Strongbox:
A Self-Securing Protection System for Distributed Programs

Bennet S. Yee, J. D. Tygar, Alfred Z. Spector

4 January 1988

Technical Report CMU-CS-87-184

Department of Computer Science

Carnegie-Mellon University

Pittsburgh, PA 15213

Abstract

We introduce a new method of approaching security in distributed systems: *self-securing programs.* These programs run securely on distributed operating systems which provide only minimal security facilities. We have built a system called Strongbox to support self-securing programs on Mach, a distributed operating system, and Camelot, a distributed transaction facility. Strongbox uses negligible overhead and is relatively small, making the task of verification easier. Our paper presents the implementation of self-securing programs on Mach and Camelot and an overview of the algorithms used. We describe the performance and the current status of Strongbox.

This research was sponsored by IBM and the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864 (Amendment 20), under contract F33615-87-C-1499 monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB.

1. Introduction

Security is a pressing problem for distributed systems. Distributed systems exchange data between 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, ...* 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 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. It is hard just to make a distributed system work efficiently without considering security issues.

We want to construct and use trusted application systems that run efficiently on machines having have only minimal security facilities. Rather than depending on a tight kernel of security code, our applications perform operations that allow us to guarantee computer security. We call these trusted programs *self-securing*. By limiting dependence on underlying system components, we separate the security problem from the rest of the system, simplify the task of the engineer who must build such a system, and allow existing distributed systems to be retro-fitted with security. Our concern here is with security issues arising from protecting the privacy of data and the integrity of data from alteration. We do not consider issues of denial of service, covert channel analysis, or traffic analysis of message patterns. An important assumption in this work is the integrity and privacy of address spaces.

We have developed a family of algorithms that support self-securing programs. To show the effectiveness and efficiency of our methods, we have implemented them in a package called *Strongbox* and measured their performance in our computational environment. We have successfully built on two ongoing systems research projects at Carnegie Mellon University: Mach, a distributed operating system which is upwardly compatible with UNIX [27]; and Camelot, a distributed transaction facility which runs on Mach [36, 35]. Although the Mach project addressed several important security issues [30], it is not a secure system. Our tools allow users to run application program securely on Mach and Camelot. To demonstrate our tools, we have built a capability based protection system for Camelot which runs with negligible overhead. This protection system will be publicly distributed with Camelot.

After reviewing a few facts about Camelot and Mach, we give the architecture of secure programs and their application to Camelot. We then present an overview of two of the basic algorithms that self-securing programs use: zero knowledge authentication and fingerprinting. We give our performance measurements for the system and discuss the current status of the software. We conclude by presenting applications and limitations of our system, the prognosis for applying self-securing programs to other

distributed systems, and possible self-securing approaches to denial of service attacks.

2. Architecture

This section describes the architecture of Strongbox. Section 2.1 provides a substantial amount of background on Mach and Camelot that is necessary for understanding the architecture of Strongbox. Section 2.2 discusses the architecture of Strongbox.

2.1. Mach and Camelot

We assume that that distributed processing is organized according to the *client-server* model [38]. In this model, *client* processes access shared data and most system services by sending messages to *server* processes. Servers then perform computations on behalf of their clients, and send response messages containing the results. *Remote procedure calls* or (RPCs) [4] reduce the programming effort for handling request and response messages by permitting programmers to invoke services using procedure call syntax. An RPC compiler generates the code to do message packing, unpacking, and dispatching.

The Mach operating system [1, 27] provides the basic services to support the client-server model. Mach is upward compatible with Unix 4.3 bsd, but provides additional primitives for supporting multi-threaded processes (called tasks), location-transparent inter-task communication and efficient virtual memory. Mach is also relatively machine-independent, and it runs on a variety of uni- and multi-processors including the RT PC, Sun 3, Vax, and Multimax. The work in this paper was done primarily on Mach running on RT PCs and Vaxes.

The Mach Interface Generator (MIG) is Mach's RPC stub generator [14, 22]. MIG accepts a syntactic specification for procedure headers (written in a Pascal-like syntax), and generates libraries for calling and dispatching RPCs. These libraries are linked into the executable image of both clients and servers.

The Camelot distributed transaction facility is layered on Mach and MIG. It provides mechanisms for constructing reliable distributed programs that access shared data [36, 33, 35]. 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 attention a programmer must pay to concurrency and failures [12, 34]:

1. **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.

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

3. **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.

Camelot's primary interface for programmers consists of macros and procedures in the Camelot library. Primitives such as `BEGIN_TRANSACTION ... END_TRANSACTION` and `SERVER_CALL` permit programmers respectively to execute a transaction and to issue an RPC to a remote server. Programmers can also declare recoverable objects in their C programs by using a special storage allocator, or surrounding declarations within `BEGIN_RECOVERABLE_DECLARATIONS ... END_RECOVERABLE_DECLARATIONS`. Camelot distinguishes between *data servers*, which can execute transactions, issue RPCs, and store shared, recoverable objects, and *applications* which have no recoverable data, but are otherwise like servers.

The Camelot library is assisted by functions provided by eight Camelot tasks. The Camelot and master control tasks control the execution of Camelot itself. The node server is the repository of configuration data necessary for restarting a node. It stores its data in recoverable storage and is recovered before other servers. The (interactive) node configuration application permits Camelot's users to update data in the node server, thereby controlling the operation of other servers.

