In Camelot and Avalon: A Distributed Transaction Facility, eds. J. Eppinger, L. Mummert, and A. Spector. Morgan-Kaufmann, 1991, pp. 381-400

Chapter 24

Strongbox

J. D. Tygar
Bennet S. Yee

24.1 Introduction

This chapter discusses the interfaces to *Strongbox*, a library of routines providing security for Camelot. The goals of Strongbox include enabling programs to run in a secure environment while making only minimal assumptions about the security of the operating system kernel and other system components. These programs are called *self-securing*. We have made the use of Strongbox relatively transparent to programmers who write Camelot 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 quickly implement a variety of policy decisions about data protection. 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 network resources (the *covert channel* problem), or the availability of system components (the *denial of service* problem). Future versions of our work will address these concerns. However, Strongbox can be used in conjunction with any solution to these secondary concerns.

At the core of Strongbox are new routines for authentication and fingerprinting. End-to-end private key encryption protects the privacy and integrity of messages passed between clients, servers, and other system components. The authentication algorithm provides us with support for a capability based protection system. This authentication system differs from previous authentication and key exchange protocols such as Needham-Shroeder [84] in that it can be proved to not leak any information that would allow eavesdroppers to masquerade as either party.

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. Further details about these algorithms and the assumptions we use can be found in Section 24.7

In Section 24.2, we discuss the design goals of Strongbox – the functional goals for the system, the performance goals, and the design constraints. In Section 24.3, we discuss the overall architecture of Strongbox, giving an overview of how Strongbox works. In Section 24.4, we present cookbook-style instructions on how to convert existing, non-secure Camelot clients and servers to use Strongbox, building on the previous `jack` and `jill` examples given in Chapter 5. In Section 24.5, we discuss the design of the secure loader and the white pages server, key components of Strongbox. In Section 24.6, we present the administrative interface provided by Strongbox to manipulate the access control list and to check data integrity. In Section 24.7 we discuss two basic algorithms used to create self-securing programs. In Section 24.8.3, we give performance figures and code size for these algorithms, and discuss issues of booting Strongbox.

24.2 Design Goals

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 $B, C, D, \ldots$ 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 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 to make a distributed system work efficiently without considering security issues. Strongbox addresses the issues by supporting self-securing programs which use only minimal assumptions about the security of the underlying kernel.

24.2.1 Functional Goals

The primary functional goals of Strongbox are to guarantee the integrity and privacy of data handled by it. Section 24.3 shows that the architecture of Strongbox protects data from modification or guarantees that data messages will be protected by end-to-end encryption. In Section 24.7 we show that Strongbox's fingerprinting and authentication algorithms do not leak information.

An additional functional goal of Strongbox is to provide Camelot 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 to Camelot so that converting an existing client server to be secure requires only a few simple modifications to the program text. In fact, it is so easy to modify clients and servers that it is possible to write an AWK[1] program which takes as input either an insecure server and a list of server subsystems or an insecure client and computes, as output, a secure server or client.

24.2.2 Performance Goals

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

[1]AWK is the UNIX utility for pattern scanning and processing. [4]

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. We have strived to make our security routines extremely fast, and we give our performance figures in Section 24.8.3.

Another measure of 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.

24.2.3 Constraints

It is necessary to make assumptions when building secure facilities. For example, if the system design calls for use of cryptography one must make the assumption that the cryptosystem used can not be deciphered by unauthorized agents. Since cryptosystems can always be broken by non-deterministic agents (which 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 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 Strongbox, we made several assumptions. First, we made a complexity assumption that some problem, such as factoring large integers or inverting DES, is intractable. We assumed that our base operating system, Mach, supports protected address spaces, including virtual address spaces stored on a disk. We assumed that the Disk Manager and the secure loader do 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.[2] 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 these issues, see [48, 92].) 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.

24.3 Strongbox Architecture

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 24.1 shows the components of the Strongbox and their relationships to each other, and to Mach and Camelot.

[2]The current release of Camelot does not in fact satisfy this, since log values are written in unencrypted log files.

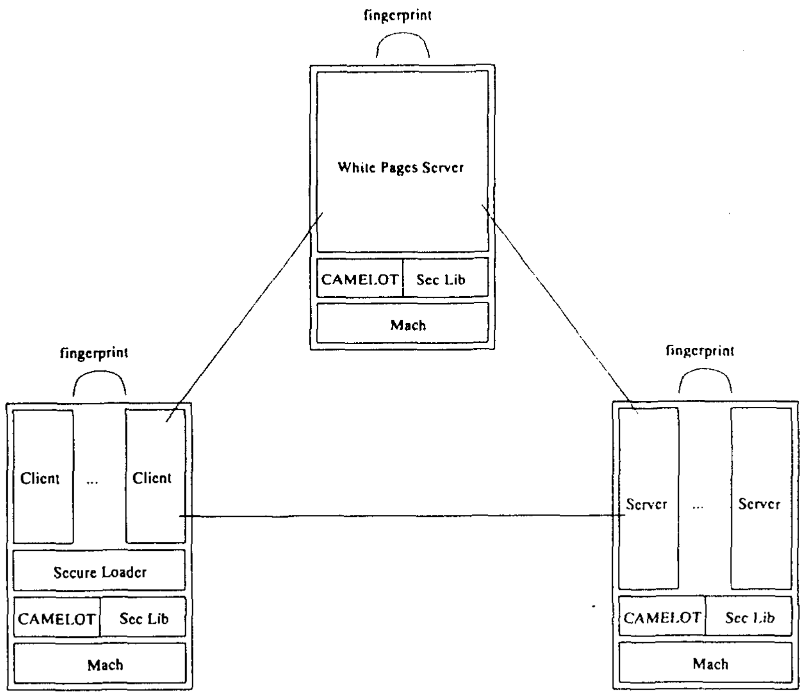

Figure 24.1: Strongbox Architecture
In this figure, we show how Strongbox interacts with Camelot and Mach. Each of the large boxes denote a computer. The client, server, and white pages server is shown as running on different machines; this is not a requirement of Strongbox, however, so they may all 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 basic Camelot servers which uses the Mach services, 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 operation which verifies that none of the files have been corrupted.

