Secure Coprocessors in Electronic Commerce Applications

Bennet Yee, Microsoft Corporation
J. D. Tygar, Carnegie Mellon University

Secure Coprocessors in Electronic Commerce Applications

Bennet Yee
Microsoft Corporation
Redmond, WA 98052
bsy@microsoft.com

J. D. Tygar
Carnegie Mellon University
Pittsburgh, PA 15213
tygar@cs.cmu.edu

Abstract

Many researchers believe electronic wallets (secure storage devices that maintain account balances) are the solution to electronic commerce challenges. This paper argues for a more powerful model — a secure coprocessor — that can run a small operating system, run application programs, and also keep secure storage for cryptographic keys and balance information.

We have built a system called Dyad, on top of a port of the Mach 3.0 microkernel to the IBM Citadel secure coprocessor. This paper describes the abstract architecture of Dyad and a general discussion of secure coprocessor implementations of a variety of electronic commerce applications:

- *Copy protection for software*
- *Electronic cash (including a critique of proposed solutions for point-of-sale electronic wallet systems)*
- *Electronic contracts*
- *Secure postage*

1 Introduction

Many researchers believe electronic wallets (secure storage devices that maintain account balances) are the solution to electronic commerce challenges [3, 31]. This paper argues for a more powerful model — a secure *coprocessor* — that can run a small operating system, run application programs, and also keep secure storage for cryptographic keys and balance information.

Secure coprocessors are tamper-proof sealed devices that have a processor, memory storage, and (optional) fast crypto-support. They are protected in that any attempt to penetrate them will result in all critical memory being erased.

Secure coprocessors have a number of advantages over electronic wallets. They have more powerful processors and larger amounts of memory, so they can do more: they can negotiate (and enforce) contracts that involve renting and redistributing intellectual information. If they incorporate a *secure display*, they can provide much greater protection for customers in point-of-sale applications. They can provide software copy protection, permitting much more general applications.

We have built a system called Dyad, on top of a port of the Mach 3.0 microkernel [19] to the IBM Citadel secure coprocessor [57]. This paper describes the abstract architecture of Dyad and a general discussion of secure coprocessor implementations of a variety of electronic commerce applications:

- Copy protection for software
- Electronic cash (including a critique of proposed solutions for point-of-sale electronic wallet systems)
- Electronic contracts
- Secure postage

A few years ago, only experimental prototypes of secure coprocessors existed, but today, the market is filling up. Manufacturers such as Cylink, IBM, National Semiconductor, Spyrus, Telequip, and others have announced secure coprocessor products. Several other major manufacturers will announce significant new products in the near future. There is a new FIPS standard for cryptographic modules (including secure coprocessors) [53] and

This work was supported in part by ARPA contracts F33615-90-C-1465 and F33615-93-1-1330, NSF Presidential Young Investigator Award CCR-9958087, matching funds from Motorola and TRW, a contract from the US Postal Service, and by an equipment grant from IBM. This work was done while the first author was at Carnegie Mellon University. This work is the opinion of the authors and does not necessarily represent the view of their employers, funding sponsors, or the US Government.

a new service to perform those evaluations [54]. This paper attempts to lay out the intellectual issues in the use of these new secure coprocessor products.

Because of length considerations, this paper does not discuss our Dyad implementation (see [60]), or additional applications of secure coprocessors arising from distributed computation (see [43, 44].)

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. (If the shield is somehow penetrated, then a secure coprocessor erases all critical memory.) 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.

2.1 Security Properties of Secure Coprocessors

All security systems rely on a nucleus of assumptions. For example, it is often assumed that encryption systems are resistant to cryptanalysis. Similarly, we 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 our physical security axiom by expending enormous resources (possibly feasible for very large corporations or government agencies). We rely on a physical work-factor argument to justify our axiom, similar in spirit to intractability assumptions of cryptography. Our secure coprocessor model does not depend on the particular technology used to satisfy the work-factor assumption. Just as cryptographic schemes may be scaled 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.

