Secure Network Objects

Leendert van Doorn * — *Vrije Universiteit*, *Amsterdam, The Netherlands*

Martín Abadi, Mike Burrows, Edward Wobber — *Digital Systems Research Center*, *Palo Alto, California, USA*

Abstract

We describe the design and implementation of secure network objects, which provide security for object-oriented network communication. The design takes advantage of objects and subtyping to present a simple but expressive programming interface for security, supporting both access control lists and capabilities. The implementation of this design fits nicely within the structure of the existing network objects system; we discuss its internal components, its performance, and its use in some applications.

1 Introduction

Object-oriented communication has become popular in distributed systems [2, 23, 19]. With objects or without them, distributed systems typically rely on networks with no low-level support for security; the vulnerability of distributed systems is by now evident and worrisome [24, 4]. Therefore, a need exists for secure object-oriented communication.

We describe the design and implementation of *secure network objects*. Secure network objects extend Modula-3 network objects [18, 2] with security guarantees. When a client invokes a method of a secure network object over the network, the main security properties are:

- The client must possess an unforgeable object reference.
- The client and the owner of the object can choose to authenticate each other.
- The arguments and results of the method invocation are protected against tampering and replay, and optionally against eavesdropping.

*Most of this work was done at Digital's Systems Research Center. The author can be contacted at leendert@cs.vu.nl.

For high-speed bulk communication, the network objects system supports buffered streams called *readers* and *writers*. We make these streams secure also.

Our design accommodates both access control lists (ACLs) [11] and capabilities [6]. It seems natural to treat network object references as capabilities; moreover, these capabilities can be implemented efficiently. However, capabilities suffer from the well-known confinement problem: it is hard to keep them sufficiently secret [10]. The support for ACLs allows implementors to limit this problem, and to use identity-based security whenever that is appropriate, in particular for auditing. Systems with both ACLs and capabilities are not new; we include some comparisons in section 6.

The central goal of our work was the integration of security and network objects. We have obtained the following features:

- Applications can use security easily, with minimal code changes.
  - Security is mostly encapsulated within the network objects run-time system.
  - Objects and methods provide convenient units of protection.
  - Subtyping expresses security properties quite simply. Secure network objects are a subtype of regular network objects.
- Through the combination of ACLs and capabilities, the security model is rich enough to enable applications with sophisticated security requirements.
- The implementation of our design is reasonably straightforward. In this respect, we benefited from the structure of the existing network objects system. We also borrowed ideas from previous work on authentication [27]: each node runs an *authentication agent* that is responsible for managing keys and for identifying local users to other nodes. We feel that our experience partially validates those previous efforts.

The next section presents our programming interface. Sections 3 and 4 describe the two main components of our system: the authentication agent and the run-time system. Section 5 discusses experience with secure network objects, including performance measurements and some example applications. Section 6 discusses related work.

2 Programming Interface

In a world with fast CPUs, no government export controls, and pervasive use of cryptographic credentials, we would give uniform security guarantees for all objects, even for those that do not require them. This would make for a simpler system and a shorter paper.

As a compromise, we define network objects with three levels of security: (1) no security; (2) authenticity; (3) authenticity and secrecy. We call the last two kinds *secure network objects*. A *secure invocation* is the invocation of a method of a secure network object.

An important characteristic of our design is that it gives security guarantees for whole objects rather than for individual methods. This allows us to specify the properties of secure network objects through subtyping, rather than by inventing new language features. We have a type of objects with no security, a subtype with authenticity, and a further subtype that adds secrecy.

These types are explained in the following sections, along with a type to represent identities. We postpone the discussion of readers and writers to section 2.6.

2.1 Network Objects (Background)

A Modula-3 *object* is a reference (pointer) to a data record paired with a method suite. The method suite is a record of procedures that take the object as first parameter. A method invocation specifies a method name, an object, and additional arguments. It may yield results, consisting of a return value and values for any `VAR` parameters.

