protect privacy can provide much stronger system-level security guarantees can be provided than were previously possible.

Furthermore, by applying transaction processing techniques to security, I built electronic currency systems where money cannot be created or destroyed accidentally. By using quorum consensus and transactions, I designed fault tolerant secure coprocessor systems.

I have analyzed the native security properties of various components of the software/hardware system, and arranged them into a security hierarchy; furthermore, I used cryptographic techniques to enhance security properties. This separation of the system architectural components by their security properties permit secure-system designers to reason realistically about what kinds of security properties are actually achievable.

The contributions of this thesis may be summarized as follows:

- end-to-end analysis of the security properties of the system components, both at the hardware level and at the software level;

- design and analysis of combined hardware-software architecture for bootstrapping security guarantees throughout the system, using cryptographic techniques at the system component boundaries (including crypto-paging and crypto-sealing);

- demonstration of the feasibility of the architecture by constructing a working prototype system, providing insights into system design issues that restrict the overall system architecture;

- design, analysis, implemention, and measurement of cryptographic protocols for zero-knowledge authentication and key exchange, suitable for use in security critical environments;
- demonstrating that secure coprocessors may be statistically checked against vendor fraud;
- showing how secure coprocessors may be operated in a fault-tolerant manner; and
- designing solutions to exemplar electronic commerce applications, including building an electronic currency application and analyzing how cryptographic stamps may be used.

Secure coprocessors exist today and can solve many pressing distributed security problems, but there remains several challenges to be solved by future developers of secure coprocessor technology. The need for a general, low-cost distributed transaction system is apparent, and it remains to be shown that one can be built to run efficiently within the secure coprocessor environment. Tools for automating the task of splitting applications are need, and the issue of providing operating system support for split secure-coprocessor applications remains to be fully explored. Most importantly, many secure applications building on secure coprocessors remain to be discovered.

Bibliography

[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] Mike Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. Mach: A new kernel foundation for Unix development. In *Proceedings of Summer USENIX*, July 1986.

[3] L. Adleman, K. Manders, and G. Miller. On taking roots in finite fields. In *Proceedings of the Nineteenth IEEE Symposium on Foundations of Computer Science*, pages 715–177, October 1977.

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

[5] 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.

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

[7] E. R. Berlekamp. Factoring polynomials over large fields. *Mathematics of Computation*, 24(111):713–735, 1970.

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

[9] Blum, Blum, and Shub. Comparison of two pseudorandom number generators. *Advances in Cryptology: CRYPTO-82*, pages 61–79, 1983.

[10] 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.

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

[12] Stefan Brand. An efficient off-line electronic cash system based on the representation problem. Technical Report CS-R9323, Centrum voor Wiskunde en Informatica, 1993.

[13] Gilles Brassard. Modern cryptology: A tutorial. In *Lecture Notes in Computer Science*, volume 325. Springer-Verlag, 1985.

[14] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. In *Journal of Computer and System Sciences*, pages 156–189, October 1988.

[15] Julius Cæsar. *Cæsar's Gallic Wars*. Scott, Foresman and Company, 1935.

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

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

[18] U. S. Internal Revenue Code. Internal revenue code volume 1, 1993.

[19] U. S. Legal Code. 1989 Amendments to the Omnibus Crime Control and Safe Street Act of 1968, Public Law 101-162. United States Legal Code, U. S. Government Printing Office, 1989.

[20] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, 1993.

[21] I. Damgard. On the existence of bit commitment schemes and zero-knowledge proofs. In *Lecture Notes in Computer Science*, volume 325. Springer-Verlag, 1985.

[22] C. J. Date. *An Introduction to Database Systems Volume 2*. Addison-Wesley, Reading, MA, 1983.

[23] Peter J Denning. *Computers Under Attack: Intruders, Worms, and Viruses*. ACM Press, New York, N.Y., 1990.

[24] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, NY, 1986.

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

[26] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

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

[28] Edward W Felton and John Zahorjan. Issues in the implementation of a remote memory paging system. Technical Report 91-03-09, University of Washington, 1991.

[29] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, May 1987.

[30] 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.

[31] 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.

[32] Vice President Al Gore. Speech at the National Press Club, December 1993.

[33] James N. Gray. A transaction model. Technical Report RJ2895, IBM Research Laboratory, San Jose, California, August 1980.

[34] James N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Very Large Database Conference*, pages 144–154, September 1981.

[35] Louis Claude Guillou, Michel Ugon, , and Jean-Jacques Quisquater. The smart card : A standardized security device dedicated to public cryptology. In Gustavus J Simmons, editor, *Contemporary cryptology : The science of information integrity*. IEEE Press, Piscataway, NJ, 1992.

[36] M. Herlihy and J. D. Tygar. Capabilities without a trusted kernel. In A. Avizienis and J. Laprie, editors, *Dependable Computing for Critical Applications*. Springer-Verlag, 1991.

[37] Maurice P. Herlihy. General quorum consensus: A replication method for abstract data types. Technical Report CMU-CS-84-164, Carnegie Mellon University, December 1984.

[38] Maurice P. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1), February 1986.

[39] 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*.

[40] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop of Micro-Kernels and Other Kernel Architectures*, April 1992.

