In Carnegie Mellon Computer Science: A 25-Year Commemorative, ed. R. Rashid.
ACM Press and Addison-Wesley, 1991, pp. 163-197.
(Note: The first printing of this volume had incorrect text due to a production error.)

7

Strongbox: A System for Self-Securing Programs

J. D. Tygar and Bennet S. Yee

7.1 Introduction

Security is a pressing problem for distributed systems. Distributed systems exchange data among a variety of users over a variety of sites, which may be geographically separated. A user who stores important data on processor A must trust not just processor A but also the processors $B, C, D, \ldots$ with which A communicates. The distributed security problem is difficult, and few major distributed systems attempt to address it. In fact, conventional approaches to computer security are so complex that they actually discourage designers from trying to build a secure distributed system: A software engineer who wishes to build a secure distributed data application finds that he or she must depend on the security of a distributed database which depends on the security of a distributed file system which depends on the security of a distributed operating system kernel, etc. Under this traditional design approach, security necessarily becomes a secondary concern since just making an (unsecure) distributed system work efficiently is a daunting task. And adding security raises the difficulty by an order of magnitude.

We propose a new model of security: *self-securing programs*. Self-securing programs are designed to run in environments where only a minimal number of assumptions are made about the security of the operating system kernel. Self-securing program run in a client-server model, use advanced authentication and fingerprinting techniques, and can guarrantee extremely high levels of security without requiring a secure kernel. We have built a system for constructing self-securing programs called Strongbox. We have made the use of Strongbox relatively transparent to programmers who write multithreaded servers. This allows existing servers to be retrofitted with security and allows programmers to separate security from other concerns. Strongbox provides facilities to protect the privacy of data and the integrity of data from alteration, and to implement quickly a variety of policy decisions about data protection.

Strongbox depends on two types of assumptions: one concerning the privacy of a process's memory and one concerning cryptographic security. Clearly protected memory is requisite for security --- otherwise an adversary could ''spy'' on a computation and break security. The cryptographic assumption is used to implement a new *zero-knowledge* authentication protocol. This protocol performs substantially better than previous authentication protocols, and includes facilities for key exchange. (Our method is faster than previously proposed zero-knowledge authentication protocols such as [11]. Moreover, our method can be *proven* not to leak any information about the authentication keys we use --- this stands in contrast to authentication protocols such as Needham and Schroeder's method [23, 24].) Finally, our method of demonstrating that the security of our protocol differs from those suggested by Burrows, Abadi, and Needham [6] in that our proof technique does not suffer from the type of drawbacks noted in [15].

The current version of Strongbox does not yet address several secondary concerns including *traffic analysis* of data message exchange, communication by adjusting the use of system/network resources (the *covert channel* problem), or the availability of system components (the *denial of service* problem). Our future work will focus on these concerns, and some preliminary thoughts toward these problems are discussed in Section 8.8. However, Strongbox can be used in conjunction with any solution to these secondary concerns.

This chapter begins by discussing our goals for Strongbox --- both functional and performance goals. In Section 8.2 we discuss the basic computational model that we assume for our system. In Section 8.3 we give a high level description of the methods used by Strongbox. Sections 8.4 and 8.5 present an overview of the architecture used in implementations that are built on top of Camelot, a distributed transaction system, and Mach, a UNIX-compatible distributed operating system, respectively. (More information about the Camelot implementation is given in [41].) In Section 8.6, we give performance figures and code size for these algorithms, and discuss issues of bootstrapping Strongbox. In Section 8.7 we give full descriptions of our new algorithms, and efficient implementation techniques for those algorithms.

7.1.1 Our Goals

The primary functional goals of Strongbox are to guarantee the integrity and privacy of data handled by it. Section 8.3 shows that the architecture of Strongbox protects data from modification and guarantees that data messages are protected by end-to-end encryption. In Section 8.7 we show that Strongbox's fingerprinting and authentication algorithms do not leak information. An additional functional goal of Strongbox is to provide programmers with a security library that can be easily used in a server or client. We do not expect programmers to master the subtleties of a delicate protection mechanism. We have structured our interface so that converting an existing client/server to be secure requires only a few simple modifications to the program text.

Security is typically expensive. It is not uncommon for secure versions of operating systems to run an order of magnitude slower than their insecure counterparts. We view this as completely unacceptable for real applications; we demand that the overhead for security, amortized over all computations, should use no more than 5% of the processor cycles, excluding encryption. We have worked hard to make our security routines extremely fast, and our performance figures are in Section 8.6.

Another measure of the effectiveness of security code is the size of the code. The smaller the code is, the less likely it is to contain errors and the easier it is to verify, whether by formal or other methods. Since our library isolates simple points of communication, we believe that we have met those goals.

7.2 Statement of Model

When designing a security system, it is important to keep the system model in mind. Strongbox is intended to be used within the client-server model where client programs running on the behalf of users invoke operations within servers using interprocess communication (IPC). In Strongbox, we restrict client/server interactions to remote procedure calls (RPCs) --- access is controlled by servers at this level. It is necessary to make some assumptions about the rest of the system when building secure facilities. For example, if the system design assumes an insecure communication mechanism, then communication security must be provided by other means, such as cryptography, and one must make the assumption that the cryptosystem used can not be compromised by attackers. Since cryptosystems can always be broken by nondeterministic agents (who can simply guess the cleartext and the key and verify that the encryption function holds), if we adopt the practice of considering operations in P as tractable and operations in NP - P as intractable, then showing a secure cryptosystem exists is equivalent to showing P is different from NP, a well known open (and difficult) problem.

In building a security system, it is necessary to make some assumptions. We believe that our assumptions are the minimal ones needed to provide security. We need to make a complexity assumption that some problem, such as factoring large integers or inverting the data encryption standard (DES), is intractable. We use this assumption in our authentication, key exchange, and encryption algorithms. (It is important to note that for our authentication algorithm, this weak assumption will allow us to authenticate processes while guaranteeing that no bits of information are leaked to either party or to an eavesdropper.) We also need to assume that our base operating system supports protected memory, including contents of virtual memory stored on a disk,[1] since without this assumption no privacy is possible between processes on a single host because no process can have secrets. In addition, we need assumptions about physical security; for example, we assume that the local host is physically secure, that it is configured properly so that there are no security holes outside of Strongbox's domain, i.e., the terminal lines from the user are not tapped, the central processing unit (CPU) and display are not bugged to leak information, etc. We do *not*, however, make any assumptions about the security of the network --- we assume that an adversary can eavesdrop on messages, replay messages, inject his or her own messages, and prevent messages from being delivered. Since servers that do not use Strongbox are not offered any protection, we assume that application programs use our protection scheme uniformly. We assume that our algorithms were implemented without error, and that the compiler produced correct object code for them.