To run a secure task, we access the secure loader. The secure loader first copies the load image of the task into memory, and then verifies that the load image has not been corrupted while it was stored in the file system. The check must be done after the loader reads the image into memory because we do not trust the file system: if we just check the copy of the image in the file, we would be vulnerable to an adversary modifying the image during the timing window between when the loader checks the image and when the loader actually loads it. To do the checking, the secure loader calculates a cryptographic checksum, called a *fingerprint*. After the fingerprint has been verified to be identical to the one computed when the image was last written to the file system, the secure loader initializes the task's address space and starts the task running. We assume that the software development tools are secure against tampering, and that they can be trusted to write out the load image correctly; the software development may be done on a secure, isolated machine or on a machine with tools that have been converted to using Strongbox's fingerprinting. The initialization of the secure task is performed by using Mach's virtual memory primitives to set up the address space and thread control primitives to set up the initial contents of CPU registers.

After the secure task starts running, the secure loader sends it the solution to a randomly generated authentication puzzle which will be used later for authenticating the task to others, and checks in the puzzle (but not the solution) with the white pages server. The secure task has handshaking code as part of its startup sequence which receives the authentication puzzle solution as well as a capability token for accessing the white pages server. Since we know the load image is correct, this startup sequence could not have been tampered with. This communication is completely local to a machine and thus will not appear in any network communication; furthermore, note that the puzzle solution never appears in the file system. (Section 24.7 describes the fingerprinting and authentication algorithms.)

The authentication puzzle/solution pairs are inexpensive to generate; finding a solution when given just the puzzle, however, is extremely difficult (equivalent to factoring a composite number that is the product of two large primes). Checking the validity of a solution is also easy. This work property is analogous to that of password schemes where guessing a randomly generated password is extremely difficult. Unlike conventional password schemes, however, it has the additional benefit that the validity of the puzzle solution can be verified without revealing any information on it (the *zero knowledge* property – see Section 24.7).

For Camelot applications, the secure loader is a separate program that runs directly on Mach. For servers, the secure loader is part of the 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. (See Section 24.8.1 for the discussion on checking the integrity of the secure loader and Disk Manager at boot-time.)

Our protection system is based on capability tokens (capabilities). The capabilities are used as identification tokens which specify the list of operations the holder is allowed to invoke. Most of the time, capabilities are simply used to identify a user in a way that is similar to Andrew tokens [99] – the operations listed under the capability is just those that the user is permitted. In addition, however, they can also be used for *group* identity/membership or for implementing the UNIX real/effective-uid style of access control where applications can have the access rights of multiple parties [20].

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, check in the puzzles with the white pages server, and the white pages server registers itself with the Camelot name 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., for which servers will find a puzzle when the servers look in the white pages), the client must be started up by the secure loader since the white pages server 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 has not authenticated itself. It uses a secure remote procedure call (RPC) handling macro, which attempts the operation. The operation fails, returning CAP_MECH_PERMISSION_DENIED, at which point the macro looks up the server's authentication puzzle with the white pages server 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 server 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 memory managers may be used to protect the integrity of memory by fingerprinting pages as they go onto disk and verifying the fingerprint as they come off. To protect against leaking information as well as integrity, encryption is required.

24.3.1 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.

Each server has two databases that are maintained by the protection routines. The first, stored in recoverable storage, contains a mapping from user names to permissions. When a client authenticates successfully, the protection routines construct a new capability based on the permissions in this database. The protection routines return this new capability to the client and also enter it into the second database. The permissions in the first database are changed via special capability management routines the access to which is controlled in the same way as any other RPC. The second database, stored only in virtual memory, maps capabilities to permissions and is consulted each time an RPC request arrives to check that the client has permissions to invoke the requested operation. The authentication RPCs, of course, are exempt from the permissions check. The reason that there are two databases is to minimize the amount of memory leaked due to stale capabilities: since most capabilities never expire, we keep the capabilities in non-recoverable memory to prevent memory leak from capabilities to which references has been lost by all clients. Clients that survive a server crash must reauthenticate in order to regain their privileges.

It is easy to convert an existing non-secure server to be secure because only the initialization code and the argument declarations for the operation procedures are affected. Strongbox provides a separate, high level message demultiplexing procedure that performs the permission check. Decryption is performed here also. The appropriate key is inferred from the capability argument's index field, which is not encrypted. The demultiplexing procedure then hands the messages over to the individual subsystems' (lower level) 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' body, and reduce the work required to make a server secure.

24.4 Converting Camelot Clients and Servers to be Secure

This section is a cookbook guide to the conversion of Camelot clients and servers to use Strongbox. To illustrate the changes necessary, we will modify the Jack and Jill example given in 5. We first describe the changes to the clients, using Jack as an example, and then the changes to servers, using Jill as an example.

The required changes are minimal and a simple AWK program [4] could be used to automatically convert existing programs. Typically, however, attention should be given to the security policy that is desired and further changes may be necessary.