Of more importance for this paper are the Camelot communication manager, recovery manager, transaction manager, and disk manager. The communication manager forwards inter-node Mach messages, and provides logical clock services. In addition, it knows Camelot's message format and keeps a list of all the nodes that are involved in a particular transaction. The recovery manager is responsible for recovering the system after transaction aborts, server and node failures, and media failures. The recovery manager has available to it a sequence of log records that were written during normal processing. The transaction manager coordinates the initiation, commit, and abort of local and distributed transactions. It fully supports nested transactions.

The Camelot disk manager is the central Camelot component concerned with permanent data. It loads and starts servers; allocates and deallocates permanent storage for long-lived objects; and accepts and writes log record for recovering from failures. For log records that are to be written to the distributed logging service, the disk manager works with dedicated servers on a local area network. Additionally, the disk manager writes pages to/from the disk when Mach needs to service page faults on recoverable storage or to clean primary memory.

2.2. Architecture of Strongbox

Strongbox is implemented by three components: a secure loader, a library of security routines that secure servers and secure clients use, and a white pages server (a repository of essential authentication information). Figure 2-1 shows the components of the Strongbox and their relationships to each other, and to Mach and Camelot.

When requested to run a secure task, the secure loader first copies the load image of the task into memory. It then verifies that the load image has not been corrupted while it was stored in the file system. To do this, it calculates a cryptographic checksum, called a fingerprint. After the fingerprint has been verified, the secure loader initializes the task's address space. At this time it stores with the task the solution to a randomly generated authentication puzzle which will be later used for authenticating the task to others, and checks in the puzzle (but not the solution) with the white pages server. Hence the puzzle solution never appears in the file system. (Section 4 describes the fingerprinting and authentication algorithms.) Finally, the secure loader starts the task.

For Camelot applications, the secure loader is a separate program that runs directly on Mach. For servers, the secure loader is part of the Camelot disk manager, which Camelot already uses to load servers. Because the secure loader is incorporated in the disk manager, we assume the integrity of the disk manager.

The security library routines perform several tasks: They maintain the access control database for the server; they authenticate the identities of the clients and the server; and they maintain the database of capabilities. The protection system uses these capabilities to check access permissions.

The white pages server maintains a list of tasks and the authentication puzzles which identify them. Whenever an agent is to be authenticated, the authenticator asks the white pages server for the authentication puzzle. The white pages server also maintains the fingerprints of the load images for the secure loader.

The white pages server, the secure loader, and Camelot are all started up at boot time before any user tasks can be started. At this time, the secure loader and Camelot generate new authentication puzzles/solutions for themselves, and check in the puzzles with the White Pages server. It is possible to substitute a process's name server port so that its port lookups will give erroneous results. Such substitution, however, does not corrupt the servers. To run a client that a server will listen to (i.e., server will find a puzzle for it when it looks you up in the white pages), the client must be started up by the secure loader since the white pages will accept new entries only from the secure loader.

A typical session is as follows: The client wishes to invoke a remote operation by a server to which it

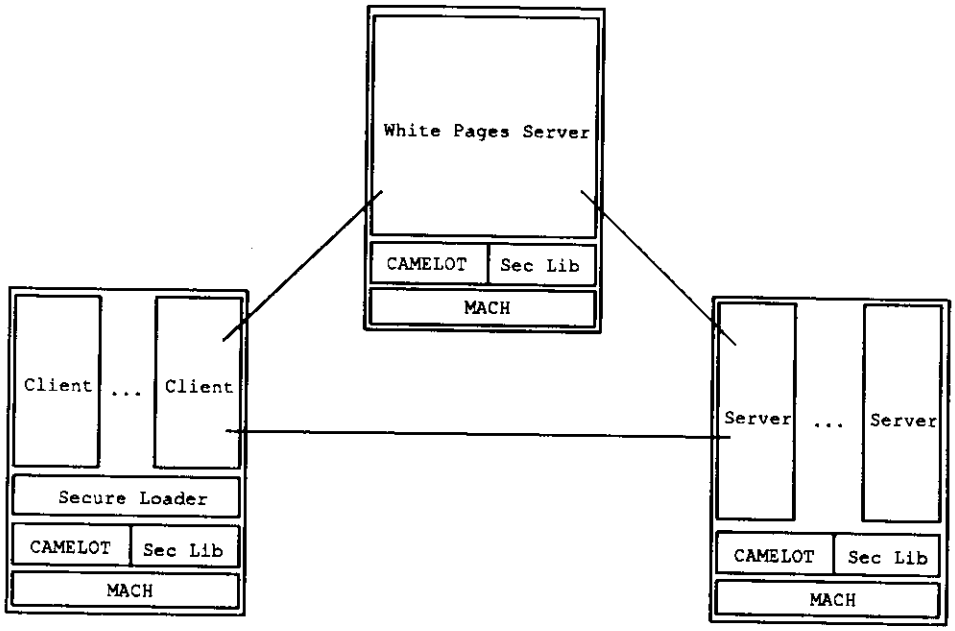

Figure 2-1: Strongbox Architecture

has not authenticated itself. It uses a secure RPC handling macro, which attempts the operation. The operation fails with an error return of `CAP_MECH_PERMISSION_DENIED`, at which point the macro looks up the server's authentication puzzle with the white pages and authenticates the client and the server identities to each other. At the server side, the protection system also looks up the client's puzzle with the white pages as part of the authentication. After authentication completes, all traffic between clients and servers are encrypted.