The current version of Strongbox does not address issues of denial of service, covert channel analysis, or traffic analysis of messages (information revealed in the pattern of message transfers). Although we have not explicitly addressed these problems, we conjecture that they may be solved by extensions to the self-securing paradigm. For example, the Camelot transaction system [10] supports fault tolerance, and the Camelot version of Strongbox makes that fault tolerance secure. We believe that these fault-tolerant facilities might be extended to the security case to support protection against denial of service attacks. (For some theoretical contributions to these issues, see [17, 28].)

A key scenario for Strongbox is the loosely coupled distributed systems case. In these systems, covert channel analysis may be considerably simplified by storing files and running processes of a single security level on each host. Interactions between security levels will take place over the communication network, which is a simpler object to examine for covert channels than an operating system on a single host. In addition, secure memory for processes will be easier to satisfy if all user-applications on the node are at the same level of security. This is a natural mode of operation for a loosely coupled network of

[1] If we are implementing Strongbox on a paging operating system, we must depend on the security of the paging system. In some cases we can assist the security of the data stored by the pager by using a pager that encrypts as it pages.

workstations.

We are continuing to explore approaches such as these in ongoing research. The current security code is publicly available from the authors. We will continue to examine its performance in large applications.

7.3 Conceptual Solution

At the core of Strongbox are new routines for key exchange, authentication, and fingerprinting. End-to-end private key encryption protects the privacy and integrity of messages passed among clients, servers, and other system components. Because we do not make assumptions about the security of communications, it is necessary to encrypt our RPC messages; the encryption mechanism, however, is modular and can be easily omitted when appropriate. In particular, in situations where communication is secure or involves no sensitive data --- but where operations on sensitive data may be requested --- only authentication is required. Key-exchange is performed using a public key system equivalent to deciding quadratic residuosity. In addition, Strongbox provides an authentication system that provides us with support for any user-supplied access control/authorization system. This authentication system differs from previous authentication and key exchange protocols such as Needham-Shroeder [24] in that it can be proved to not leak any information that would allow eavesdroppers to masquerade as either party. The authentication algorithm is based on the idea of proving identity by having the authenticator prove that he or she has the solution to an *authentication puzzle* without revealing the solution itself. It is superior to the algorithm described in [11] because it provides a level of security superexponential on the size of the puzzle.

Integrity of data or program text files is checked in Strongbox by using provably secure cryptographic checksums. These checksums, called *fingerprints*, are computed prior to storing data in the file system and are checked when the system retrieves data.

Below, we will first talk about Strongbox's operation by describing its system components and their interactions with a client when that client is started. Next, we will describe what happens when Strongbox is booted.

7.3.1 Strongbox System Components

On each host, two servers are essential to Strongbox's operation. The first is the *White Pages* server which maintains the database containing Strongbox key exchange information and authentication puzzles. The second server is the *secure loader*. The secure loader is a user-level program that uses operating system primitives to load the run-time image of Strongbox servers and clients after verifying their fingerprints. Instead of using a loader within the kernel after verifying the fingerprint, Strongbox must first read the program text into memory, verify the fingerprint, and load the new task with the text from memory. This procedure avoids the possibility of an attacker changing the executable file just after the fingerprint check but before the kernel can load the task. Exhibit 8.1 shows the relationships among the various system components in the Camelot implementation of Strongbox, which is discussed in Section 8.4. The same relationships hold for these servers in the vanilla implementation of Strongbox, modulo the absence of Camelot. (This scenario is discussed in Section 8.5).

We now consider the typical interactions of a secure client with secure Strongbox servers that it uses, the White Pages server, and the secure loader. To start the first secure client running, a user requests the secure loader to start a new client. The user authenticates his or her identity to the secure loader by a standard password mechanism. After the authentication completes, the secure loader will create a new task, load the task's memory with the fingerprint-checked program text, initialize the task's registers, and start the task running. The secure loader gives the task a seed for its psuedo-random number generator and the puzzle solution for a new, randomly created authentication puzzle.

After the secure client is bootstrapped, it runs on the behalf of the user and interacts with the user using its standard input and standard output streams. The White Pages server *WP* maintains a database of servers and client names along with their key exchange information and authentication puzzles. When the client *C* needs to make an RPC with a secure server *S*, it must first contact a *WP* to obtain *S*'s published key exchange information and authentication puzzles. If *S* is on the local host, the client will ask the local WP_{local}, and it will not need to authenticate the identity of WP_{local} (we assume that the local host was booted securely; see Section 8.3.2). The more interesting case occurs when *S* is on a remote host. In that case, we may either run an authentication with the remote White Pages server WP_{remote} directly and obtain *S*'s puzzle, or we may ask our WP_{local}; WP_{local} will forward our request to WP_{remote} after performing the appropriate key exchange and authentication steps to establish a secure channel and verify WP_{remote}'s identity. The standard routine supplied by Strongbox uses the latter method. After *C* obtains *S*'s key exchange data and authentication puzzle, *C* invokes the key exchange routine to establish a secure channel, perform an authentication, and obtain an authentication token from *S*. The authentication is symmetrical:[2] the client is assured that it is communicating with the right server, and the server is assured of the identity of the client (and that of the user) and may control access based on that identity. It is only after *C* has obtained an authentication token that it can call the remote procedures at *S*; the authentication

[2] The server, symmetrically, asks a White Pages server for *C*'s puzzle, and two rounds of authentication, one proving the identity of *C* to *S* and the other proving the identity of *S* to *C*, are run in a single, integrated protocol.