24.4.1 Changes to Clients

In order to make clients secure, several macros called inside Camelot clients must be replaced by their secure counterparts. These macros are defined in the Camelot include file `strongbox.h`. This file is included in secure servers' header files automatically generated by the Message Interface Generator (MIG). Here are instructions for converting clients to be secure:

1. Non-secure Camelot clients call `INITIALIZE_APPLICATION(`*name*`)` to register the client with the Transaction Manager. `SEC_INITIALIZE_APPLICATION(`*name*`)` replaces this macro in secure clients. This macro initializes the data structures used for authentication and for maintaining a secure channel once authenticated. If the client was not loaded by the secure loader, this macro aborts the execution of the client. For example, to make the Jack application secure, we would replace the code fragment in `jack.c`

```
if(!INITIALIZE_APPLICATION("Jack"))
{
        printf("Camelot is not running on this node!\n");
        exit(1);
    }
```

by

```
    if(!SEC_INITIALIZE_APPLICATION("Jack"))
    {
        printf("Both Camelot and the White Pages\n");
        printf("server must be running on this node!\n");
        exit(1);
    }
```

2. Non-secure clients call servers using SERVER_CALL, and SERVER_CALL_2. In secure clients the calls are replaced by SEC_SERVER_CALL, and SEC_SERVER_CALL_2. The secure versions of the server call macros perform necessary authentication steps. The first time a client calls a server with one of the secure macros, it looks up in the white pages the entry defining the server's authentication puzzle. (Note that the connection from the client to the white pages is guaranteed to be secure since it is an authenticated connection. The initial puzzle associated with the white pages is loaded into the client by the secure loader.) The macros ARGS and NOARGS pass certain parameters from the client to the server. In secure clients, these macros are replaced by SEC_ARGS and SEC_NOARGS respectively, which also pass secure parameter information such as the capabilities possessed by the client. For example, to make Jack secure, we would replace the code fragments in `jack.c`

```
    SERVER_CALL("Jill", jill_read(ARGS index, &value));
        ...
    SERVER_CALL("Jill", jill_write(ARGS index, value));
```

by

```
    SEC_SERVER_CALL("Jill", jill_read(SEC_ARGS index, &value));
        ...
    SEC_SERVER_CALL("Jill", jill_write(SEC_ARGS index, value));
```

24.4.2 Changes to Servers

Converting Camelot servers to be secure is just as simple as the client conversion. Once again we replace server macros with their secure counterparts. We also add some new macros and specify information to specify protected entry points. Here are the modifications necessary to make servers secure:

1. To include appropriate Strongbox macro definitions, in the `.defs` file insert after `#include <camelot.defs>` the line `#include <strongbox.defs>`. The modified RPC definition file should be compiled by MIG with the `-s` flag, i.e., `mig -s jill.defs`.

2. Make the macro call SEC_DECLARATIONS the first entry of the recoverable storage declarations. This allocates storage for Strongbox internal variables.[3] For example, to make Jill secure, we would replace the code fragment in `jill_globals.h`

   ```
   BEGIN_RECOVERABLE_DECLARATIONS
       struct jill_array_struct
       {
           int data[ARRAY_SIZE];
       } jill_array;
   END_RECOVERABLE_DECLARATIONS
   ```

   by

   ```
   BEGIN_RECOVERABLE_DECLARATIONS
       SEC_DECLARATIONS;
       struct jill_array_struct
       {
           int data[ARRAY_SIZE];
       } jill_array;
   END_RECOVERABLE_DECLARATIONS
   ```

3. The security system must know the location of all protected entry points in a server. The macro `SEC_SYMTAB(subsys)` generates the appropriate information to manage the entry points defined by the MIG subsystem *subsys*. The `SEC_SYMTAB(subsys)` declarations should all appear consecutively immediately before SEC_INITIALIZE_SERVER, which replaces INITIALIZE_SERVER. The SEC_INITIALIZE_SERVER macro terminates the list of entry points and calls the server initialization code. For example, the Jill server has only one system: `"jill"`, and to make Jill secure, we would replace the code fragment in `jill.c`

   ```
   INITIALIZE_SERVER(jill_init, FALSE, "Jill");
   ```

   by

   ```
   SEC_SYMTAB(jill);
   SEC_INITIALIZE_SERVER(jill_init, FALSE, "sJill");
   ```

4. The first initialization transaction executed by a server is a special initialization procedure. This procedure is the first argument to INITIALIZE_SERVER. To initialize access control specific Strongbox data, the macro SEC_INIT must be called as part of this initialization procedure. For example, to make Jill secure, we would replace the code fragment in `jill.c`

   ```
   int jill_init()
   BEGIN("jill_init")
       struct jill_array_struct temp;
       int i;
   ```

[3] SEC_DECLARATIONS also generates some padding between server data and Strongbox data. When programmers modify servers so that new recoverable storage is used, this padding may be adjusted to retain recoverability of previously stored objects. If the size of the recoverable storage is changed, use `SEC_DECLARATIONS_PAD(pad)` where *pad* is the size of the desired padding in words. *pad* typically is `SEC_DECLARATIONS_DEFAULT_PADDING - delta`, where `delta` is the change in the size of the recoverable storage. The default value of SEC_DECLARATIONS_DEFAULT_PADDING is 1024 bytes, so this can only accommodate relatively small size changes.

```
    /* Load up a temporary copy of the array with -1's */
    for (i=0; i < ARRAY_SIZE; i++)
        temp.data[i] = -1;

    MODIFY(REC(jill_array), temp);
END
```

by

```
int jill_init()
BEGIN("jill_init")
    struct jill_array_struct temp;
    int i;

    /* Load up a temporary copy of the array with -1's */
    for (i=0; i < ARRAY_SIZE; i++)
        temp.data[i] = -1;

    MODIFY(REC(jill_array), temp);
    SEC_INIT;
END
```