A *network object* is a reference meaningful throughout a network; its methods can be invoked from multiple address spaces, possibly on different nodes. A method invocation is *remote* if it crosses an address-space boundary. Method invocations have at-most-once semantics (at least in the absence of attackers). Each network object has one address space as *owner* (or *server*); the object always has the same owner. Other address spaces are *clients*. Typically, each host exports a well-known network object that acts as a local name server. To import an object, a client may contact the name server of the owner of the object, or it may receive the object as argument or result of a remote invocation.

Each client of a network object has a special object, the *surrogate*; passing a network object to a new client causes a corresponding surrogate to be created at the client. The client invokes the methods of the surrogate, which in turn invoke the methods of the object at the owner. The presence of a surrogate is invisible to the programmer.

An *object type* specifies a collection of methods. A new object type can be defined as a *subtype* of an existing object type; the new type may inherit some methods from the existing type, may override others, and may add some methods of its own. The type `NetObj.T` is the base type for the network object subtyping hierarchy: all types of network objects are subtypes of `NetObj.T`; the type `NetObj.T` does not specify any methods. In general, network objects of type `NetObj.T` are not secure.

2.2 Authenticity

We introduce a subtype `AuthNetObj.T` of `NetObj.T`. Network objects of type `AuthNetObj.T` are unforgeable references. When a client invokes a method of an object of type `AuthNetObj.T`, the following guarantees hold (even in the presence of attackers):

1. Integrity: The invocation that the server receives is exactly the one issued by the client. The results that the client receives are exactly the ones issued by the server as a response to this invocation.

2. At-most-once semantics: The server receives the invocation at most once. The client receives the response at most once.

3. Confinement: If the invocation or the response contains a secure network object, an eavesdropper does not learn enough to invoke a method of that object.

By (1), the server knows that the client has the method name, the object, and all additional arguments of the invocation; the client knows that the server has the results. In addition, since an object has a unique server, that same server responds to all invocations of methods of the object.

By (2), client and server are protected against replays.

By (3), a secure network object can be treated as a capability or protected name [17]. If a secure network object is passed only from the server to a single client in a secure invocation, the server knows that any later invocation using this object originates in this client.

More generally, if the object is passed only in secure invocations between trusted parties, the server knows that any later invocation using this object originates in a trusted party.

One could imagine extending (3) to insecure invocations; one would protect secure network objects even when they are passed in insecure invocations. However, there is not much gain in doing this. For instance, the result of an insecure invocation may be a secure network object; there is little point in protecting this result against eavesdroppers because any address space could invoke the method that gives this result.

2.3 Secrecy

For secret communication, we introduce a subtype `SecNetObj.T` of `AuthNetObj.T`. When a client invokes a method of an object of type `SecNetObj.T`, the following additional guarantee holds:

4. Secrecy: An eavesdropper does not obtain any part of the method name, the object, the additional arguments, or the results of the invocation.

However, we do not attempt to provide perfect anonymity. An eavesdropper may recognize that two method invocations are for the same object. As explained in the next section, an eavesdropper may also learn who is communicating.

2.4 Identity

An *identity* consists of a user name and a host name. For simplicity, we assume that each address space is running on behalf of one user on one host, and we associate with each address space the corresponding identity.

We introduce a type to represent identities:

```
INTERFACE Ident;

TYPE
  T <: OBJECT METHODS
         userName(): TEXT;
         hostName(): TEXT;
       END;

PROCEDURE Mine(): T;

END Ident.
```

This interface describes a type `Ident.T`, exposing the methods `userName` and `hostName`. (`Ident.T` is an object type, but not a network object type.) Each of `userName` and `hostName` takes no arguments and returns a string. An *identity object* is an object of type `Ident.T`. The interface also provides a procedure `Ident.Mine`; a call to `Ident.Mine` in an address space always returns the identity of the address space. Attempting to pass any identity other than `Ident.Mine()` in a remote invocation is a run-time error that raises an exception; this behavior is chosen for programmer convenience, but is not necessary for security.