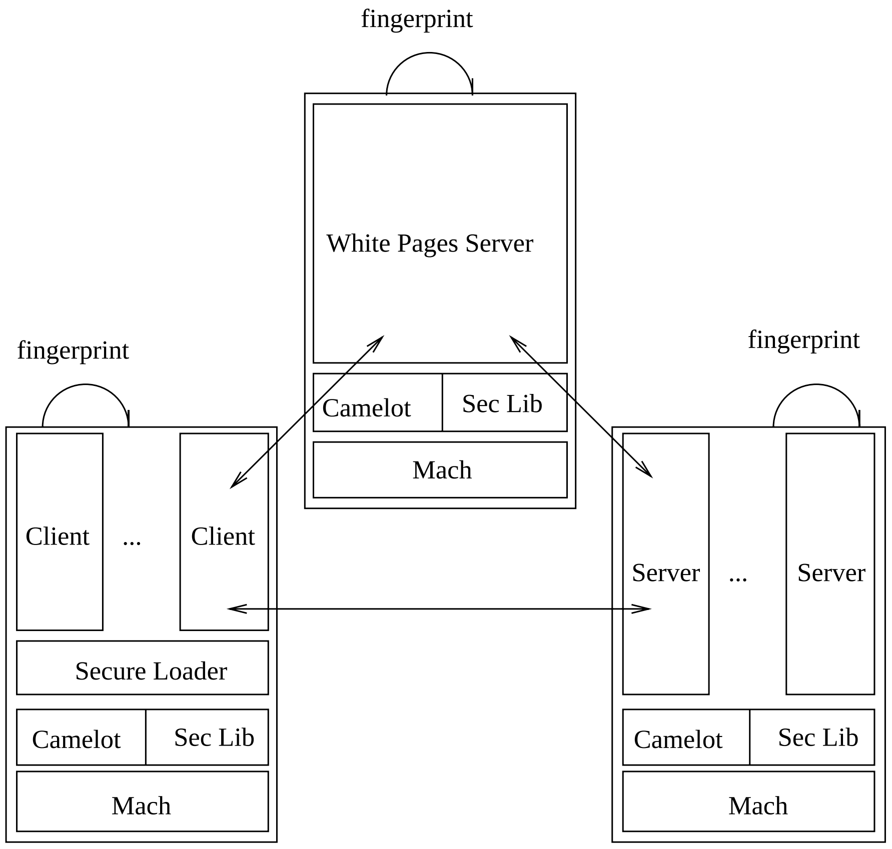

EXHIBIT 7.1
Strongbox architecture --- Camelot implementation. In this figure, we show how Strongbox interacts with other system components. Each of the large boxes denotes a computer. The client, server, and White Pages server are shown as running on different machines; this is not a requirement of Strongbox, however, so they may all reside on the same computer. The lines among the client, server, and White Pages server boxes denote communication that may be visible on the communication network. Within each computer, the smaller boxes denote the major software components: the Mach operating system kernel, the Security Library which is used by every secure server or client, the White Pages server, the secure loader, and the secure clients and servers. The curved arrows denote the fingerprint operations which verify that none of the files have been corrupted.

token is used in all subsequent requests to this server, therefore depending on the life time of the client, the cost of key exchange, and the cost of authentication, already low, may be amortized over many RPC operations.

7.3.2 Booting Strongbox

To run securely, Strongbox must be booted in a secure fashion. We assume that the Strongbox host computer is physically secure and that a trusted operator boots the machine. How do we validate the integrity of the host? The same solution that we use to verify the integrity of data files and program text when Strongbox is running is applied to the operating system kernel and system utilities: simply maintain a list of fingerprints for the system components. We verify their correctness at boot time, and a copy of the fingerprint code is placed in the boot read-only memory (ROM) to deter tampering. The fingerprint list is kept secret, and the operator must enter a decryption key to initiate the boot sequence. Along with using encryption, the list of fingerprints of the system modules may itself be fingerprinted; if desired, each host may use a distinct encryption key and a different system fingerprint key. Since the puzzle for the White Pages server is common to *all* White Pages servers, the solution, which is compiled into the binary of the server, must also be kept secret via encryption. Note that the plaintext version of the binary need not ever reside in the file system: we can decrypt entirely in main memory and then run the server directly.

Currently, a single puzzle/solution pair is used for all White Pages servers. To avoid the problem of a single untrustworthy site causing problems, multiple puzzle/solution pairs may be employed. One approach to doing this is outlined in [17].

7.4 Camelot Implementation

We initially implemented a Strongbox interface to work with the Camelot distributed transaction system. Camelot extends the usual programming model [4] to include the *transaction* abstraction for persistent memory objects. Before we describe our implementation for Camelot, we first give an overview of Mach and Camelot.

7.4.1 Mach/Camelot Overview

The Mach operating system [1, 35] provides the basic services needed to support the client-server model for which Strongbox was designed. Mach is upward compatible with 4.3 BSD UNIX, but provides additional primitives for supporting multithreaded processes (called tasks), location transparent intertask communication, and efficient virtual memory. Mach is also relatively machine-independent, running on a variety of uniprocessors and multiprocessors including the IBM RT, Sun 3, Sun 4, DEC VAX and Pmax, Encore Multimax, and the NeXT workstation. The work reported in this chapter was done primarily on Mach running on RT/APCs and MicroVAXs.

The Mach Interface Generator (MIG) is Mach's RPC stub generator [19]. MIG accepts a syntactic specification for procedure headers (written in a Pascal-like syntax), and generates libraries for calling and dispatching RPCs. These libraries link into the executable image of both clients and servers. The client library contain RPC procedure stubs that copy input arguments into a Mach message, send the message to the server, and unpack the reply message contents into output arguments. The server library contains a demultiplexing routine that, given a message sent by some client, figures out from the message ID in the header the RPC to which the message corresponds, unpacks the arguments appropriately, and calls the service routine with the unpacked arguments. When the service routine completes, the demux routine packs the output arguments into a reply message and sends that message back to the client.

Another detail of Mach IPC that is important to Strongbox is the message format. Each IPC message has a fixed header followed by a variable data part. The variable data part of a message is an array of data descriptors and data. The data descriptor is a structure containing the size and type of the next datum. MIG places arguments in the message in the order of declaration, so by constraining the RPC declaration we can be sure of the argument's location in a message. In particular, this is how Strongbox extracts the authentication token from generic RPC requests.

The Camelot distributed transaction facility is layered on Mach and MIG. It provides mechanisms for constructing reliable distributed programs that access shared data [9, 38]. Camelot simplifies the handling of network, processor, and software failures; performs synchronization of concurrent programs; manages storage resources; and reduces the complexity of invoking and building shared databases.

The most important abstraction provided by Camelot is the transaction, a collection of operations bracketed by two markers: `BEGIN_TRANSACTION` and `END_TRANSACTION`. Transactions provide three properties that reduce the amount of attention that a programmer must pay to concurrency issues and failures [14, 39]:

- *Failure atomicity*. Failure atomicity ensures that if a transaction's work is interrupted by a failure, any partially completed results will be undone. A programmer or user can then attempt the work again by reissuing the same or a similar transaction.

- *Permanence*. If a transaction completes successfully, the results of its operations will never be lost, except in the event of catastrophes that damage stable storage. Systems can be designed to reduce the risk of catastrophes to any desired probability.

- *Serializability*. Transactions are allowed to execute concurrently, but the results will be the same as if the transactions executed serially. Serializability ensures that concurrently executing transactions cannot observe inconsistencies. Programmers are therefore free to cause temporary inconsistencies during the execution of a transaction knowing that their partial modifications will never be visible.