Depending on the security of the paging devices, external pagers may protect the integrity of memory by fingerprinting pages as they go onto disk and verifying the fingerprint as they come off.

2.3. Security Library Architecture

The security library provides client-server authentication, capability-based access to a server, and access control bookkeeping routines used in a server. We have designed the security library such that typical usage is very similar to that of the standard, non-secure Camelot library.

In the Camelot library, the `SERVER_CALL` macros handle the transactional bookkeeping for invoking an RPC. For secure communication between the client and the server, the various versions of Camelot's

SERVER_CALL macros are supplanted by versions that are prefixed by SEC_. (For example, SERVER_CALL becomes SEC_SERVER_CALL.) The SEC_ macros automatically perform authentication and provide the client with a capability to the server. The capability is used transparently; the client program need not reference it at all. The secure macros automatically re-authenticate after server failure, thereby allowing transparent recovery for most applications.

The client may elect to perform explicit authentication instead of using the automatic authentication provided. This gives the client control over the management of the capabilities: a client may authenticate once as itself and another time as a member of a privileged group. By using versions of the SERVER_CALL macros prefixed by SEC_CAP_, a client may specify the capability to be used with the server. This gives a client the ability to use the access rights of two or more agents. The details of authenticated SERVER_CALLS are described in the following section.

The server-side protection system is based upon two databases both maintained by the capability component. The first, stored in recoverable storage, contains a mapping from user names to permissions. When the authentication component asserts to the capability component that a client has authenticated, the capability component constructs a new capability based on the permissions in this database. The capability component gives it to the authentication component to return to the client and also enters this new capability into the second database. This second database, stored only in virtual memory, maps capabilities to permissions. The capability component consults this database each time an RPC request arrives to check that the client has permissions to invoke the requested operation.

It is easy to convert an existing non-secure server to be secure because only the initialization code is affected. Strongbox provides a separate message demultiplexing procedure that performs the permission check. The procedure then hands the messages over to the individual subsystems' demultiplexing procedures (which unpack the arguments and make the actual server-side procedure call). This service is performed for all RPCs except those in the authentication protocol. By performing the permission check at this point, we eliminate the need to modify the service procedures, and reduce the work required to make a server secure.

3. Interfaces

In this section, we discuss in turn the local interface between a server and the protection system; the local interface between a client and the protection system; and the remote procedure interface between secure clients and secure servers.

3.1. Server/Protection System Interface

The server must first allocate recoverable and non-recoverable storage for the protection system. A server must then initialize the authentication system by calling `Sec_SetServerParam`:

Declaration

```
EXPORT Sec_SetServerParam(IN secAuthPred,secAuth,secPubAccess,secPrivate)
int     (*secAuthPred)();       /* (*secAuthPred)(ClaimedID,capPtr) */
int     (*secAuth)();           /* (*secAuth)(ProvenID,capPtr) */
int     (*secPubAccess)();      /* (*secPubAccess)(ID,&Pub) */
sec_priv_t      *secPrivate;
```

The authentication subsystem invokes the function `secAuthPred` each time a user attempts to authenticate. A nonzero return value from `secAuthPred` indicates that the authentication system must authenticate the user. If `secAuthPred` returns zero, then no authentication is required. Permitting unauthenticated users makes it possible to provide some globally available basic services to non-registered users.

The authentication subsystem invokes the function `secAuth` whenever an authentication has been completed. This is the way the authentication system asserts that the client has authenticated. Normally, this is the function `CapMech_Authenticated`, which the protection system supplies. However, the server may provide another function, which can perform auxiliary operations as well; e.g., logging.

The authentication subsystem invokes the function `secPubAccess` whenever it needs to look up the published puzzle of a client. `secPubAccess` normally obtains this information from the white pages server. The authentication subsystem looks up published puzzles whenever a client authenticates.

The private information pointed to by `secPrivate` is the private solution to the published puzzle for the server. It is used when a client wants to authenticate the server.

After initializing the authentication system, the server initializes the capability system by calling `CapMech_Init`:

Declaration

```
EXPORT void CapMech_Init(tbl,sz)
        cap_mech_symtab_t       *tbl;
        int                     sz;
```

The argument `tbl` is a table that contains an entry for each exported remote procedure call subsystem. Each entry in the table contains the mapping from request codes to service procedures and the normal dispatch routine used to parse request messages that is generated by MIG for that subsystem.

After initializing the non-recoverable part of the protection system, the server performs the initializing transaction in the INITIALIZE_SERVER macro provided by the Camelot library, in which the server initializes the recoverable part of the capability subsystem. In this transaction, the server invokes the RECOVER_RCV_HEAP macro provided by the recoverable memory allocator package (because the protection system requires dynamic memory allocation), and then the CAP_MECH_RCVR_INIT macro to initialize the capability subsystem's recoverable store. This will also create the administrative user's entry (admin) and initializes his permissions.

3.2. Client/Protection System Interface

A client of a secure server must have an authentication solution to a puzzle that secPubAccess returns. This must be stored in the global variable _ClientPzSoln and is initialized by the secure loader [2]. Instead of using SERVER_CALL and other similar macros to issue RPCs, the client uses macros such as SEC_SERVER_CALL.