The following guarantee is associated with the type `Ident.T`:

5. If an identity object is passed as argument or result of a secure remote invocation, the receiver is assured that calls on `userName` and `hostName` will return the identity of the sending address space.

In contrast, an identity object received in an insecure invocation may not identify its sender—the sender may have been dishonest.

Even if an invocation is secure, the anonymity of an address space that sends `Ident.Mine()` is not protected: an eavesdropper can obtain the corresponding user name and host name. In fact, an eavesdropper can deduce host names simply from the pattern of communication between address spaces, even when no identities are sent. On the other hand, if an interface does not require the exchange of identities, its users can remain anonymous; in principle, an address space discloses the name of its user only as a deliberate step.

Our treatment of identity suffices for our current applications and enables us to take advantage of existing mechanisms such as Kerberos [25] or Sun's secure RPC system [26]. However, it would be straightforward to elaborate our notion of identity and to allow each address space to have multiple identities. In particular, we could borrow from the work of Lampson, Wobber, et al. [12, 27]; our type `Ident.T` is a simplified version of the type `Auth` described in that work.

It is not hard to imagine other schemes for communicating identities. For example, identity objects could be passed explicitly or implicitly on every call. In choosing our scheme, we have been careful to avoid language changes, and have attempted to minimize overhead.

2.5 Discussion: Using ACLs, Capabilities and Identities

One of the main applications of identity objects is in bootstrapping trust. A client typically obtains its first secure network object as result of an invocation on an insecure network object, often a name server. A priori, there is no trust between the owner of the secure network object and the client. This trust is established when client and server exchange and test identity objects. Once this has been done, they may choose to pass other secure network objects. Because these objects are obtained as part of secure invocations, further checks of identity may not be necessary.

For example, an insecure name server may export a secure network object `fs` of type `FileServer.T` providing access to a remote file server:

```
INTERFACE FileServer;

IMPORT AuthNetObj, Ident, Buffer;

TYPE
 T = AuthNetObj.T OBJECT METHODS
   owner(): Ident.T;
   open(id: Ident.T; name: TEXT): File;
 END;

 File = AuthNetObj.T OBJECT METHODS
   read(): Buffer.T;
   write(Buffer.T): INTEGER;
 END;

END FileServer;
```

By convention this object provides an `owner` method which returns the identity of the principal on whose behalve the file server is running. This result is used by a client to authenticate the file server.

Although `fs` is secure, anyone can obtain `fs` and invoke its methods. Therefore, it is reasonable for the file server to expect an identity as argument of any method invocation. When a client `c` issues the call `fs.open(Ident.Mine(), "/etc/motd")`, the `open` method should check the identity of `c`. If this check succeeds, the `open` method may return another secure network object `f`, which represents the open file `"/etc/motd"`. Client `c` may then invoke `f.read()`. Because `c` and the file server have authenticated one another, `c` and the owner of `f` may choose not to authenticate one another further, even though the owner of `f` need not be the file server.

Identity objects have important applications beyond bootstrapping. They allow a server to confine the use of capabilities to particular clients. Additionally, identities can be logged to construct an audit trail.

In our example, `c` could pass its identity to the `read` method of `f`, which could check that `c` is a particular user, or belongs to a particular group. The identity check provides protection even if `c` publishes `f`. The `read` method could also record the names of all callers, for future inspection.

Another important application of the type `Ident.T` is the implementation of various authorization mechanisms, and particularly reference monitors with ACLs. Several designs are possible. For example, a secure network object may include a method `checkACL` for making access-control decisions; the arguments of `checkACL` are a mode `m` (such as `read` or `write`) and an identity `i`; the result is a boolean that indicates whether the user with identity `i` is allowed access with mode `m` to the object. In the intended implementation, `checkACL` compares `i` with names kept in lists (that is, in ACLs). Additional methods enable the modification of the ACLs. When group-membership checks are needed, `checkACL` can consult group registries across the network using secure invocations.