The properties of transactional memory is exploited in the Camelot version of Strongbox to simplify the structure of the servers.

7.4.2 Camelot Strongbox

Camelot entities are divided into two classes, clients and servers; typically, clients initiate operations at servers via RPCs, though a server may act as a client of a second server as well. Instead of using the MIG-generated RPC stub routines directly, the RPCs were performed by wrapping the RPC request in a C preprocessor macro --- Camelot does not allow ‘‘raw’’ RPCs because it must keep track of transaction IDs, etc. Since the IPC among Camelot entities are so highly constrained, we were able to engineer the integration of Strongbox so that modifying an insecure Camelot server or client is as painless as possible. Using Strongbox usually involves changing the macro invocations to use special Strongbox-style macros that manage both Strongbox bookkeeping and Camelot bookkeeping.

In nonsecure Camelot, the `SERVER_CALL` macros handle the transactional bookkeeping for invoking a RPC for both servers and clients. Servers never receive RPC requests directly; rather, a Camelot supplied routine receives the incoming request and perform some preprocessing prior to invoking the MIG generated demultiplexing routine. In secure Camelot, instead of using the `SERVER_CALL` macro for performing RPCs, we require that the `SEC_SERVER_CALL` macro be used instead.

The secure RPC macro hides a bit more detail than the nonsecure version --- in addition to the extra input parameters already hidden by `SERVER_CALL` (e.g., the current transaction ID), it hides another argument, an authentication token,[3] which will also be sent along with the ‘‘normal’’ RPC parameters. Prior to performing the actual RPC request, the secure macro performs Strongbox

[3] These tokens are *capabilities* that could be transferred among programs. We do not make any distinction between authentication tokens and capabilities: in particular, a successful authentication for group membership just results in a new capability in addition to the old one rather than a modification to the data entry associated with the old token. See [18].

bookkeeping such as looking up the token associated with the server. If there are no valid tokens or if the RPC request is denied for some other reason (e.g., the token expired or the server restarted), the macro automatically runs the key exchange and authentication protocols to obtain a valid token. By adding cached tokens to the RPC parameters and obtaining new tokens when needed automatically, the token management is completely transparent to the application. Hiding all this activity from the interface is desirable for simplifying the task of converting existing non-secure Camelot applications to use Strongbox. Another Strongbox macro, `SEC_CAP_SERVER_CALL`, allows the programmer to supply a capability token as a parameter of the RPC, thus enabling the program to explicitly manage its tokens.

On the server side, we hide Strongbox-related activities by providing a cover routine for demultiplexing incoming RPC requests. To handle the RPC demultiplexing, Camelot servers normally gives the name of the demux procedure generated by MIG to the Camelot library via the `START_SERVER` macro. Strongbox provides the `SEC_START_SERVER` macro to be used in lieu of `START_SERVER` macro, which substitutes in a special predemux procedure that implements access control. Since all initial segments of Strongbox RPC message bodies contain a capability token and all RPC headers contain the RPC request ID, we easily provide coarse-grained access control at the RPC entry-point level. The access control routine consults a simple, per-server authorization database (implemented using Camelot's recoverable memory objects) implemented with a library provided with the Camelot version of Strongbox to decide whether to grant access to a particular RPC routine. If access is denied, the access controller would just abort the transaction; if access is permitted, the access controller invokes the normal demux procedure. By performing access control before the service routines are invoked, we eliminate the need to change the sources for the service routines. This method provides only coarse-grained access control; a simpler mechanism is used in the vanilla version of Strongbox described in the next section.

Why are the authentication tokens unforgeable? When the network is insecure, the Strongbox RPC library encrypts all traffic between clients and servers. The encryption used is based on using a cryptographically secure pseudo-random number generator (see [5]) as a source of random bits for a one-time pad,[4] so multiple encryptions of the same data results in different ciphertext: replaying an encrypted token from a message will not aid in spoofing subsequent messages. There is a caveat: the message header and the type information for the data fields cannot be encrypted because they are interpreted by the kernel, and a small integer (part of the token) must remain visible within the message. The integer identifies

[4] We designed a pseudo-random number generator that is based on the assumption that inverting DES is intractable. Our generator has considerable performance advantages over the generator described in [5].

the secure channel to which the message belongs so Strongbox will decrypt the message with the appropriate key. This means that, in its current form, RPC communication is not protected against simple traffic analysis.

The authorization database that the access control routine consults is maintained within the server. In particular, Camelot's recoverable memory is used, simplifying the issues of saving and restoring this database. Because the memory is transactional, once we commit modifications to the authorization database, the change is permanent --- the server may be arbitrarily crashed and restarted without fear of damage to the database. We assume that the underlying Camelot core system servers are secure, and that the transactional logging of the values of the recoverable memory is performed in a secure fashion.[5] When a secure Camelot server initially starts up, it performs a once-only transaction that initializes its recoverable memory. At this time, the Strongbox server initializes its authorization database to allow an administrative user to access Strongbox-provided authorization handling RPCs. All other access permissions are derived from the administrative user.

7.5 Vanilla Implementation

The ''vanilla'' implementation of Strongbox runs directly on Mach. There are a few differences between the Camelot implementation of Strongbox and the vanilla implementation. This section will describe these differences.

The overall architectures of the two implementations are the same. The main observation is that, without the transactional abstraction provided by Camelot, writing large servers become more complex. Depending on system reliability, a server has several strategies to prevent corrupting its database if the node crashes: it may simply checkpoint its database periodically; it may maintain its database entirely on disk, using main memory as a write-through cache; or it may log changes prior to modifying values so it can reconstruct its database from the log. Furthermore, Strongbox places no trust in the filesystem; if a server is to share that assumption, it must encrypt its database when writing it out to disk. For this purpose, Strongbox provides a routine for using DES in block chaining mode to encrypt contiguous blocks of memory.

In the vanilla version of Strongbox, the access control is no longer at the RPC level. We simply provide a standardized interface between the authentication routines and any authorization mechanism (perhaps provided as library) that the programmer wishes to use. All Strongbox servers' service routines take an authentication token as an argument --- as before, its placement in the argument list is constrained so that encryption can be used to establish a secure channel.

[5] The Camelot log must be encrypted so no data can be leaked and fingerprinted so no data can be modified, or the logging must be to a secure device.

Unlike the Camelot implementation, however, access is not controlled at the RPC entry point, and the programmer must explicitly control access.