5. In Camelot, the macro START_SERVER causes the server to go into a state in which it may accept requests and process transactions. In the secure version of servers, this macro is replaced by SEC_START_SERVER, which starts the server so that it can only accept secure server calls. If START_SERVER_2 is used, it is replaced with SEC_START_SERVER_2. For example, to make Jill secure, we would replace the code fragment in `jill.c`

```
START_SERVER(jill_server, 10, 0);
```

by

```
SEC_START_SERVER(jill_server, 10, 0);
```

6. In secure servers operation procedures are passed the capability used by the client to access the entry point of the server. This capability can be used for tracing, auditing, and more elaborate protection schemes. It is passed as the first argument, of type `cap_t`, to every operation procedure. For example, to make Jill secure, we would replace the code fragments in `jill.c`

```
void op_jill_read(index, valPtr)
int index, *valPtr;
{
        ...
}

void op_jill_write(index, value)
int index, value;
{
        ...
}
```

by

```
void op_jill_read(cap, index, valPtr)
cap_t       cap;
int index, *valPtr;
{
        ...
}

void op_jill_write(cap, index, value)
cap_t       cap;
int index, value;
{
        ...
}
```

24.5 Secure Loader and White Pages Server

24.5.1 Secure Loader

The secure loader is a trusted server that invokes secure Camelot clients for the user. It loads the executable image and starts the client after checking that the client has not been corrupted. Then, it obtains a random puzzle solution from the white pages for the client when it registers the client with the white pages. It also initializes the client with the puzzle associated with the white pages so the client can make a secure server call to the white pages. At the same time, it also passes a secure random seed to the client's pseudo-random number generator. Finally, it gives a default capability to the client for using the white pages server to perform puzzle lookups. The puzzle solution, the random seed, and the initial capability is sent to the client via a RPC using the *bootstrap* port. Since both the secure loader and the newly started secure client are resident on the same machine, the content of these messages will never be visible on any networks. The Disk Manager plays a corresponding role for servers.

24.5.2 White Pages

As we will discuss in Section 24.7, the security of Strongbox depends on a constant which is the product of two large primes. Successfully breaking Strongbox's authentication routine is algorithmically equivalent to factoring the constant. We have provided a default composite number in the Camelot release. If this value should be changed for a particular installation, a new number can be generated by running the `NewSecrets` program, also included with the Camelot release. `NewSecrets` generates new header files containing a new composite number and secret factors with which Strongbox is built. Of course, binaries compiled with one set of constants are incompatible with those compiled with another set of numbers.

24.6 Interfaces

The key idea here is that every RPC message is prepended with a capability which is checked prior to invoking the operation at the server. The messages are also encrypted, so that somebody listening on the network will not have access to the data (excluding port names).[4] The macros mentioned in Section 24.4 performs the prepending, and the Strongbox demultiplexing procedure transparently performs the access check. The first argument of every operation routine in a secure server is the capability used by the client; the routines may use this argument to perform finer grain access checks.

24.6.1 Client/Server Interfaces

In addition to the normal service remote operations that a server provides, each secure server has other remote procedures that are defined by Strongbox. Strongbox's 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 Strongbox's remote operations are treated in the same manner as the service remote operations, an administrator may give administrative privileges to a trusted user as well.

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

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

`authCap` the invoking user's capability (administrator).
`userName` the symbolic name of the user who is being added.

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

`authCap` the invoking user's capability (administrator).
`userName` the symbolic name of the user who is being permitted to invoke the operation.
`operation` the name of the operation which the user `userName` is now allowed to invoke.

[4]The prepended capability is in two parts, an index and a secret random number. The index is not encrypted so that the receiver can determine which key to use to decrypt the rest of the message; the random number is encrypted along with the rest of the data, so the capability is not leaked.

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

`authCap` the invoking user's capability (administrator).
`userName` the symbolic name of the user who is being unpermitted to invoke the operation.
`operation` the name of the operation which the user `userName` is now disallowed from invoking.

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 an administrator may give permission to invoke `CapMech_Unpermit`, for example, to security personnel.

The administrator login is `local1` by default. This should be changed as appropriate for the installation.

24.6.2 White Pages Interface

```
void op_Wp_Fp_Check(cap, fn, data, size, ok)
cap_t           cap;
sec_filename_t  fn;
char            *data;   /* OOL */
vm_offset_t     size;
int             *ok;
```

`cap` the capability of the invoker.
`fn` the name of the file whose data is being checked.
`data` the out-of-line transmitted data which is the contents of the file.
`size` the size of data transmitted.
`ok` the return area for whether the data fingerprints correctly.

This routine allow users to check the fingerprints of data. The argument *data* contains the contents of the file `fn`. The white pages returns in `*ok` a nonzero if the data's fingerprint matches the stored fingerprint.

24.7 Security Algorithms

24.7.1 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 [95]. 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 $L(m.n) = L(E(m).n)$ [68]. Indeed, the problem is much worse than this. Alexi, Chor, Goldreich, and Schnorr recently proved that if an adversary can, 50% + ϵ of the time, find the low order bit of m given $E(m)$, the adversary can invert arbitrary RSA encryptions [5, 19]. A corollary to this result is that the usual query-response methods for encryption, such as the family of protocols described in [79], 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 [84]. 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 [76]. For example, the first author has found a method to subvert the authentication scheme used by the Andrew File System VICE [100], 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 [37, 32]. 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 information about the vertices in Q.

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

2. A prepares $\frac{n(n-1)}{2}$ 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 [57], 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 [34]. 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.