[41] IBM Corporation. *Common Cryptographic Architecture: Cryptographic Application Programming Interface Reference*, SC40-1675-1 edition.

[42] Stuart Itkin and Josephine Martell. A PDF417 primer: A guide to understanding second generation bar codes and portable data files. Technical Report Monograph 8, Symbol Technologies, April 1992.

[43] A. Longacre Jr. Stacked bar code symbologies. *Identification Journal*, 11(1):12–14, January/February 1989.

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

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

[46] Stephen Thomas Kent. *Protecting Externally Supplied Software in Small Computers*. PhD thesis, Massachusetts Institute of Technology, September 1980.

[47] Gene H. Kim and Eugene H Spafford. The design and implementation of tripwire: A file system integrity checker. Technical Report CSD-TR-93-071, COAST Laboratory, Department of Computer Science, Purdue University, 1993.

[48] Donald Ervin Knuth. *The Art of Computer Programming, Volume 2*. Addison-Wesley, Reading, MA, 2 edition, 1968.

[49] Samuel J. Leffler, Marshall K. McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, 1989.

[50] A. K. Lenstra, Jr. H. W. Lenstra, M. .S. Manasse, and J. M. Pollard. The number field sieve. In *Proceedings of the 22nd ACM Symposium on the Theory of Computing*, pages 564–572, 1990.

[51] R. Lipton. Personal communication.

[52] Steven Low, Nicholas F. Maxemchuk, and Sanjoy Paul. Anonymous credit cards. Technical report, AT&T Bell Laboratories, 1993. Submitted to *IEEE Symposium on Security and Privacy*, 1993.

[53] Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, CA, 1994.

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

[55] Brock N Meeks. The end of privacy. *WIRED*, 2(04):40–50, April 1994.

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

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

[58] National Semiconductor, Inc. iPower chip technology press release, February 1994.

[59] 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.

[60] Ivan N. Niven, Herbert S. Zuckerman, and Hugh L. Montgomery. *An Introduction to the Theory of Numbers*. Wiley, New York, 5 edition, 1991.

[61] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 514–523, May 1990.

[62] R. C. Palmer. *The Bar-Code Book*. Helmers Publishing, 1989.

[63] José Pastor. CRYPTOPOST: A universal information based franking system for automated mail processing. *USPS Advanced Technology Conference Proceedings*, 1990.

[64] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of ACM SIGMOD '88*, pages 109–116, 1988.

[65] Theo Pavlidis, Jerome Swartz, and Ynjiun P. Wang. Fundamentals of bar code information theory. *Computer*, 23(4):74–86, April 1990.

[66] Theo Pavlidis, Jerome Swartz, and Ynjiun P. Wang. Information encoding with two-dimensional bar codes. *Computer*, 24(6):18–28, June 1992.

[67] 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.

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

[69] 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.

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

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

[72] Michael O Rabin. How to exchange secrets by oblivious transfer. Technical report, Harvard Center for Research in Computer Technology, 1981.

[73] Michael O. Rabin. Transaction protection by beacons. Technical Report TR-28-81, Harvard Center for Research in Computer Technology, 1981.

[74] Michael O. Rabin, 1987. Personal Communication.

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

[76] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, May 1989.

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

[78] 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.

[79] R Sandberg. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1985 USENIX Technical Conference*, pages 119 – 130. USENIX, June 1985.

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

[81] M. Satyanarayanan. Distributed file systems. In S. Mullender, editor, *Distributed Systems*. Addison-Wesley and ACM Press, second edition, 1993.

[82] Fred B Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

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

[84] U. S. Postal Service. Annual report of the postmaster general, fiscal year 1991.

[85] U. S. Postal Service and U. K. Royal Mail. Personal communications.

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

[87] R. D. Silverman. The multiple-polynomial quadratic sieve. *Mathematics of Computation*, 48(177):329–339, 1987.

[88] Marvin Sirbu. Internet billing service design and prototype implementation. In *IMA Intellectual Property Project Proceedings*, number 1 in 1, 1994.

[89] Sean Smith, David B. Johnson, and J. D. Tygar. Asynchronous optimistic rollback recovery using secure distributed time. Technical Report CMU-CS-94-130, Carnegie Mellon University, March 1994.

[90] Sean Smith and J. D. Tygar. Secure and privacy for partial order time. Technical Report CMU-CS-94-135, Carnegie Mellon University, April 1994.

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

[92] Richard Stallman. *Gnu-emacs Manual*.

[93] 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.

[94] Clifford Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage*. Doubleday, New York, 1989.

[95] Ken Thompson. Reflections on trusting trust. In Robert L. Ashenhurst and Susan Graham, editors, *ACM Turing Award Lectures, The First Twenty Years, 1966 – 1985*, pages 163–170. Addison-Wesley, 1987.

[96] Theodore T. Tool. MIT guide to lock picking. Distributed anonymously, 1987.

[97] J. D. Tygar and B. S. Yee. Strongbox. In Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors, *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[98] 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.

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

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

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

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

[103] 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.

[104] 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.

[105] 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.

[106] 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.

[107] W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[108] Michael Wayne Young, Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David L. Black, and Robert V. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating System Principles*, pages 63–76. ACM, November 1987.