Typically the programmer implements access control by placing queries to authorization routines at appropriate places within the service routine with the authentication token and name of the operation/suboperation as parameters. The authentication routines provide the ability to map from the authentication tokens to client/user names as strings or internal ID numbers, so the authorization routines can use these identifiers as indices into its access control database. Finer grained protection such as that needed by a file server may need authorization routines that allow object identifiers as well. Where the access control matrix for the previous case was indexed by the tuple *user* $\times$ *operation*, here it is indexed by the triplet *user* $\times$ *operation* $\times$ *object*.

7.6 Performance and Implementation Issues

This section gives timing figures for our implementations of the authentication algorithm and fingerprinting algorithm described in Section 8.3. Our timing figures are for an IBM RT/APC, which is a reduced instruction set computer (RISC) running at 4 MIPS.

An IBM RT/APC requires 105 milliseconds (mS) to perform (one-way) authentication in addition to the RPC overhead. To perform the authentication, the client invokes two RPCs. The overhead for performing an RPC is approximately 35 mS [40]. We have a software implementation of DES that works at a rate of 220 encryptions per second.

An IBM RT/APC achieves a fingerprinting rate of over 880 KBytes/sec. The fingerprinting routine uses a 65536 (2^{16}) entry table of precomputed partial residues to achieve this speed. Another implementation which uses a 256-entry table achieves a fingerprinting rate of 710 Kbytes/sec. The residue table initialization algorithm is described in Section 8.7.2. For the large table, the time required is approximately 1 sec; the time for the small table is negligible.

The trade-off in data size and speed between the two versions of fingerprint implementation indicates using the smaller version for most cases. The large table version is useful where the same irreducible polynomial is used for a large amount of data; the small version wins out when the irreducible is changed often, or where there are tight memory requirements.

While not a requirement when implementing security code, smaller code size is desirable. When the code is smaller, the system is easier to verify and is less likely to contain bugs. The key exchange routines consists of 80 lines of C code, not including comments. The authentication routines consists of 75 lines of C code, not including comments. 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, not including comments. Our total core routines are relatively small: 366 lines of C code.

7.7 Algorithms and Analysis

This section discusses and analyzes the key algorithms in Strongbox. **Warning**: The material in this section is substantially more difficult than the rest of the chapter. A casual reader may wish to skip it. The notation used is standard from number theory and algebra (groups, rings, and fields). 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 the one given by Rabin in [32].

7.7.1 Description of Algorithms

Before we launch into the description of our algorithms, let us 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. As we will see later, Blum moduli have special number theoretic properties that we will make use of in our 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.[6]

Key Exchange. End-to-end encryption of communication channels is mandatory when the security of the channels is suspect. To do this efficiently, we use private-key encryption coupled with a public-key encryption algorithm used for key exchange. We will 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 difficult [27]. Unfortunately, Rabin's system is brittle, i.e., if the agents can be made to decrypt ciphertext chosen by an attacker, it is easy to subvert the system, divulging the secret keys. The RSA encryption algorithm [36], while

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

believed to be strong, has not been proven secure. Chor [7] 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 Strongbox is based on the problem of deciding quadratic residuosity, another well-known problem in number theory that is believed to be difficult.

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

Strongbox's public key system works as follows: All entities in the system publish via the 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.

Now, observe that Blum moduli have the property that the multiplicative group $\mathbb{Z}^*_{M_i}$ has -1 as a quadratic nonresidue. 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: [7]

$$L(-1, p) = -1^{(p-1)/2} \tag{7.1}$$

$$L(m \cdot n, p) = L(m, p) \cdot L(n, p) \tag{7.2}$$

When $p = 4k + 3$, from (8.1) we have $L(-1, p) = -1^{2k+1} = -1$, so -1 is a quadratic nonresidue. Further, it is easy to randomly generate random quadratic residues and nonresidues: simply chose a $r \in \mathbb{Z}^*_{M_i}$ randomly [8] 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 nonresidue, use $-r^2 \bmod M_i$.

We have established that, given $n = p \cdot q$ where both p and q are of the form $4k+3$, it is easy to generate random quadratic residues and quadratic nonresidues. Next, we need to 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

[7]See [25] for a list of identities involving the Legendre symbol.

[8] 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 difficult, this is an highly improbable event.

efficiently computed using a simple algorithm similar to the Euclidean algorithm for computing the *gcd*. 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 [13]. 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, we compute $x_j \equiv -1^{b_j} r_j^2 \pmod{M_i}$ 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 or she 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 nonresidues formed as residues modulo M_i of $-r^2$ will also have 1 as the value of the Jacobi symbol.[9]

When 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 Strongbox managed RPC traffic between the client and the server.

Authentication. Whether or not our communication channels are secure against eavesdropping or tampering, some form of authentication is needed to verify the identity of the party with whom we are establishing communication. Even if our physical network links are secure, we still need to use authentication: to look up the communication ports of remote servers, we must ask a nameserver on a remote, untrusted machine. Since we make no assumptions about the network name servers, even the identity of a remote host is suspect. Thus on top of the existing Mach nameserver, Strongbox provides a White Pages server that maintains authentication information (in addition to key exchange moduli when

[9] 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. If only a limited number of messages or message formats are used, attackers can easily gather statistical information on the distribution of messages.

applicable) and is itself an authenticated agent. For the purposes of this discussion, the role of the White Pages server is to serve as a repository of *authentication puzzles*. Authentication is based on having the authenticator prove that he or she can solve the published puzzle without revealing the solution.

The best available protocols for authentication all rely on a crucial observation made by Michael Rabin in [27]: 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 Michael Rabin [31], and a protocol developed by Feige, Fiat, and Shamir (the FFS protocol) [11]. Between the FFS and Rabin's protocols, Rabin's method is much stronger because it provides a superexponential security factor. In contrast to Needham and Schroeder's authentication protocol[24], 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. Strongbox uses an authentication protocol which is a modified version of Rabin's method. Like Rabin's protocol, the Strongbox protocol has the important properties of being decentralized and having a superexponential 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 --- eavesdroppers may listen to the entire authentication exchange, and nobody will gain any information at all that would enable them to later masquerade as the authenticator; furthermore, both ends of the protocol may be simulated by any entity even though they have no knowledge of the secrets known only to the authenticator. We will see how this is possible in Section 8.7.2.

Let's see how Strongbox authentication works. After establishing a secure communication channel with the remote entity, we query the White Pages server for the corresponding authentication puzzle. These authentication puzzles are randomly generated and can be solved only by their owners who knows their secret solutions. In the protocol, however, the remote entity is not asked to exhibit a solution to their puzzles, but rather is asked to show a solution to a randomized version of their puzzle. Our 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 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 our 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 $\vec{E} \circ \vec{E} = \lfloor \frac{n}{2} \rfloor$, and $\phi \in S_n$ a permutation.[10] We can represent ϕ as a number φ from 0 to $n! - 1$ which represents elements of S_n under a canonical numbering.[11]

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

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

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 $P_{pub} \equiv Q_{pub} \equiv 3 \pmod{4}$. 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 8.7.2, $\vec{X}$ is used to randomize the puzzle in order to keep the solution from being

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

[11] 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 [21, 8].