24.7.2 Our protocol

In research described in [118], we have developed a family of zero-knowledge protocols which are efficient for real use in applications. Our timing figures are given in Section 24.8.3. 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 [89]: *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 $\mathrm{GCD}(r, n) \neq 1$ (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 $s \not\equiv \pm r \bmod n$. If $s = \pm r \bmod n$, then we can pick a new r and repeat the algorithm. If $s \not\equiv \pm r \bmod n$, then it is the case that $\mathrm{GCD}(r + s.n) = p$ 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 $< v_1^2 \bmod n. v_2^2 \bmod n, \ldots, v_t^2 \bmod n >$.
2. B flips t independent coins and send back a vector of t random bits $< b_1 . \ldots . b_t >$ to A.
3. For $1 \leq i \leq t$, A computes

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

   A transmits the vector $< z_1 . \ldots . z_t >$.
4. B verifies that for $1 \leq i \leq t$, that

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

   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 $rv_i \bmod n$, since $r = rv_i(v_i)^{-1} \bmod n$. Finally, all that B sees is a series of random values of the form $< z_i . z_i^2 \bmod n >$. 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.

24.7.3 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. See [118] for details of our key-exchange algorithm.

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.

24.7.4 Fingerprinting

Karp and Rabin introduced an algorithm which computes a *cryptographic checksum* [58]. 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 [17]. 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 $< F, \phi_k(F) >$. 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 [91, 90]) and we have implemented two different efficient routines for doing so. Compute $r(x) = f_s(x) \bmod g(x)$. Then $r(x)$ 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 [33]. 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 24.8.3).

24.8 Special Issues

24.8.1 Booting

Special problems arise when a Camelot system is booted. The boot program must pick up the fingerprint key so that it can validate the authenticity of the Mach kernel, the Disk Manager, the white pages server, and the secure loader; similarly, the white pages server must pick up a random seed for a pseudo-random number generator which it will use for authentication and for generating seeds for secure servers that the secure loader brings up.

In the current version of Strongbox, we omit checking the fingerprints of the core programs and the initial pseudo-random seed is obtained from system statistics such as the last modification time of the `/tmp` directory, the control terminal, the current time, etc. Note that these system statistics are *not* good sources of random numbers: they are used merely for expediency. In future versions, we will address these problems by having the fingerprint values and initial random seed be entered by an operator. There are several ways to insure that the white pages server, secure loader, Disk Manager, and kernel are correct; one of the most straightforward is to make sure that the boot ROM associated with the machine calls a routine which calculates the fingerprint of the secure loader and the operating system kernel before booting those programs. If the fingerprinted values match the correct values, we can be sure that all is well since forging a fingerprint will succeed only 2^{-32} of the time. But how can we insure that the operator entering these values can be trusted? Several solutions to the "trusted operator" problem are possible including physical security measures and password based authentication of operators. Strongbox validates users by requiring them to enter passwords when they initially bringing up the system. Once the secure loader and Disk Manager are running, the system will continue running securely since the secure loader will verify the fingerprint of every server it brings up and will pass it a random seed it can use for its own pseudo-random number generation.

The pseudo-random numbers are used for authentication, key exchange, and puzzle generation. All puzzles other than that of the white pages server are generated as needed. The white pages server's puzzle solution as well as the key exchange secret may be changed for any particular instance of Strongbox, but once fixed they can not be changed. These secrets are present in C header files that are not exported to the user. These files should not be placed on an untrustworthy machine. Similarly, the values are present in the `wp` binary, so that file should also be secret. Currently, we simply assume that this binary is kept on removable media and that the operator removes it after booting Camelot.

24.8.2 Starting Strongbox

This section describes how to start up the current version of Strongbox after Camelot is booted. The start up procedure does not yet address the issues described above; future versions of Strongbox will have a simpler and more automated start up procedure that will be run when the kernel is booted.

After successfully bringing up Camelot, there are some additional steps needed before secure servers and clients can be run. First, the white pages server and the secure loader must be brought up by using the NCA program. Add the two servers using the `as` command, and start them using the `ss` command. The white pages server must be completely up prior to starting the secure loader.

When the white pages server and the secure loader are up, secure servers can be added to the

Node Server's database and started. Secure clients are run via the `sec_run` interface. The two system clients `wp_admin` and `sec_admin` are automatically fingerprinted when the white pages server starts up; other clients must be first fingerprinted. To do this, run the `wp_admin` program. Use the `f (fingerprint)` command to fingerprint the secure client, commit the transaction, and now `sec_run` can be used to execute the client. Note that if the secure server was just initialized, nobody except the administrator will have any access permissions, so `sec_admin` must be run to give users permissions. Start up `sec_admin` via `sec_run` as was done with `wp_admin`, and give the name of the secure server at the `"Server"` prompt (this is the name in the `SEC_INITIALIZE_SERVER` macro). The `p (permit)` command will prompt for the name of the RPC and the user name to whom permission should be given.

24.8.3 Performance

This section gives timing figures for Strongbox's implementations of authentication and fingerprinting. 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 [108]. 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 880 KByte/sec. The fingerprinting routine uses a 65536 (2^{16}) entry table of pre-computed partial residues to achieve this speed.[5] 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 $x^{32}a(x) \bmod g(x)$ 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.

Currently, the secure loader fingerprints the entire image of the client program before allowing execution. This has some performance impacts: instead of demand paging from the file and performing page-out to a secure paging device, reading the entire program causes excessive paging/disk activity that might be avoidable (some pages of the program may never be needed). This would mean that any external memory managers must handle page-in requests of program text by checking a per-page fingerprint against a known value given by the white pages server's database.