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.

2.2 Potential Platforms

Several physically secure processors exist, and more are forthcoming. Some of these are very secure, satisfying the highest security level specified by FIPS PUB 140-1 [53]. (This publication gives four security levels for cryptographic modules, including secure coprocessors.) Announced secure coprocessors include the μABYSS [55] and Citadel [57] systems from IBM, the iPower [34] encryption card by National Semiconductor, some extended implementations of the Clipper and Capstone systems [2, 51, 52] proposed by the NSA as DES replacements, the Crypta Plus [47] encryption card by Telequip, the CY512i [13] chip from Cylink, and various smartcard systems such as some GEMPlus or Mondex cards [22]. There will be additional announcements of systems with increased processing power from major vendors in the next few months. For a fuller descriptions of potential platforms, see [60].

3 Applications

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

3.1 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 in [60], requires the use of a secure coprocessor.

Software copy protection is complementary to electronic commerce. Without copy protection, rental or per-use charging of software is not possible. Here we discuss tradeoffs in using a secure coprocessor to implement software copy protection. (Dyad includes a software protection mechanism [60].)

3.1.1 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,[1] 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.

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 4.3 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 [56], White and Comerford show how such a partitioning should be done to maximize the difficulty of reverse engineering the secure portion of the application.[2]

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.

3.1.2 Previous Work

A more primitive version of the copy protection application for secure coprocessors appeared in [28, 56]; a secure-CPU approach using oblivious memory references (i.e., apparently random patterns of memory accesses) giving a poly-logarithmic slow down, appears in [18] and [35].

3.2 Electronic Currency

We 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

[1] 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. Secure coprocessors may be used in a fault-tolerant fashion; see [60].

[2] We also examined a real application, gnu-emacs 19.22 [45], to show how it could be partitioned to run partially within a secure coprocessor. The X Windows display code 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 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.

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.

We argue that secure coprocessors can not only support electronic wallet functionality, but that they also offer stronger guarantees than existing and proposed electronic wallets. In particular, electronic coprocessors offer consumer protection unavailable with existing electronic wallets. (We have built an electronic currency mechanism on top of Dyad, see [60].)

3.2.1 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 [20, 21]:

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

The first electronic money model is based on the cash analogy. In this model, 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 [10], which lacks transactional properties such as failure atomicity — see section 3.2.3), 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.

We 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 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 [7, 15, 58] 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. Furthermore, 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 4.3). 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.[3]

Secure coprocessor mediated electronic currency transfer is analogous to rights transfer (not to be confused with rights copying) in a capability-based protection system [59]. Using the electronic money — e.g., spending it when running a pay-per-use program — is analogous to the revocation of a capability. This type of model relies on the idea of secure-coprocessor-protected unforgeable electronic tokens. In addition to electronic money, these unforgeable tokens are useful for many other applications. Electronic tokens can be 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. However, the trusted electronic currency manager running in the secure coprocessorcan use 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.)

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 [46], if properly implemented [4], 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.

[3] Newer smartcards such as GEMPlus or Mondex cards [22] feature limited physical security protection, though the types of attacks these cards can withstand have not been published.

3.2.2 Point-of-Sale Terminals

In addition to their use in networked computers, secure coprocessors can be used for commercial transactions at point-of-sale terminals. For this application, we would need portable secure coprocessor form factors, such as smartcards or PCMCIA cards. Unlike the networked PC scenario where the users can be familiar with particular PCs they use, customers at a point-of-sale terminal have no reason to trust its integrity.

Point-of-sale use of secure coprocessors is vulnerable to a very important class of threats: communication spoofing between the secure coprocessor and the user. This problem arises because there is no private communications path [50] between the user and the secure coprocessor. A *secure display* only displays data to the user originating from the secure coprocessor and guarantees that the displayed data can not be tapped by a third party; such a display would provide secure one-way communication from the secure coprocessor and the user.