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 $\forall j : x_{\phi(j)} \equiv y_j^2 \cdot v_{i,j}^{e_j} \pmod{M_i}$ holds.

Authentication and Secret Agreement. Instead of running key exchange and authentication as separate steps, we 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 an RPC, 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 Strongbox since we only 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 calculate a Blum modulus $M_i = P_i Q_i$, with P_i and Q_i primes of the form $4k + 3$. i keeps the values of P_i and Q_i secret and publishes M_i. 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 $z_{i,j}$ 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 $\mathcal{A}$, first $\mathcal{C}$ chooses a challenge $(\vec{E}, \phi)$ as before, with $\vec{E} \in \{0, 1\}^n$ such that $\vec{E} \circ \vec{E} = \lfloor \frac{n}{2} \rfloor$, 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 canoninical 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 $\vec{R}$

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

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 $\mathbb{Z}^*_{M_aM_c}$, 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 $r_j = w_j^2 \bmod \mathbb{Z}_{M_aM_c}$. Next, $\mathcal{A}$ computes the puzzle randomizer $\vec{X}$ from $\vec{R}$ as before, setting $x_j = r_j^2 \bmod \mathbb{Z}_{M_aM_c}$, 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

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

where b_j is a random bit.

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

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

holds. There are $\lceil \frac{n}{2} \rceil$ 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, $b_j = 1$.

Practical Authentication and Secret Agreement. In this section, we present another protocol for simultaneous authentication and secret agreement that also requires two rounds of interaction but requires many 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 is highly appropriate for use in security systems such as Strongbox.

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 $v_{a,i} = 1/s_{a,i}^4$. Published for all to use is a modulus M_{pub}; the two prime factors of M_{pub}, P_{pub} and Q_{pub}, are forgotten.

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 $\vec{E} \circ \vec{E} = \lfloor \frac{n}{2} \rfloor$, 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 as two numbers using mappings $f\colon \{\vec{E}\} \leftrightarrow \mathbb{Z}_\zeta$ and $g\colon S_n \leftrightarrow \mathbb{Z}_{n!}$. Let $E = g(\phi) \cdot \zeta + f(\vec{E})$, the combined encoding for the two parts of the challenge,[12] and let $C = E^2 \bmod M_{pub}$. The value C is used to commit $\mathcal{C}$'s challenge to $\mathcal{A}$. $\mathcal{C}$ 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 $x_i = r_i^4$. $\mathcal{A}$ 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 $\lceil \frac{n}{2} \rceil$ such values. Let those i such that $e_i = 0$ be the set I.

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, $\mathcal{C}$ extract 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

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

where $i \in I$, $u_i = r_\phi(i)$, w_i is a nonce value, and E_k is an element of a family F of 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 $e_i = 0$:

[12] If $|E| \not\approx |M_{pub}|$, extra random pad bits may be necessary.

since

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

and $\mathcal{C}$ knows the factorization of M_c, $\mathcal{C}$ can extract the four square roots of $y_i \pmod{M_c}$, one of which was the original $r_{\phi(i)}$ chosen by $\mathcal{C}$.[13] To choose the proper root of y_i, $\mathcal{C}$ uses the i-th element of $\vec{W}$. $\mathcal{C}$ can try all four square roots of $y_i \pmod{M_c}$ 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.)

Fingerprints. Next, we describe the Karp-Rabin fingerprinting algorithm, which is crucial to Strongbox'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[14] and the likelihood of an attacker forging a fingerprint without knowing the irreducible polynomial is exponentially small on the degree of the polynomial.

In the current Strongbox implementation, we choose random irreducible polynomials p from $\mathbb{Z}_2[x]$ of degree 31 by the algorithm due to Rabin [29, 20, 33].

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 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 $f(x) \bmod p(x) = \varphi(f(x))$.

7.7.2 Analysis of Algorithms

Key Exchange. The correspondence between the problem of deciding quadratic residuosity and the protocol is direct. For a detailed analysis, see [13].

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

[13] Standard algorithms for performing this are [3, 2].

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

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 $v = 1/s^2$; the puzzle randomizer be $x = r^2$ (r known only to the authenticator); the challenge be $e \in \{0, 1\}$; and 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 $\frac{1}{2}$ probability, 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, he must be able to handle the case where e is either 1 or 0, and thus he 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 [27] 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

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

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

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

(using the Stirling's approximation of $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$) which shows that P is clearly superexponentially small. By using longer vectors or multiple vectors (iterating) the security factor can be made arbitrarily high. Note that since the security factor is superexponential on 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

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

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 $\vec{E} \circ \vec{E} = \lfloor \frac{n}{2} \rfloor$ (i.e., the number of 1 bits in $\vec{E}$ is $\lfloor \frac{n}{2} \rfloor$). 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 1 values in $\vec{E}$, of the permutation ϕ gives us.

Why do we restrict $\vec{E}$ to have $\lfloor \frac{n}{2} \rfloor$ 1 bits? If $e = \vec{E} \circ \vec{E}$ 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 $e = 0$ is most probable. This weakens the security factor of our algorithm. In the limit case where $\vec{E}$ are all zeros, then our algorithm no longer provides superexponential security.

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

The authentication protocol not only provides superexponential security when the moduli cannot be factored, but is also zero knowledge. The encoded challenge vector, $\vec{C}$, performs *bit commitment*, 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 our model differs slightly from the usual model for zero knowledge interactive proofs in that here 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 is assumed to be infinitely powerful as in the usual model, the prover can 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 [37] requiring more multiplications but use fewer random bits.

Merged Authentication and Secret Agreement. Like the first authentication algorithm, the merged authentication and key exchange algorithm reveals no information assuming that factoring and deciding quadratic residuosity are intractible.

How does the merged algorithm differ from the original algorithm. The difference is that we use $M_a M_c$ as the modulus for the nonce vectors, and we 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

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

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

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

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 B compromise the authentication? To do so, B must guess the values of $\vec{E}$ and ϕ just as in the vanilla authentication protocol. The probability of somebody breaking the authentication is superexponentially small as before. (See Section 8.7.2)

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 intractible 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 $L(y_j, P_c)$.

Practical Authentication and Secret Agreement. Assuming that factoring is intractable, the third ''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 we assume that 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 we omit that here. (See Section 8.7.2.) 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

$$y_j = r_{\phi(j)}^2 \bmod M_c$$

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.