[5] We also have a less memory intensive version which uses a 256 entry table fingerprints at a rate 710 KByte/sec.

24.9 Conclusions

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 these algorithms. We will continue to improve Strongbox and plan to address solutions to the denial of service problem in future work.

Bibliography

[1] G. M. Adelson-Velskiĭ and E. M. Landis. An Algorithm for the Organization of Information. *Soviet Mathmatics*, 3(5):1259–1262, September 1962. Translated from Russian *Doklady, Akademiĭ Nauk SSSR* 146(2):263-266.

[2] Rakesh Agrawal. A Parallel Logging Algorithm for Multiprocessor Database Machines. In *Proceedings of the Fourth International Workshop on Database Machines*, pages 256–276, March 1985.

[3] Rakesh Agrawal and David J. DeWitt. Recovery Architectures for Multiprocessor Database Machines. In *Proceedings of ACM-SIGMOD 1985 International Conference on Management of Data*, pages 132–145, May 1985.

[4] Alfred Aho, Brian Kernighan, and Peter Weinberger. *The AWK Language*. Addison-Wesley, 1987.

[5] W. Alexi, B. Chor, O. Goldreich, and C. P. Schnorr. RSA and Rabin Functions: Certain Parts are as Hard as the Whole. *SIAM Journal on Computing*, 17(2), April 1988.

[6] Anonymous, et al. A Measure of Transaction Processing Power. *Datamation*, 31(7), April 1985. Also available as Technical Report TR 85.2, Tandem Corporation, Cupertino, California, January 1985.

[7] Emmanuel A. Arnould, François J. Bitz, Eric C. Cooper, H. T. Kung, Robert D. Sansom, and Peter A. Steenkiste. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 205–216, April 1989. Also available as Technical Report CMU-CS-89-101, Carnegie Mellon University, January 1989.

[8] Robert V. Baron, David L. Black, William Bolosky, Jonathan Chew, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. Mach Kernel Interface Manual. Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1987.

[9] P. Bernstein and N. Goodman. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *ACM Transactions on Database Systems*, 9(4):596–615, December 1984.

[10] David L. Black, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. The Mach Exception Handling Facility. Technical Report CMU-CS-88-129, Carnegie Mellon University, April 1988.

[11] Joshua J. Bloch. *A Practical Approach to Replication of Abstract Data Objects*. PhD thesis, Carnegie Mellon University, May 1990. Also available as Technical Report CMU-CS-90-133, Carnegie Mellon University, May 1990.

[12] Joshua J. Bloch, Dean S. Daniels, and Alfred Z. Spector. A Weighted Voting Algorithm for Replicated Directories. *JACM*, 34(4), October 1987. Also available as Technical Report CMU-CS-86-132, Carnegie Mellon University, July 1986.

[13] Andrea J. Borr. Transaction Monitoring in Encompass (TM): Reliable Distributed Transaction Processing. In *Proceedings of the Very Large Database Conference*, pages 155–165, September 1981.

[14] Andrea J. Borr. Robustness to Croach in a Distributed Database: A Non Shared-Memory Multi-Processor Approach. In *Proceedings of the Very Large Database Conference*, pages 445–453, August 1984.

[15] Richard S. Brice and Stephen W. Sherman. An Extension of the Performance of a Database Manager in a Virtual Memory System using Partially Locked Virtual Buffers. *ACM Transactions on Database Systems*, 2(2):196–207, June 1976.

[16] Mark R. Brown, Karen N. Kolling, and Edward A. Taft. The Alpine File System. *ACM Trans. on Computer Systems*, 3(4):261–293, November 1985.

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

[18] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

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

[20] Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720. *UNIX Programmer's Reference Manual (PRM)*, April 1986.

[21] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988.

[22] Dean S. Daniels. *Distributed Logging for Transaction Processing*. PhD thesis, Carnegie Mellon University, December 1988. Also available as Technical Report CMU-CS-89-114, Carnegie Mellon University, August 1988.

[23] Dean S. Daniels and Alfred Z. Spector. An Algorithm for Replicated Directories. In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 104–113. ACM, August 1983. Also available in *Operating Systems Review*, 20(1):24-43, January 1986.

[24] Dean S. Daniels, Alfred Z. Spector, and Dean Thompson. Distributed Logging for Transaction Processing. In *Sigmod '87 Proceedings*. ACM, May 1987. Also available as Technical Report CMU-CS-86-106, Carnegie Mellon University, June 1986.

[25] David Detlefs, Maurice P. Herlihy, and Jeannette M. Wing. Inheritance of Synchronization/Recovery Properties in Avalon/C++. *Computer*, 21(12), December 1988.

[26] Dan Duchamp. *Transaction Management*. PhD thesis, Carnegie Mellon University, June 1989. Also available as Technical Report CMU-CS-88-192, Carnegie Mellon University, June 1989.

[27] Wolfgang Effelsberg and Theo Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[28] Jeffrey L. Eppinger. *Virtual Memory Management for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, February 1989. Also available as Technical Report CMU-CS-89-115, Carnegie Mellon University, February 1989.

[29] Jeffrey L. Eppinger and Alfred Z. Spector. Virtual Memory Management for Recoverable Objects in the TABS Prototype. Technical Report CMU-CS-85-163, Carnegie Mellon University, December 1985.

[30] K. P. Eswaran, James N. Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[31] D. DeWitt et. al. Implementation Techniques for Main Memory Database Systems. In *Proceedings of ACM SIGMOD '84*, pages 1–8, May 1984.

[32] Uriel Feige, Amos Fiat, and Adi Shamir. Zero Knowledge Proofs of Identity. In *Proceedings of the Nineteenth ACM Symposium on Theory of Computing*, pages 210–217, May 1987.