The client is given a capability containing all of his privileges when he authenticates. The capabilities not only saves the overhead of re-authenticating, but they also give more flexibility to the system in that (limited) capabilities may be given away to another client. The SEC_ macros perform the authentication when necessary and cache the capability automatically, so normally the client does not need to do anything else special. If a client wishes to explicitly authenticate itself to a server (perhaps using a group id), the function Sec_Authenticate may be invoked explicitly.

Declaration

```
EXPORT int Sec_Authenticate(IN servPort,transID,waitTime,
                            OUT capPtr,IN secPublic,secPrivate,
                                secAuthenticateServer)
port_t          servPort;
tid_t           transID;
unsigned int    waitTime;
cap_t           *capPtr;
sec_pub_t       *secPublic;
sec_priv_t      *secPrivate;
int             secAuthenticateServer;
```

There are corresponding SEC_CAP_ prefixed macros which may be used in place of the SERVER_CALL macros. The argument macro should be SECCAPARGS or SECCAPNOARGS.

3.3. Client/Server Interface

In addition to the normal service remote operations that a server provides, each secure server has other remote procedures that are defined by the capability subsystem and the authentication subsystem. The capability subsystem exports remote operations to perform access control administration such as adding a new user, giving a user the rights to invoke a remote operation, or removing a user's rights. Since the protection system's remote operations are treated in the same manner as the service remote operations, the administrator may give a user administrative privileges as well. The authentication subsystem exports two remote calls for performing the zero-knowledge authentication.

There are currently three remote routines which allows the administrative login to manipulate the access permissions list.

Declaration

```
EXPORT void op_CapMech_AddUser(authCap,userName)
        cap_t                          authCap;
        cap_symbolic_user_name_t       userName;
```

Declaration

```
EXPORT void op_CapMech_Permit(authCap,userName,operation)
        cap_t                          authCap;
        cap_symbolic_user_name_t       userName;
        cap_operation_name_t           operation;
```

Declaration

```
EXPORT void op_CapMech_Unpermit(authCap,userName,operation)
        cap_t                          authCap;
        cap_symbolic_user_name_t       userName;
        cap_operation_name_t           operation;
```

The operation `CapMech_AddUser` adds a new user; `CapMech_Permit` gives a user permission to invoke the named operation; and `CapMech_Unpermit` removes the user's permission to invoke the operation.

Note that these operations are treated in the same fashion as service remote operations, so the administrator may give permissions to invoke `CapMech_Unpermit`, for example, to security personnel.

4. Algorithms

4.1. Zero Knowledge Authentication

Authentication is at the heart of the security system for any loosely-coupled distributed operating system. For notational convenience, we refer to a client denoted A and a server denoted B. A and B may reside on different processors. A and B must prove their identities. The problem is made more difficult since A and B must typically accomplish this by exchanging messages over a potentially vulnerable data network. Since messages transmitted over the network may be intercepted by a third party, C, A and B must find a way to prove their identities without revealing information which would allow C to successfully feign an identity as A or as B.

How well do existing authentication methods accomplish this goal? In practice, not very well. For example, Rivest, Shamir, and Adleman proposed an authentication method based on the RSA public-key signature methods [29]. In their protocol, values are encrypted according to a public-key encryption function $E(m)=m^e \bmod n$, where m is a message, e is an encryption key, and n is the product of two large primes p and q. Decryption is accomplished through the function $D(c)=c^d \bmod n$, where c is the ciphertext, d is chosen so that $ed=1 \bmod (p-1)(q-1)$. It is true there is no published method for quickly decrypting messages given only e and n, and not the factorization of n. However, this method leaks information. For example, Lipton points out that the well known Legendre function L satisfies the relation $L(m,n)=L(E(m),n)$ [18]. Indeed, the problem is much worse than this. Alexi, Chor, Goldreich, and Schnorr recently proved that if an adversary can find the low order bit of m $50\%+\varepsilon$ of the time given $E(m)$, the adversary can invert arbitrary RSA encryptions [2, 6]. A corollary to this result is that the usual query-response methods for encryption, such as the family of protocols described in [20], an adversary can emulate another agent in the system after engaging in $O((\log n)^2)$ authentications.

Needham and Schroeder have suggested an authentication method which uses private-key cryptographic methods [21]. Needham and Schroeder's work presupposes on a secure key distribution method and a private key cryptosystem. Recent work by Luby and Rackoff suggests that authentication methods depending on DES are vulnerable to a "low-bit" attack similar to the one mentioned above for the RSA cryptosystem [19]. For example, the second author has found a method to subvert the authentication scheme used by the Andrew File System VICE [32, 31], which uses a strategy similar to Needham and Schroeder's.

To give users confidence in a system, we would like to be able to prove that an authentication method does not leak information. Several researchers have independently proposed protocols, termed *zero-knowledge protocols*, which satisfy this constraint given the complexity assumption that P $\neq$ NP [10, 3, 11, 7]. To get a flavor of the type of argument used, we summarize a zero knowledge protocol below for A proving to B that some graph G with n vertices known to both A and B contains a k-clique (that is, a set Q of k vertices such that between every two vertices in Q there exists an edge). (This version of

the proof is due to M. Blum.) Let G be a graph with n vertices be known to both A and B. Suppose that A knows a k-clique in G. Since the problem of finding a k-clique in an arbitrary graph is NP-complete, B can not in general find the k-clique. This protocol will allow A to prove to B that G has a k-clique without revealing any info about the vertices in Q.