Fingerprints. Before we analyze the peformance 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 $\varphi: \mathbb{Z}_2[x] \twoheadrightarrow GF(2^m)$, where $\ker\varphi$ is the ring generated by the irreducible polynomial p. To be precise, φ identifies the indeterminate x with u, a root of the irreducible polynomial in the field $\widetilde{\mathbb{Z}_2}$. To wit, $\varphi: \mathbb{Z}_2[x] \twoheadrightarrow \mathbb{Z}_2(u) \cong GF(2^m)$. There are exactly $(2^m - 2)/m$ such homomorphisms. To compute the fingerprint of a file, we 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 $\lfloor \frac{n}{m} \rfloor$ 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 giving 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%. 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. We initially describe the algorithm for arbitrary sized p; optimizations specific to $m = \deg p = 31$ will be described afterward.

Let T be the table of residues of all polynomials of the form $g(x) \cdot x^m$, where we allow the g to vary 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 from the input stream at a time instead of one at a time. View $f(x)$ now as

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

where $\deg a_i(x) < k$. The algorithm to compute the residue $r(x) = f(x) \bmod p(x)$ becomes the code shown in Exhibit 8.2.

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 Strongbox, we have two versions of the fingerprinting code, one for $k = 8$

```
$r(x) = 0$;
for ($i = \lceil \frac{n}{k} \rceil$; $i \geq 0$; --$i$) {
    $r'(x) = r(x) \cdot x^k + a_i(x)$;
    $r(x) = r'(x) \bmod p(x)$;
}
```

EXHIBIT 7.2
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 $r(x) = r''(x) + h(x)$.

and the other for $k = 16$, both of which used irreducible polynomials of degree 31. Because we want to read the input stream a full 32-bit word at a time, we 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. We 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 Exhibit 8.3 illustrates the algorithm for $k = 16$.

For the case where $k = 16$, the initialization of T will be time consuming if the simple brute force method is used. 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[15] close to a multiple of word size, though the number of working registers required (if implementing

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

```
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;
}
```

EXHIBIT 7.3
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 Strongbox, it omits loop unrolling, forces memory to be aligned, and may perform unnecessary memory references.

on a 32-bit machine) doubles. Our current implementation is largely limited by the main memory bandwidth on the 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 degrades dramatically.

7.8 Future Work

We have shown that the Strongbox system allows one to realize self-securing programs --- programs that can be run securely in environments that provide only minimal security. We have provided algorithms that substantially outperform existing algorithms. We have implemented our system in two different environments: the distributed transaction system Camelot, and the distributed operating system Mach.

What directions are next for the theory of self-securing programs? In addition to considering implementing Strongbox in other environments and putting more sophisticated access control mechanisms (such as those suggested in [26, 30]), we are continuing to consider basic research issues related to self-securing programs. We are pursuing two possible avenues for the future evolution of Strongbox: attacking the denial of service problem and using secure coprocessors in conjunction with Strongbox.

7.8.1 The Denial of Service Problem

Traditionally the concerns of availability and security have been thought to be contradictory [34]. To see one reason why, consider the use of *replication* to provide high availability. When we attempt to to guarantee a distributed system's availability the following problem arises: the larger the number of independently failing components, the smaller the likelihood they will all be working simultaneously, and the smaller the likelihood the system will be accessible when needed. This well-known phenomenon is typically addressed by designing distributed systems to be *fault-tolerant,* i.e., able to function correctly in the presence of some number of failures. In particular, the availability of long-lived data can be enhanced by storing the data redundantly at multiple sites, a technique commonly known as *replication* [12, 16].

But this physical distribution of security also makes security more difficult. When repositories for data are physically distributed it is more difficult to ensure that each one is physically secure. As the number of sites increases, so does the number of ways in which the secrecy and integrity of the data can be compromised.

In [17] we have proposed that we describe and analyze several encryption-based secrecy protocols that, for a given threshold value t, ensure that an adversary cannot ascertain the object's state by observing the contents of fewer than t repositories. We then extend these protocols to guarantee integrity, ensuring that the object's state cannot be altered by an adversary who can modify the contents of fewer than t repositories.

Our method successfully provides full file system security simultaneously with high availablity. The approach matches well with the Strongbox philosophy. We would like to develop techniques that provide other availability concerns.

7.8.2 Secure Coprocessors

One assumption we built on was the security of process memory. But building this into existing systems can be quite difficult: in fact it is often impossible to even promise the physical security of memory in a distributed system. In many distributed systems, some degree of physical security is provided by *ad hoc* approaches, but no completely satisfactory solution to this problem has been implemented.

In Strongbox, we use authentication protocols to provide security. But our authentication protocols, like all known authentication protocols, use some sort of key. The possession of this key is accepted as prima facie evidence of identity. However, it is usually impossible to guarantee the identity of all parties in the system, because the management of authentication keys makes certain implicit assumptions about physical security that do not hold in general. To see this, consider the management of a key used to establish the identity of a client machine to a server. At some point in the authentication, a key must be located somewhere. (This key might be formed by combining information from several different sources.) The key could be stored in the client machine, it could be held by the user, or it could be stored in some auxiliary device (such as a ‘‘smartcard’’ [22]) used by the user. If the key is stored in the client machine, it becomes vulnerable to physical attack. If an adversary is able to physically read memory or to load new system software that will allow him to examine memory locations, then he can find the value of the key. On the other hand, if the user, or an auxiliary device held by the user, holds the key then it remains to be seen how the user can trust the integrity of the client machine. The client machine may be running bogus software. Even an attempt to take a cryptographic checksum of the client machine will not prove the trustworthiness of the machine, since it is easy to create a pair of system software: one trustworthy and used for generating cryptographic checksums, the other untrustworthy and actually executed on the client machine.

Recently, a new architecture, called a secure coprocessor, has been proposed [42, 43]. A *secure coprocessor* is a processor and memory that is tightly coupled with the client machine, and that is physically protected. The physical protection can take various forms, but at the least we need a guarantee that the memory of the secure coprocessor is safe from attack. Any attempt to actually physically access the memory of the machine results in the memory being erased.

The secure coprocessor consists of a Central Processing Unit (CPU), some memory, and often some special encryption hardware. The secure coprocessor is realized as a board that is added on the bus of an existing workstation and that can work in tandem with the regular processor on that workstation. The secure coprocessor is protected by ‘‘tamper proof’’ packaging. Any attempt to penetrate the secure coprocessor will result in a total loss of the secure coprocessor memory. Thus a key that is stored on a secure coprocessor will remain secret, unless the secure coprocessor itself reveals the key. Traces of computation on the secure coprocessor, as well as intermediate values generated by that computation, also remain secret. New software can be stored on the secure coprocessor only by using previously established protocols; the secure coprocessor will reject any unauthorized new software. This architecture raises exciting possibilities for providing trust in systems.