[33] 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.

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

[35] D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Engineering*, June 1985.

[36] David K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 150–162. ACM, December 1979.

[37] 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*, pages 291–304, May 1985.

[38] James N. Gray. Notes on Database Operating Systems. In R. Bayer and R. M. Graham and G. Seegmüller, editor, *Operating Systems - An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978. Also available as Technical Report RJ2188, IBM Research Laboratory, San Jose, California, 1978.

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

[40] Mach Networking Group. Network Server Design. Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1988.

[41] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in Five Easy Pieces. Technical Report 5, DEC Systems Research Center, July 1985.

[42] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[43] Andrew B. Hastings. Distributed Lock Management in a Transaction Processing Environment. Technical Report CMU-CS-89-152, Carnegie Mellon University, May 1989.

[44] Pat Helland. Transaction Monitoring Facility. *Database Engineering*, 8(2):9–18, June 1985.

[45] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. Group Commit Timers and High Volume Transaction Systems. Presented at the Second International Workshop on High Performance Transaction Systems, Asilomar, September 1987.

[46] Maurice P. Herlihy. General Quorum Consensus: A Replication Method for Abstract Data Types. Technical Report CMU-CS-84-164, Carnegie Mellon University, December 1984.

[47] Maurice P. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems*, 4(1), February 1986.

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

[49] Maurice P. Herlihy and Jeannette M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*. IEEE, July 1987.

[50] Maurice P. Herlihy and Jeannette M. Wing. Reasoning About Atomic Objects. In *Proceedings of the Symposium on Real-Time and Fault-Tolerant Systems*, Warwick, England, September 1988. Also available as Technical Report CMU-CS-87-176, Carnegie Mellon University.

[51] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[52] IBM Corporation. *Customer Information Control System/Virtual Storage, Introduction to Program Logic*, sc33-0067-1 edition, June 1978.

[53] IBM Corporation, World Trade Systems Center, San Jose, CA. *IMS Version 1 Release 1.5 Fast Path Feature Description and Design Guide*, 1979.

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

[55] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 225–235. ACM, 1985.

[56] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 225–235. ACM, January 1985.

[57] R. M. Karp. Reducibility among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[58] Richard M. Karp and Michael O. Rabin. Efficient Randomized Pattern-Matching Algorithms. Technical Report TR-31-81, Aiken Laboratory, Harvard University, December 1981.

[59] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[60] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[61] Butler W. Lampson. Atomic Transactions. In G. Goos and J. Hartmanis, editors, *Distributed Systems - Architecture and Implementation: An Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, chapter 11, pages 246–265. Springer-Verlag, 1981.

[62] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[63] Richard Allen Lerner. Reliable Servers: Design and Implementation in Avalon/C++. In *Proceedings International Symposium on Databases in Parallel and Distributed Systems*, pages 13–21, Austin, TX, December 1988. IEEE. Also available as Technical Report CMU-CS-88-177, Carnegie Mellon University, September 1988.

[64] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239. ACM, August 1986.

[65] Bruce Lindsay, John McPherson, and Hamid Pirahesh. A Data Management Extension Architecture. In *Proceedings of ACM SIGMOD '87*, pages 220–226. ACM, May 1987.

[66] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and Communication in R*: A Distributed Database Manager. *ACM Transactions on Computer Systems*, 2(1):24–38, February 1984.

[67] Bruce G. Lindsay, et al. Notes on Distributed Databases. Technical Report RJ2571, IBM Research Laboratory, San Jose, California, July 1979. Also appears in Droffen and Poole (editors), *Distributed Databases*, Cambridge University Press, 1980.

[68] R. Lipton. Personal communication.

[69] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 111–122. ACM, November 1987.

[70] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl. Argus Reference Manual. Technical Report TR-400, MIT Laboratory for Computer Science, Cambridge, MA, November 1987.

[71] B. Liskov, A. Snyder, R.Atkinson, and C. Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8), August 1977.

[72] Barbara H. Liskov. Overview of the Argus Language and System. Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, February 1984.

[73] Barbara H. Liskov. Progress Report of the Programming Methodology Group. In *MIT LCS Progress Report*, volume 21, pages 142–176. MIT Press, June 1984.

[74] Barbara H. Liskov, Robert Scheifler, Edward Walker, and William Weihl. Orphan Detection. Programming Methodology Group Memo 53, Laboratory for Computer Science, Massachusetts Institute for Technology, February 1987.

[75] Barbara H. Liskov and Robert W. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

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

[77] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[78] David McDonald, Scott Fahlman, and Alfred Spector. An Efficient Common Lisp for the IBM RT PC. In *IBM Academic Information Systems University AEP Conference*, pages 556–580, June 1987. Also available as Technical Report CMU-CS-87-134, Carnegie Mellon University, July 1987.

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

[80] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. Technical Report RJ6649, IBM Almaden Research Center, January 1989.

[81] C. Mohan and B. Lindsay. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions. In *Proceedings of the Second Annual Symposium on Principles of Distributed Computing*, pages 76–88. ACM, August 1983.

[82] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

[83] E. T. Mueller, J. D. Moore, and G. J. Popek. A Nested Transaction Mechanism for LOCUS. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 71–89, October 1983.

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

[85] R. J. Peterson and J. P. Strickland. LOG Write-Ahead Protocols and IMS/VS Logging. In *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 216–243. ACM, March 1983.

[86] Jonathan B. Postel. Internetwork Protocol Approaches. In Paul E. Green, Jr., editor, *Computer Network Architectures and Protocols*, chapter 18, pages 511–526. Plenum Press, 1982.