1. A secretly labels each vertex of G with random unique integer from 1 to n.

2. A prepares $n(n-1)/2$ envelopes labeled uniquely with a pair of integers $\langle i,j\rangle, i<j$. A puts "Yes" in the envelope labeled $\langle i,j\rangle$ if an edge exist between the vertices labeled i and j, and "No" otherwise.

3. A seals the envelopes and presents them to B. B flips a coin and reports its value to A. If it is heads, A must open all the envelopes and show the numbering of the vertices of G. B then verifies that the descriptions are correct. On the other hand, if B gets tails, A must then open just the envelopes which are labeled $\langle i,j\rangle$ where i and j belong to Q. B then verifies that all envelopes contain "Yes".

4. The above protocol is repeated t times (with an independent random numbering assigned each time in step 1). If A successfully responds to B's queries, the probability that A does know know a proof is 2^{-t}.

It is clear that if A knows a k-clique and correctly follows the above protocol, A will succeed. On the other hand, suppose C is trying to masquerade as A. Since C does not know a k-clique, C has two choices: it can correctly perform step 2 (in which case it is caught whenever B gets tails) or it can put false values in the envelopes (in which case it is caught whenever B gets heads). Hence in each of the t iterations of the protocol the probability that C's ignorance is revealed is 1/2. After t iterations, C will be caught with probability $1-2^{-t}$. Finally, notice that B does not get any information about the location of the clique. If B could find information about the clique from the above protocol, it could generate the same information by flipping a coin and generating a random numbering of the graph when it gets heads and a random numbering of a complete graph on k vertices when it gets tails.

Notice that since any problem lying in NP can be reduced to the NP-complete problem k-clique [15], A could use this protocol to prove to B that it had a proof or disproof of, for example, Fermat's Last Theorem. At the end of this protocol, B would be convinced that A did in fact have a proof or disproof without having any idea which way the problem was resolved, much less any idea of the technique used to solve the problem. Certainly, this protocol could be used to generate authentication proofs: A would publish the graph G in a public white pages. To prove its identity, A would give a zero-knowledge proof of the existence of a k-clique in the graph.

In practice, this protocol would not work well. First, A would have to find a graph G in which it was computationally intractable to find a k-clique. While it is true that our complexity hypothesis guarantees that such graphs must exist, most random graphs with k-cliques can have those cliques found through

efficient heuristics [9]. Second, A and B would have to develop a good cryptographic scheme for implementing "envelope exchange". For even a modest security level the size of data involved here is on the order of 10^{200} bytes. Using the highest bandwidth transmission techniques available today, execution of this protocol would exceed the time remaining before the heat death of the universe.

4.2. Our protocol

In research described in [37], we have developed a family of zero-knowledge protocols which are efficient for real use in applications. Our timing figures are given in Section 5. Below we give a simplified (and slightly less efficient) version of the protocol.

The protocol we use can depend on one of two complexity assumptions: that factoring large integers can not be done in polynomial time, or that it is hard to invert messages encrypted by random keys under DES. Other similar complexity assumptions may be used instead. The protocol described below depends on the complexity of factoring integers. We recall the following lemma by Rabin [23]: *If there exists a polynomial time algorithm for finding square roots modulo $n=pq$, where p and q are large primes, then we can factor pq in polynomial time.* Rabin observed that we can take a random integer r between 1 and $pq-1$; check that $GCD(r,n) \neq 1$ (if this value is p or q, then we have factored n). Calculate $x=r^2 \bmod n$ and find a square root s, so that $s^2=x=r^2 \bmod n$; a simple number-theoretic argument demonstrates that x has four square roots modulo n, including r and $-r$. Since r is chosen at random, there is a 50% chance that $s \neq \pm r \bmod n$. If $s=\pm r \bmod n$, then we can pick a new r and repeat the algorithm. If $s \neq \pm r \bmod n$, then it is the case that $GCD(r+s,n)=p$ or q. Hence finding square roots is equivalent to factoring.

In this protocol, we assume that the system manager publishes a product of two large primes $n=pq$, keeping the factorization secret. This n can be used for all authentication protocols, and no one need ever know its factorization. To initialize its puzzle, A picks a random r and publishes $x=r^2$ in the white pages. A will prove it knows a square root of x without revealing any information about the value.

Here is the protocol:

1. A computes t temporary random values, $v_1, v_2, \ldots, v_t$, where each v_i satisfies $1 \leq v_i \leq pq-1$. A sends to B the vector $\langle v_1^2 \bmod n, v_2^2 \bmod n, \ldots, v_t^2 \bmod n \rangle$.

2. B flips t independent coins and send back a vector of t random bits $\langle b_1, \ldots, b_t \rangle$ to A.

3. For $1 \leq i \leq t$ A computes

$$z_i = \begin{cases} v_i & \text{if } b_i=0 \\ r v_i \bmod n & \text{otherwise} \end{cases}$$

A transmits the vector $\langle z_1, \ldots, z_t \rangle$.

4. B verifies that for $1 \le i \le t$, that

$$z_i^2 = \begin{cases} v_i^2 \bmod n & \text{if } b_i = 0 \\ x v_i^2 \bmod n & \text{otherwise} \end{cases}$$

If the equalities hold, A has authenticated its identity to B with probability $1-2^t$.