Today's smartcards and PCMCIA cards do not incorporate secure displays. Thus, for point-of-sale use, the user must rely on the display on the point-of-sale terminal to inform him of the total price. Unlike traditional paper credit-card-imprint slips, a secure coprocessor's digital signature is on a document that is never shown to the user — whatever per-signature user authorization required is performed blind, and the secure coprocessor might sign a purchase order for a $10,000 gold watch when the point-of-sale terminal is displaying "$1.98 watch batteries." Furthermore, to prevent a user authorization replay attack, some method for securely transferring the user authentication/authorization input to the secure coprocessor is required.

To permit secure input of user passwords to a secure coprocessor and to display purchase information, a secure display suffices: we use the secure display as a one-way secure channel over which we transmit a one-time pad, i.e., a cryptographically random string. The user then uses the point-of-sale terminal's keyboard (perhaps via arrow keys) to modify the displayed string into the user's password. For example, if your password was "SHOELACE" and the displayed string was "QZKNCFLX", you would press the $\boxed{\uparrow}$ arrow twice to change the "Q" to an "S", and then press the $\boxed{\rightarrow}$ arrow to advance to the next character, etc. (This is an idea adapted from [1].) Price information can be shown on a secure display in the obvious way.

Without a secure display of purchase data and secure entry of passwords, point-of-sale use of secure coprocessors does *not* increase the security of point-of-sale commerce over existing credit card systems. One much touted property of using smartcards in lieu of mag-stripe credit cards is customer non-repudiation and the elimination of merchant fraud. However, while the cryptographic signature keys may be secure, smartcards without some form of secure display can not link the signature to the purchase due to the absence of customer review. Thus customers are still vulnerable to merchant fraud — rather than modifying the numbers on a credit card slip after the fact, the merchant can simply introduce a difference between data presented to the user and the users' secure coprocessor.

3.2.3 Previous Work

An alternative to the secure coprocessor managed electronic currency is Chaum's Digicash protocol [8, 10]. 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).[4]

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 [12], employee payments must be reported similarly for tax purposes [11], stock transfers must be

[4] 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.

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 class of electronic money system is server-based. NetBill [42] is one type of such a system. NetBill 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 true 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 [30] or anonymous message forwarders to protect against traffic analysis, at the cost of adding centralized servers back to the system.

3.3 Electronic Contracts

One of the most exciting applications of secure coprocessors is the use of electronic contracts. Electronic contracts are a natural extension to the "basic" electronic commerce approach. Where existing electronic commerce systems provide a basic, two-party contract which offered money for goods, a full electronic contract approach permits multi-party contracts, delegation, and a richer set of contractual primitives.

Electronic contracts provide enabling technology for creating electronic marketplaces [17]. Applications include the idea of superdistribution of software [33], and the creation of electronic futures markets.

In superdistribution, the idea is that the traditional software distribution channel is replaced by allowing a software buyer to resell the software on the manufacturer's behalf. When we look at this in the electronic contracts viewpoint, the customer is entering into a contract with the software manufacturer whereby the customer not only obtains the rights to use that software, but also the rights to make the same contract with other potential customers on the manufacturer's behalf. Such a self-replicating contract is a relatively simple three-party contract, where all of the contractual terms — electronic money transfer, rights to run a program, and making more electronic contracts — are enforceable by a secure coprocessor. See Figure 1 for an example superdistribution contract.

Having an expressive electronic contract language also enables the creation of electronic markets not previously possible. For example, air travel requirements — travel destination and approximate times — may be written up as an electronic contract containing the maximum price that the user is willing to pay, and this contract may be put up for auction. Travel agents bid for and buy the right (and obligation) to fulfill such contracts, increasing the efficiency of the travel market; additionally, travel agents may speculate on airline pricing and offer a higher bid in anticipation of fare reductions. Note, furthermore, that airline tickets may also be objects handled by the electronic contract system: these may simply be electronic documents signed by the airline giving the customer the right to travel on a particular flight, or even a token of a specific token type which permits travel on a certain flight.