2.6 Readers and Writers

Modula-3 includes buffered streams, called readers and writers. For example, a reader might be the stream of data from a file or from a terminal. The network objects system allows readers and writers to be passed between address spaces, as follows: an address space creates a reader or writer, and passes it to one other address space, which reads from it or writes to it, but never passes it. The two address spaces can then use the reader or writer to transmit data directly on their underlying network connection, without the overhead of method invocations.

Readers and writers are treated specially when passed in secure invocations:

6. If a reader or writer `rw` is passed as argument or result of a secure invocation, operations on `rw` have the same security guarantees as the invocation.

For instance, if a reader `rd` is passed as argument in the invocation `o.m(rd)`, and `o` is of type `AuthNetObj.T`, then operations on `rd` are secured in the same way as operations on an object of type `AuthNetObj.T`. Therefore, data read from `rd` is protected just as if it had been passed as an argument in the invocation. The example given in section 2.7 further illustrates the use of readers and writers.

An alternative treatment could be based on new types for readers and writers with security properties (analogous to `AuthNetObj.T` and `SecNetObj.T`). Modula-3 does not support multiple inheritance, so introducing these new types could have been troublesome. We have chosen our scheme in order to allow existing classes of readers and writers to be made secure without modification.

2.7 An Example

Let us consider a trivial terminal server that offers shells for remote users. Before a user gets a shell, the server obtains the user's identity, both to verify that the user is legitimate and to associate the identity with the shell. On the other hand, the user obtains the server's identity, and can check that the server is the expected one and not an impostor. Once the user has the shell, the user's commands and their results are protected against tampering, replay, and eavesdropping.

The terminal server exports the following interface:

```
INTERFACE STS;

IMPORT SecNetObj, Ident, Rd, Wr;

TYPE
  T = SecNetObj.T OBJECT METHODS
    owner(): Ident.T;
    create_shell(id: Ident.T; rd: Rd.T; wr: Wr.T);
  END

END STS.
```

This interface declares the object type `STS.T` as a subtype of `SecNetObj.T` with two new methods, `owner` and `create_shell`. The `owner` method returns the identity of the server (by calling `Ident.Mine`). The `create_shell` method creates a connection, starts a shell with the identity of the user, and connects the streams `rd` and `wr` as standard input and output, respectively.

A simple client program might be:

```
MODULE Client EXPORTS Main;

IMPORT NetObj, Ident, STS, Stdio;

CONST serverHost = "foo.bar.ladida";

VAR
  agent: NetObj.Address;
  sts: STS.T;
  server_ident: Ident.T;
BEGIN
  agent := NetObj.Locate(serverHost);
  sts := NetObj.Import("STS", agent);
  server_ident := sts.owner();
  IF  server_ident.userName() = "root"
  AND server_ident.hostName() = serverHost
  THEN
    sts.create_shell(Ident.Mine(),
        Stdio.stdin, Stdio.stdout);
  ELSE
    (* the server is an impostor *)
  END;
END Client.
```

The client program imports an object `sts` of type `STS.T` and verifies the identity `server_ident` of its owner. If this check succeeds, the client program starts a shell by calling `sts.create_shell`. Since `STS.T` is a subtype of `SecNetObj.T`, the shell's standard input and output benefit from the guarantees associated with `SecNetObj.T`.

3 Authentication Agents

So far we have focused on the design of a programming interface for security; in the remainder of the paper we describe our implementation for this programming interface. Our system has two main components, the authentication agent and the run-time system. We describe the first in this section and the second in the next.

In our system, each node runs an authentication agent. An authentication agent is a process that assists application address spaces for the purposes of security. It communicates with its local clients only via local secure channels. In our implementation, the authentication agent is a user-level process; as local secure channels we have used Unix domain sockets in one version of our implementation and System V streams in another. Each agent is responsible for managing identities and keys, as follows.

When an address space receives an identity object, it can ask the local agent for the corresponding user name and host name. The agent answers this question by communicating with its peers; each agent knows the name of its host and the user names associated with its clients.