Once again note that if A knows a value r such that $r^2 = x \bmod n$, then it can easily follow the above protocol. Suppose C is trying to masquerade as A. Since C doesn't know such a value r, it can not know both v_i and $r v_i \bmod n$, since $r = r v_i (v_i)^{-1} \bmod n$. Finally, all that B sees is a series of random values of the form $\langle z_i, z_i^2 \bmod n \rangle$. If B could find any information about r from the above protocol, it could do so by generating a set of t random values and squaring them modulo n, and thus factoring n.

The above protocol uses only an expected $3t$ multiplications to generate security of $1-2^{-t}$. Our improved protocol uses only expected $1.5t$ multiplications to achieve the same level of security.

4.3. Self securing programs

As mentioned above, our algorithm presumes communication networks which are potentially vulnerable. A and B need to use methods to protect the privacy and integrity of their data. If we extend an authentication algorithm to also support key exchange, A and B can transmit their messages though highly secure private-key encryption methods. We have adapted our algorithms to also perform this operation -- A can send a temporary key e_A and B can send a temporary key e_B. Both parties can then use a trusted private key encryption method, such as DES with the key $e = e_A \oplus e_B$, where $\oplus$ is bit-wise exclusive-or. Hence, if A and B have protected address spaces, we can make all messages transmitted in the system public, since no observer can find the encryption key used. Also, sending messages encrypted by the key removes the need to re-authenticated until either party decides to establish a new temporary key. This approach yields a *self securing program* which requires only a minimal amount of security in our base operating system.

Indeed, our algorithm obviates the need to ever use public key cryptography. If A wishes to transmit a message to B without having shared private key, A can simply authenticate itself to B exchanging a temporary encryption key. All further communications are protected by encryption. The signature functions of public key cryptography can be performed by the fingerprinting algorithm given below.

4.4. Fingerprinting

Karp and Rabin introduced an algorithm which computes a *cryptographic checksum*. [25, 17, 16] Their algorithm takes a bit string s of arbitrary length and secret key k of d bits (where $d-1$ is prime) and returns a *fingerprint* sequence of $d-1$ bits $\phi_k(s)$. Each key k defines a fingerprint function, and if the keys are

chosen with uniform distribution, the family of fingerprint functions ϕ_k can be viewed as a provably good random hash function in the style of [5, 28]. Without the secret key, computing a fingerprint given a string of bits is intractable. On the other hand, if the secret key is known, it is easy to compute the fingerprint.

Given the fingerprint algorithm, the problem of protecting the integrity of data from alteration becomes much simpler. For example, to protect a file F, we could store $\langle F,\phi_k(F)\rangle$. If an adversary attempts to alter the file by replacing it with F', he will need to calculate $\phi_k(F')$. But since the adversary does not know k, he can not compute the fingerprint of F'. Even if the adversary attempts to find a F' with the same fingerprint $\phi_k(F)=\phi_k(F')$, he will be thwarted, since the problem of finding an input which generates a given fingerprint is intractable without the key value k.

The fingerprinting algorithm views a sequence of bits s as a polynomial $f_s(x)$ over the integers modulo 2. For example, the bit sequence s="100101001" is taken to be the polynomial $f(x)=x^8+x^5+x^3+1$. The secret key for this algorithm corresponds to a random irreducible polynomial $g(x)$ of degree $d-1$ over the integers modulo 2. It is extremely easy to generate these polynomials (several approaches are outlined in [24, 25]) and we have implemented two different efficient routines for doing so. Compute $r(x) = f_s(x) \bmod g(x)$. $r(x)$ is a polynomial over the integers modulo 2 of degree at most $d-1$. Both the polynomial $r(x)$ and the key k can be represented as a string of d bits. The bits produced by the algorithm define the ϕ function.

Because this algorithm can be conveniently implemented as a systolic array, a group of researchers led by H. T. Kung chose it to implement in hardware [8]. By using new techniques, we have software implementations of this algorithm which run on the IBM RT/APC in time comparable with the fastest hardware implementation (see Section 5).

5. Implementation Issues and Performance

This section gives timing figures for the prototype implementations of authentication and fingerprinting. The authentication timings given are for the current version of the authentication routine, which uses the algorithm in [7]. Our timing figures are for an IBM-RT/APC, which is a RISC machine running at 4 MIPS.

An IBM-RT/APC requires 105 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 msec [36]. We have an efficient software implementation of DES which works at a rate of 220 encryptions per sec.

An IBM-RT/APC achieves a fingerprinting rate of 800 KByte/sec. The fingerprinting routine uses a 65536 (2^{16}) entry table of pre-computed partial residues to achieve this speed. The table contains the value $x^{32}\, a(x) \bmod g(x)$ where $a(x)$ is the two byte index considered as a polynomial modulo 2, and $g(x)$ is

the irreducible polynomial used (in our implementation, $deg(g(x)) = 31$). The inner loop simply reads data two bytes at a time and uses the value as an index into this table to obtain the partial residue to exclusive-or into a running residue.