In full generality, the objects referred to within electronic contracts will not always be objects that are managed by secure coprocessors, and this necessarily implies that external adjudication will be required when breaches of contracts occur. Furthermore, the user may not be able to satisfy the contractual demands, e.g., a broker who (speculatively) sells run-time on a mainframe may find all the cycles already allocated.

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

Unforgeable tokens are protected objects conserved by secure coprocessors; they are freely transferable, but can

```
software_distributor(signatory  id_t      manuf,
                     signatory id_t      distributor,
                     key_t               sw_key,
                     int                 manuf_profit,
                     id_t                prev_distr,
                     int                 prev_distr_cut,
                     time_t              expire)
{
        int     price;

        terminates when
                date() >= expire;

access(none):
        super_buy(id_t          buyer,
                  money_t       cash @ buyer)
        {
                int  profit;
                     /* profit for this distributor; no Amway tree */
                if (cash->amount < price) reject;
                     /* cannot sell at a loss */
                xfer(cash,manuf->in_register,manuf_profit);
                xfer(cash,prev_distr->in_register,prev_distr_cut);
                profit = price - manuf_profit - prev_distr_cut;
                xfer(cash,distributor->in_register,profit);
                xfer(sw_key,buyer,1);
                software_distributor(manuf,buyer,sw_key,manuf_profit,
                                     distributor,profit,expire);
                     /* to do Amway, we would pass profit up
                      * the distr chain rather than all at once here */
        }
access(distributor):
        set_price(int           new_price)
        {
                /* pricing must at least pay for manufacturer profit */
                if (new_price < manuf_profit + prev_dist_cut) reject;
                price = new_price;
                enable_access(super_buy,all);
        }
}
```

Figure 1: Software Superdistribution Contract

In this example, the software retail distributor enters into a contract with a software manufacturer, which enables the distributor to sell the software to customers for customer use and at the same time permit the customer to redistribute the software under the same contractual terms. For the duration of the contract, the distributor gains the power to make new contracts on the manufacturer's behalf.

be created and destroyed only by the agents that issued them (or their designees). Furthermore, the transfer of tokens occur in a transactional manner, so that the number of tokens is a conserved quantity (excepting explicit action by their issuer).

Tokens are useful for representing electronic currency and execute-only rights to a piece of software (much as in capability systems). In the case of rights such as execute-only rights, the token provides access to cryptographic keys that may be used (only) within the secure coprocessors to run code. Electronic currency and execution rights are subtypes of tokens and inherit the transactional transfer property from tokens.

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

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

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

Contractual clauses can require one of the parties to accept further contracts of certain types. One example of this is a requirement for action to be completed by a certain deadline, e.g., for a contractor to solve some problem or write some code before a project completion date. Another is a contract between a distributor and a software house, where the software house requires the distributor to accept sales contracts from users for upgrading a piece of software.

3.4 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* [9]), 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. (We are actively working with US Postal Service to define standards for the use of secure postage [23].)

The US Postal Service, with almost 40,000 autonomous post office facilities, handles an aggregate total of over 165 billion pieces of mail annually [40]. Most mail is metered or printed. (Figure 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 franked, the amount is deducted from the machine's credit register. Postal meters are subject to at least four types of attack: (1) the credit recorded in the postage meter may be tampered with, allowing the user to steal postage; (2) the postage meter indicia 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.[5]

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 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 indicia could protect against forgery or copying. A rough outline of how such a system might work was detailed by Pastor [36].

Unfortunately, a digitally signed indicia 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

[5] 82,000 postage meters in the U. S. are currently reported as lost or stolen [41].