[87] C. Pu. Personal communication.

[88] C. Pu. *Replication and Nested Transactions in the Eden Distributed System*. PhD thesis, Univ. of Washington, 1986.

[89] Michael Rabin. Digitalized Signatures and Public-Key Functions as Intractable as Factorization. Technical Report TR-212, MIT Laboratory for Computer Science, January 1979.

[90] Michael Rabin. Fingerprinting by Random Polynomials. Technical Report TR-81-15, Aiken Laboratory, Harvard University, May 1981.

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

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

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

[94] David P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, MIT, September 1978.

[95] 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.

[96] Paul Rovner, Roy Levin, and John Wick. On Extending Modula-2 for Building Large, Integrated Systems. Technical Report 3, DEC Systems Research Center, January 1985.

[97] Giovanni Maria Sacco and Mario Schkolnick. Buffer Management in Relational Database Systems. *ACM Transactions on Database Systems*, 11(4):473–498, December 1986.

[98] Robert Daniell Sansom. *Building a Secure Distributed Computer System*. PhD thesis, Carnegie Mellon University, May 1988. Also available as Technical Report CMU-CS-88-141, Carnegie Mellon University, May 1988.

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

[100] M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and 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 Technical Report CMU-ITC-039, Carnegie Mellon University, April 1985.

[101] Peter M. Schwarz. *Transactions on Typed Objects*. PhD thesis, Carnegie Mellon University, December 1984. Also available as Technical Report CMU-CS-84-166, Carnegie Mellon University, December 1984.

[102] Peter M. Schwarz and Alfred Z. Spector. Synchronizing Shared Abstract Types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984. Also available in Stanley Zdonik and David Maier (editors), Readings in Object-Oriented Databases. Morgan Kaufmann, 1988. Also available as Technical Report CMU-CS-83-163, Carnegie Mellon University, November 1983.

[103] Stephen W. Sherman and Richard S. Brice. Performance of a Database Manager in a Virtual Memory System. *ACM Transactions on Database Systems*, 1(4):317–343, December 1976.

[104] M. D. Skeen. *Crash Recovery in a Distributed Database System*. PhD thesis, University of California, Berkeley, May 1982.

[105] Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, and Peter M. Schwarz. Support for Distributed Transactions in the TABS Prototype. *IEEE Transactions on Software Engineering*, SE-11(6):520–530, June 1985. Also available in Proceedings of the Fourth Symposium on Reliability in Distributed Software and Database Systems, Silver Springs, Maryland, IEEE, October, 1984 and as Technical Report CMU-CS-84-132, Carnegie Mellon University, July 1984.

[106] Alfred Z. Spector, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, and Randy Pausch. Distributed Transactions for Reliable Systems. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 127–146. ACM, December 1985. Also available in Concurrency Control and Reliability in Distributed Systems, Bharat K. Bhargava, ed., pp. 214-249, Van Nostrand Reinhold Company, New York, and as Technical Report CMU-CS-85-117, Carnegie Mellon University, September 1985.

[107] Alfred Z. Spector and Peter M. Schwarz. Transactions: A Construct for Reliable Distributed Computing. *Operating Systems Review*, 17(2):18–35, April 1983. Also available as Technical Report CMU-CS-82-143, Carnegie Mellon University, January 1983.

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

[109] Lindsey L. Spratt. The Transaction Resolution Journal: Extending the Before Journal. *Operating Systems Review*, 19(3):55–62, July 1985.

[110] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, 1984.

[111] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[112] Michael Stonebraker. Virtual Memory Transaction Management. *Operating Systems Review*, 18(2):8–16, April 1984.

[113] Michael Stonebraker and Larry Rowe. The Design of POSTGRES. In *Proceedings of ACM SIGMOD '86*, pages 189–222. ACM, May 1987.

[114] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.

[115] Tandem. *NonStop SQL Benchmark Workbook*, 84160 edition, March 1987.

[116] Dean Thompson. Coding Standards for Camelot. Camelot Working Memo 1, June 1986.

[117] Irving L. Traiger. Virtual Memory Management for Database Systems. *Operating Systems Review*, 16(4):26–48, October 1982. Also available as Technical Report RJ3489, IBM Research Laboratory, San Jose, California, May 1982.

[118] J. D. Tygar and Bennet S. Yee. Strongbox: A System for Self-Securing Programs. In Rachard F. Rashid, editor, *Carnegie Mellon Computer Science: A 25-Year Commemorative*. Addison-Wesley, Reading, Massachusetts, 1991.

[119] R. W. Watson. IPC Interface and End-To-End Protocols. In B. W. Lampson, editor, *Distributed Systems - Architecture and Implementation: An Advanced Course*, volume 105 of *Lecture Notes in Computer Science*, chapter 7, pages 140–174. Springer-Verlag, 1981.

[120] William E. Weihl. *Specification and Implementation of Atomic Data Types*. PhD thesis, MIT, March 1984.

[121] Matthew J. Weinstein, Thomas W. Page, Jr., Brian K. Livezey, and Gerald J. Popek. Transactions and Synchronization in a Distributed Operating System. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 115–126. ACM, December 1985.

[122] R. Williams, et al. R*: An Overview of the Architecture. Technical Report RJ3325, IBM Research Laboratory, San Jose, California, December 1981.

[123] J. M. Wing. Using Larch to Specify Avalon/C++ Objects. *IEEE Transactions on Software Engineering*, pages 1076–1088, September 1990.

[124] Michael Wayne Young, Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub, Jeffrey L. Eppinger, Jonathan Chew, William Bolosky, David L. Black, and Robert V. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 63–76. ACM, November 1987.