A brute force initialization of the 65536-entry table would take considerable time. To initialize the 65536-entry table efficiently, we first compute 256-entry table of partial residues. The 256-entry table contains the residues $x^{32}\, a(x)$ mod $g(x)$ where $a(x)$ is the byte index considered as a polynomial modulo 2, and $g(x)$ is the irreducible polynomial as before. The larger table is computed from the smaller by indexing into the smaller table first by the upper 8 bits of the larger table index, and then indexing again using the lower 8 bits of the larger table index exclusive-or'ed with the upper 8 bits of the result of the previous look-up. The initialization cost is approximately 1 sec.

In security, smaller code size is desirable since the system is easier to verify and is less likely to contain bugs. The core authentication routines consists of 75 lines of C code not including comments. The fingerprinting code consists of 211 lines of C code not including comments. Our total core routines are relatively small: 248 lines of C code.

6. Status and Future Work

We have implemented the high speed zero knowledge authentication algorithm, the fingerprinting algorithm, the protection system, and a preliminary version of the white pages. The secure loader remains to be implemented. There are limitations in the prototype that will be removed in the January 1988 version:

A client may currently have only one capability; the `SEC_CAP_` prefixed macros have not been written. The architecture of the prototype easily allows multi-capability clients. The only thing necessary is to provide an interface where the capability used in the server-call macros is visible.

The problem of network spying must be addressed by using encryption. This will be done by using end-to-end encryption within the RPC stub generator (MIG). No key exchange takes place in the current implementation of the authentication algorithm.

With the prototype, the only way to revoke privileges is to shut the server down. This is because capabilities retain their rights even when the "source" user's permissions go away. The problem of revocation of capabilities is easily fixed: since a lookup must be done to check the capability each time an RPC is performed, we can simple delete the capability from server's database and the errant client will find that its request will fail due to `CAP_MECH_PERMISSION_DENIED`. We will provide functions `op_CapMech_RevokeAll` and `op_CapMech_RevokeCap` to revoke capabilities.

7. Conclusion

We have demonstrated that self-securing programs are an effective and efficient solution to the distributed computer security problem. Our approach is aimed at the client-server model, and could be straightforwardly ported to other client-server systems. While we have not attempted to address other models of distributed computation, any system in which various agents communicate or store data may be able to use some of algorithms.

In building our security facility, we needed to make a number of assumptions. First, we made a complexity assumption that some problem, such as factoring large integers or inverting DES, is intractible. We assumed that our base operating system, Mach, supports protected address spaces, including virtual address spaces stored on a disk. We assumed that the Camelot disk manager and the loader does not reveal authentication information or fingerprint keys to any other agent. We assumed that application programs use our protection scheme uniformly, and do not explicitly bypass protection mechanisms by revealing confidential information by writing out values in unencrypted log files or other unencrypted public files. We assumed that our algorithms were implemented without error, and that the compiler produced correct object code for them.

We did not address issues of denial of service, covert channel analysis, or of leaking information through traffic analysis of messages. Although we have not explicitly addressed these problems, we conjecture that they may be solved by approaches motivated by the self-securing paradigm. For example, Camelot supports fault tolerance, and Strongbox makes that fault tolerance secure. We believe that the security facility on Camelot could be extended to support protection against denial of service attacks. (For some theoretical contributions to this these issues, see [13, 26].) In loosely-coupled distributed systems, covert channel analysis may be considerably simplified by assigning to entire processors just a single security classification. Interactions between security levels will take place over the data network, which is a simpler object to examine for covert channels than the entire distributed operating system. We are continuing to explore approaches such as these in ongoing research. Our current security code is publicly available and will be included with the Camelot release. We will continue to examine its performance in large applications.

8. Acknowledgement

We are indebted to Michael Rabin for a number of helpful comments and ideas about authentication systems. Thanks to Roger Needham for a useful discussion about key distribution. We are grateful to David Applegate for allowing us to use his high speed DES implementation. Thanks to Marc Ringuette for proofreading a preliminary version of this paper.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.

[2] W. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr.
RSA and Rabin Functions: Certain Parts are as Hard as the Whole.
In *Proceedings of the 25th IEEE Symposium on the Foundations of Computer Science*. November, 1984.
To appear in SIAM J. on Computing.

[3] Laszlo Babai.
Trading Group Theory for Randomness.
In *Proceedings of the 17th ACM Symp. on Theory of Computing*, pages 421-429. May, 1985.

[4] Andrew D. Birrell, Bruce J. Nelson.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems 2(1):39-59, February, 1984.

[5] J. Carter and M. Wegman.
Universal Classes of Hash Functions.
In *Proceedings of the 17th IEEE Foundations of Computer Science*, pages 106-112. May, 1976.

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

[7] Uriel Feige, Amos Fiat, and Adi Shamir.
Zero Knowledge Proofs of Identity.
In *Proceedings of the 19th ACM Symp. on Theory of Computing*, pages 210-217. May, 1987.

[8] A. L. Fisher, H. T. Kung, L. M. Monier, and Y. Dohi.
Architecture of the PSC: a Programmable Systolic Chip.
Journal of VLSI and Computer Systems 1(2):153-169, 1984.

[9] A. M. Frieze.
Parallel Algorithms for Finding Hamilton Cycles in Random Graphs.
Information Processing Letters 25:111-117, 1987.

[10] Shafi Goldwasser, Silvio Micali, and Charles Rackoff.
The Knowledge Complexity of Interactive Proof Systems.
In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. May, 1985.