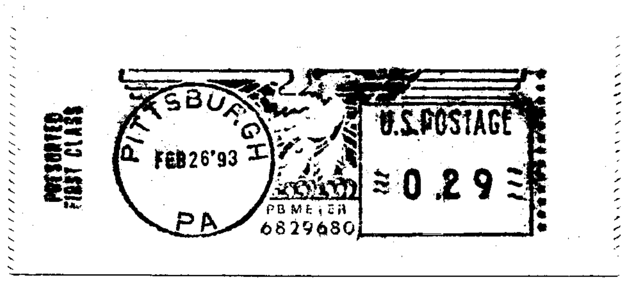

Figure 2: Postage Meter Indicia Can Be Easily Copied or Forged

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 meter indicia appears infeasible.)

We outline a protocol for protecting cryptographic indicia, and demonstrate that the use of a secure coprocessor can address all of the above concerns. With the use of cryptography and secure coprocessors, it is possible to build a PC-based system that can produce fully secure postage indicia.

3.4.1 Cryptographic Indicia

A cryptographic postage indicia 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 indicia depends on the address. A malicious user may copy a cryptographic indicia, 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 indicia 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 software instance producing the indicia, a sequence number for the indicia, 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 indicia on the envelope within a reasonable amount of space. Symbol Technologies' PDF417 [26, 37, 38], for example, can encode 400 bytes per square inch, sufficient for cryptographic indicia. Figure 3 shows an example of PDF417's density.

Six lines of 40 full ASCII characters for each address,[6] four bytes each for hierarchical authorization number, a serial number for the software instance that produced the indicia, the indicia 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 indicia, since the cryptographic signature prevents them from modifying the indicia to change the destination addresses, etc, so the copied indicia 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 indicia also includes source in-

[6] Instead of a 40 character address, a 11-digit extended ZIP code presently in use internally by the U. S. Postal Service may be used instead; an eleven digit address fits in 37 bits.

Figure 3: PDF417 encoding of Abraham Lincoln's Gettysburg Address

formation, a serial number, and the return address, duplicated indicia 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 indicia, they will know that some form of forgery has occurred. We examine the practicality of replay detection next.

3.4.2 Detecting Replays

With a kilobyte of data per indicia, 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 indicia expire six months after printing,[7] an average regional office will see approximately 130,000,000 indicia out of a national total of 80,000,000,000 indicia. If we store one kilobyte of information per indicia (doubling the above estimate) and assume that the entire current mail volume uses cryptographic indicia, this would require only 130 gigabytes of disk storage per facility for logging, well within the capacity of a single disk array system. The indicia database can be viewed as a sparse boolean matrix indexed in one dimension by software instance serial number and in the second dimension by indicia sequence number for that software instance.

To make replay detection even easier, we exploit the physical locality property: pieces of mail franked by a single device are likely to enter the mail processing system at the same primary sorting site. Therefore, cryptographic indicia from the same device are very likely to be canceled at the same regional office, and we can detect replays there. If any cryptographically franked 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 indicia database at each regional office need not be large — each device can simply encrypt a counter value in its indicia. We need only fast access to a bit vector of recently used, unexpired indicia counter values. These bit vectors are indexed by the device's serial

[7] 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.

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.

4 System Architecture

We have implemented Dyad, a prototype secure coprocessor system. The Dyad architecture is based on operational requirements arising from the security applications in section 3. This section discusses Dyad's abstract system architecture based on the operational requirements during system initialization and during normal, steady state operation. A more detailed discussion of the concrete system architecture may be found in [60].

4.1 Operational Requirements

We begin by examining how a secure coprocessor interacts with the host during system boot and then describe 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.

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; and
- encrypted communication channels (key exchange, authentication, private key encryption, etc).[8]

4.2 Secure Coprocessor Architecture

The boot procedure described above made assumptions about secure coprocessor capabilities. We 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 [39], multi-round Snefru [32], or IBM's MDC [25] 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 [27]). 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 to 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.