The question of authenticating identity becomes much simpler, since authentication can be performed, using standard techniques, between the user and the coprocessor and between the server and the coprocessor. An initial authentication key can be stored in the secure coprocessor at the time it is manufactured, and this key can be used for future authentications. Furthermore, encryption can be done by the secure coprocessor, obviating a wide variety of key management and storage concerns. Secure software that is loaded on the client machine can be trusted by locating particular portions of the secure software on the coprocessor.

Indeed, it seems that secure coprocessor used with a system such as Strongbox could provide a large number of novel applications in computer security. We are extremely excited by the possibilites of this new technology and are actively pursuing new applications in ongoing research.

7.9 Acknowledgments

The authors had the privilege of learning about security and algorithms from Michael Rabin. Michael also suggested to us the idea of using a permutation of elements in a solution/puzzle vector in authentication protocols. For all this, and his advice and encouragement, we are deeply indebted to him. We owe another great debt to Alfred Spector for many discussions about the Camelot implementation of Strongbox. Alfred actively participated in the design phases of that implementation, and provided the facilities for us to integrate Strongbox into Camelot. Section 8.4.1 of our paper is based on a description prepared by Alfred Spector. Thanks also to James Aspnes for some helpful comments. Thanks also to Rick Rashid for his kind advice, and his Mach operating system which provided such a pleasant testbed for Strongbox.

This research was sponsored by the Defense Advanced Research Projects Agency under Contract No. F33615-87-C-1499 and by a National Science Foundation Presidential Young Investigator Award, Contract No. CCR-88-58087.

The views and conclusions contained in this chapter are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation, or the U. S. Government.

References

[1] M. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, Jr., and M. W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of Summer Usenix*, 1986.

[2] L. Adleman, K. Manders, and G. Miller. On taking roots in finite fields. in *19th IEEE Symposium on Foundations of Computer Science*, pp 715-177, October 1977.

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

[4] J. J. Bloch. The Camelot library: A C language extension for programming a general purpose distributed transaction system. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, 1989.

[5] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850-864, 1984.

[6] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *Proceedings of the Twelfth ACM Symposium on Operation Systems Principles*, 1989.

[7] B. Z. Chor. *Two Issues in Public Key Cryptography: RSA Bit Security and a New Knapsack Type System*. ACM Distiguished Dissertations. Cambridge, MA: MIT Press, 1986.

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

[9] J. L. Eppinger, L. B. Mummert, and A. Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. San Mateo, CA: Morgan Kaufmann, 1991.

[10] J. L. Eppinger and A. Z. Spector. Transaction processing in UNIX: A Camelot perspective. *Unix Review*, 7(1):58-67, 1989.

[11] U. Feige, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pp. 210-217, 1987.

[12] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating System Principles*, pp. 150-162. ACM, 1979.

[13] S. Goldwasser and S. 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.

[14] J. N. Gray. A transaction model. Technical Report RJ2895, IBM Research Laboratory, San Jose, California, 1980.

[15] N. Heintze. and J. D. Tygar. A critique of Burrows', Abadi's, and Needham's *A Logic of Authentication*. Carnegie Mellon University Technical Report, Carnegie Mellon University, Pittsburgh, PA, 1991.

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

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

[18] M. P. Herlihy and J. D. Tygar. Implementing distributed capabilities without a trusted kernel. In A. Avizienis and J. C. Laprie (eds.), *Dependable Computing for Critical Applications*. Vienna: Springer-Verlag, 1991.

[19] M. B. Jones, R. P. Draves, and M. R. Thompson. MIG --- the Mach interface generator. Mach Group document, Carnegie Mellon University, Pittsburgh, PA, 1987.

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

[21] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1981.

[22] J. A. McCrindle. *Smart Cards*. Berlin: Springer-Verlag, 1990.

[23] R. M. Needham. Using cryptography for authentication. In S. Mullender (ed.), *Distributed Systems*. New York: ACM Press and Addison-Wesley Publishing Company, 1989.

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

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

[26] M. O. 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, 1988.

[27] M. O. Rabin. Digitalized 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.

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

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

[30] M. O. Rabin and J. D. Tygar. ITOSS: An integrated toolkit for operating system security. In W. Litwin and H.-J. Schek *Foundations of Data Orgainzation and Algorithms*. Berlin: Spring-Verlag, 1989.

[31] M. O. Rabin. Private communication. 1987.

[32] M. O. Rabin. Probabilistic algorithms for testing primality. *Journal of Number Theory*, 12:128-138, 1980.

[33] M. O. Rabin. Probabilistic algorithms in finite fields. *SIAM Journal on Computing*, 9:273-280, 1980.

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

[35] R. F. Rashid. Threads of a new system. *Unix Review*, 4(8):37-49, 1986.

[36] 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.

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

[38] A. Z. Spector. Distributed transaction processing and the Camelot system. In Yakup Paker et al. (ed.), *Distributed Operating Systems: Theory and Practice*, Nato Advanced Study Institute Series --- Computer and Systems Sciences, pp. 331-353. Springer-Verlag, 1987. Also available as Carnegie Mellon Report CMU-CS-87-100, 1987.

[39] A. Z. Spector and P. M. Schwarz. Transactions: A construct for reliable distributed computing. *Operating Systems Review*, 17(2):18-35, 1983. Also available as Technical Report CMU-CS-82-143, Carnegie Mellon University, 1983.

[40] A. Z. Spector, D. Thompson, R. Pausch, J. L. Eppinger, R. Draves, D. Duchamp, D. S. Daniels, and J. J. Bloch. Camelot: A distributed transaction facility for Mach and the Internet --- an interim report. Technical Report CMU-CS-87-129, Carnegie Mellon University, Pittsburgh, PA, 1987.

[41] J. D. Tygar and B. S. Yee. Strongbox. In J. L. Eppinger, L. B.Mummert, and A. Z. Spector (eds.), *Camelot and Avalon: A Distributed Transaction Facility including the Avalon Language*. San Mateo, CA: Morgan Kaufmann, 1991.

[42] S. H. Weingart. Physical security for the μabyss system. In *Proceedings of the IEEE Computer Society Conference on Security and Privacy*, pp. 52-58, 1987.

[43] S. R. White and L. Comerford. Abyss: A trusted architecture for software protection. In *Proceedings of the IEEE Computer Society Conference on Security and Privacy*, pp. 38-51, 1987.