[11] S. Goldwasser and M. Sipser.
Arther Merlin Games versus Zero Interactive Proof Systems.
In *Proceedings of the 17th ACM Symp. on Theory of Computing*, pages 59-68. May, 1985.

[12] James N. Gray.
A Transaction Model.
Technical Report RJ2895, IBM Research Laboratory, San Jose, California, August, 1980.

[13] Maurice P. Herlihy and J. D. Tygar.
How to Make Replicated Data Secure.
In *Advances in Cryptology, CRYPTO-87*. Springer-Verlag, August, 1987.

[14] Richard P. Draves, Michael B. Jones, Mary R. Thompson.
MIG - The Mach Interface Generator.
1987.
Mach Group document.

[15] R. M. Karp.
Reducibility among Combinatorial Problems.
Complexity of Computer Computations.
Plenum Press, New York, 1972, pages 85-103.

[16] Richard M. Karp.
1985 Turing Award Lecture: Combinatorics, Complexity, and Randomness.
Communications of the ACM 29(2):98-109, February, 1986.

[17] Richard M. Karp and Michael O. Rabin.
Efficient Randomized Pattern-Matching Algorithms.
IBM Journal of Research and Development 31(2):249-260, March, 1987.

[18] R. Lipton.
Personal communication.

[19] Michael Luby and Charles Rackoff.
Pseudo-random Permutation Generators and Cryptographic Composition.
In *Proceedings of the 18th ACM Symp. on Theory of Computing*, pages 356-363. May, 1986.

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

[21] Roger M. Needham, Michael D. Schroeder.
Using Encryption for Authentication in Large Networks of Computers.
Communications of the ACM 21(12):993-999, December, 1978.
Also Xerox Research Report, CSL-78-4, Xerox Research Center, Palo Alto, CA.

[22] Randy F. Pausch, Jeffrey L. Eppinger.
Generating Interfaces for Camelot Data Servers.
In Alfred Z. Spector, Kathryn R. Swedlow (editors), *The Guide to the Camelot Distributed Transaction Facility: Release 1*, pages 21-27. Carnegie Mellon, 1987.

[23] Michael Rabin.
Digitalized Signatures and Public-Key Functions as Intractable as Factorization.
Laboratory for Computer Science Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology, January, 1979.

[24] Michael O. Rabin.
Probabilistic Algorithms in Finite Fields.
SIAM Journal on Computing 9:273-280, 1980.

[25] Michael Rabin.
Fingerprinting by Random Polynomials.
Center for Research in Computing Technology, Aiken Laboratory TR-81-15, Harvard, May, 1981.

[26] Michael O. Rabin.
Efficient Dispersal of Information for Security and Fault Tolerance.
Technical Report TR-02-87, Aiken Laboratory, Harvard Univerisity, April, 1987.

[27] Richard F. Rashid.
Threads of a New System.
Unix Review 4(8):37-49, August, 1986.

[28] J. Reif and J. D. Tygar.
Efficient Parallel Pseudo-Random Number Generation.
In *Advances in Cryptology: CRYPTO-85,* pages 433-446. Springer-Verlag, August, 1985.
To appear in SIAM J. on Computing.

[29] 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.

[30] Robert D. Sansom.
Security in a Network Operating System.
In *Securicom 86 - 4th Worldwide Congress on Computer and Communications Security and Protection.* March, 1986.

[31] M. Satyanarayanan.
Integrating Security in a Large Distributed Environment.
Technical Report CMU-CS-87-179, Carnegie-Mellon University, November, 1987.

[32] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, Michael J. West.
The ITC Distributed File System: Principles and Design.
In *Proceedings of the Tenth Symposium on Operating System Principles,* pages 35-50. ACM, December, 1985.
Also available as Carnegie-Mellon Report CMU-ITC-039, April 1985.

[33] Alfred Z. Spector.
Distributed Transaction Processing and the Camelot System.
In Yakup Paker et al. (editors), *Nato Advanced Study Institute Series - Computer and Systems Sciences: Distributed Operating Systems: Theory and Practice,* pages 331-353. Springer-Verlag, 1987.
Also available as Carnegie Mellon Report CMU-CS-87-100, January 1987.

[34] Alfred Z. Spector, Peter M. Schwarz.
Transactions: A Construct for Reliable Distributed Computing.
Operating Systems Review A 7(2):18-35, April, 1983.
Also available as Technical Report CMU-CS-82-143, Carnegie-Mellon University, January 1983.

[35] Alfred Z. Spector, Kathryn R. Swedlow, ed.
The Guide to the Camelot Distributed Transaction Facility: Release 1
0.7(33a) edition, Carnegie Mellon University, Pittsburgh, PA, 1987.

[36] Alfred Z. Spector, Dean Thompson, Randy F. Pausch, Jeffrey L. Eppinger, Richard Draves, Dan Duchamp, Dean S. Daniels, Joshua J. Bloch.
Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report.
Carnegie-Melton Report CMU-CS-87-129, Carnegie-Mellon University, June, 1987.

[37] J. D. Tygar and Bennet S. Yee.
Efficient Zero Knowledge Key Exchange.
To appear.

[38] R.W. Watson.
Distributed system architecture model.
In B.W. Lampson (editors), *Lecture Notes in Computer Science.* Volume 105: *Distributed Systems - Architecture and Implementation: An Advanced Course,* chapter 2,, pages 10-43. Springer-Verlag, 1981.