4.3 Crypto-paging and Sealing

Crypto-paging is another technique for trading off memory for speed. A secure coprocessor encrypts its virtual

[8] Presumably remote hosts will also contain a secure coprocessor, though everything will work fine as long as remote hosts follow the appropriate protocols.

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 virtual memory to disk, except for different cost coefficients. Well-known analysis techniques can be used to tune such a system [29, 61]. The cost variance will likely lead to new tradeoffs: computing cryptographic checksums is faster 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. We observed in [48, 49] 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 [60]). Intermediate-size tables yield intermediate speedups at a slightly higher increase in code size. Similar tradeoffs can be found for software implementations of DES.

4.4 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 its partitioned system structure.

Like normal workstation (host) kernels, the secure coprocessor kernel must provide separate address spaces for vendor and user code in 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 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 [19], the NT executive [14], or QNX [24]. 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 the Citadel secure coprocessor.)

4.5 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[9] to prevent replay attacks where old, buggy, secure coprocessor software is reintroduced into the system. Depending on the algorithm, storage of the fingerprint information can requires only integrity or both integrity and secrecy.

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

[9] One way is to use a cryptographically secure pseudo-random number generator [5, 6] with its internal state entirely in secure non-volatile memory.

upgrade must appear transactional, that is, it must have the properties of *permanence* (results of completed transactions are never lost), *serializability* (there is a sequential, non-overlapping view of the transactions), and *failure atomicity* (transactions either complete or fail such that any partial results are undone [16, 20, 21]). Non-volatile memory gives us permanence; 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.

5 For Further Information

Dyad allows a much broader class of electronic commerce activities than more narrowly defined electronic wallet systems. By using a secure coprocessor model, we are able to add substantial functionality.

Information on secure coprocessors and the Dyad implementation can be found on our WWW page: *http://www.cs.cmu.edu/afs/cs/project/dyad/www/*. Please send any email or inquiries for information to `dyad@cs.cmu.edu`.

References

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

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

[3] Ross Anderson. Making smartcard systems robust. In *IFIPS First Smartcard Research and Advanced Application Conference*, pages 1–14, Lille, France, October 1994.

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

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

[6] 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.

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

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

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

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

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

[12] 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.

[13] Cylink Corp. CY512i press release, February 1995.

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

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

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

[17] Merrick Furst. Personal communications.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] Nevin Heintze, J. D. Tygar, and Bennet S. Yee. Cryptographic postage indicia, 1995. To appear.

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

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

[26] 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.

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

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

[29] 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.

[30] 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.

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

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

[33] Ryoichi Mori and Maraji Kawahara. Superdistribution: An overview and the current status. *Technical Reports of the Institute of Electronics, Information, and Communication Engineers*, 89(44), 89.

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

[35] 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.

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

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

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

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

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

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

[42] Marvin Sirbu and Doug Tygar. Netbill: An internet commerce system optimized for networked delivered services. *IEEE Compcon '95 Conference*, pages 20–25, March 1995.

[43] Sean Smith, David Johnson, and J.D. Tygar. Completely asynchronous optimistic rollback recover with minimal rollbacks. In *IEEE 25th Symposium Fault Tolerant Computing*, Pasadena, CA, June 1995. To appear.

[44] Sean Smith and J. D. Tygar. Security and privacy for partial order time. In *ISCA International Conference on Parallel and Distributed Computing Systems*, pages 70–79, Las Vegas, NV, October 1994.

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

[46] 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.

[47] Telequip, Inc. Crypta Plus press release, January 1995.

[48] 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.

[49] 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.

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

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

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

[53] U. S. National Institute of Standards and Technology. Federal information processing standards publication 140-1: Security requirements for cryptographic modules, January 1994.

[54] U. S. National Institute of Standards and Technology. Csl newsletter, February 1995.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] Bennet S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[61] 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.