The agent also provides *channel keys.* A channel key is an encryption key shared by two address spaces. The agent performs key exchange with its peers in order to generate these channel keys for its clients.

When the agent negotiates a new channel key, it also negotiates an expiry time for the key and a key identifier. A key identifier allows the sender of a message to tell the receiver which key was used for constructing the message. The key identifier can be transmitted in clear as part of the message, while the key itself should not be. The agent picks key identifiers so that they are unambiguous.

Our design encapsulates all of the key exchange machinery in the authentication agent. We have tried two implementations of key exchange, one based on our own protocol and another that relies on Sun's secure RPC authentication service. The change of implementation was transparent to user address spaces. In the second implementation, we wrote 1400 lines of C code for the agent.

We took the idea of using an authentication agent from the work of Wobber et al. [27]. The authentication agent described in that work is more elaborate than ours; for example, it deals with delegation and supports channel multiplexing. These features could be incorporated in our system, though they may preclude the use of off-the-shelf authentication software.

4 The Run-time System

In this section we describe the security-related code that runs in application address spaces. We adapted the original code of Modula-3 network objects, making a few changes:

- We defined secure network object references to be capabilities.
- We extended the protocol used by network objects with security information and with functions like message digesting and encryption.
- We modified the run-time system, adding marshaling code and cryptography.

The changes were relatively small. We added 2500 lines of Modula-3 code to the run-time system. We also integrated public-domain implementations of DES [16] and MD5 [22] in C. We did not modify the overall structure of network objects.

4.1 Capabilities

The wire representation of an insecure network object is a pair (s, objid), where:

- s is an address space identifier for the owner of the object,
- *objid* is an object identifier, which distinguishes this object from others with the same owner.

The wire representation of a secure network object is a tuple

$$(s, \mathit{objid}, \mathit{capid}, \mathit{key}, \mathit{exp})$$

with the following additional components:

- *capid* is a capability identifier,
- *key* is a key associated with *capid*,
- *exp* is an expiration time.

The tuple $(s, \mathit{objid}, \mathit{capid}, \mathit{key}, \mathit{exp})$ is a capability. The capability is created by the owner of the object. The key is the secret that is shared between the holders of the capability and the owner of the object. The key may never appear in clear on the network: it is encrypted using a channel key when transmitted between address spaces in secure invocations. The components s, *objid*, and *capid* determine *key* uniquely, and thus serve as key identifier.

The capability becomes invalid once its expiration time has passed. The use of an expiration time can be beneficial in revoking a capability; it also limits the use of the key. The client run-time system refreshes the capabilities it holds before they expire. This is transparent to the client application. Each secure network object has a hidden method that the client run-time system can call to obtain a fresh capability for the object.

In general, more than one capability may be in use for any given object. The owner of an object maintains a set of valid capabilities for the object, and is careful not to give out capabilities that are about to expire.

4.2 Protocol

If a client c holds a network object reference (s, objid) for an object of type `NetObj.T`, an invocation of a method of the object consists of a request from c and a reply from the owner s. The request is $(\texttt{Request:}\ c, s, \mathit{objid}, \mathit{reqdata})$, where *reqdata* includes the method name and arguments of the invocation. The reply is $(\texttt{Reply:}\ c, s, \mathit{repdata})$, where *repdata* contains the results of the invocation.

If a client c holds a capability $(s, \mathit{objid}, \mathit{capid}, \mathit{key}, \mathit{exp})$ for an object of type `AuthNetObj.T`, both the request and the reply are modified. We assume that the local authentication agents for c and s have authenticated one another, and that they make available a channel key *chankey* for communication between c and s. This key will be used for signing the request and the reply, and sometimes for encryption, as follows.

The request has two parts, a body and a signature. The body, *reqbody*, is:

$$(\texttt{Request:}\ c, s, \mathit{objid}, \mathit{capid}, \mathit{mid}, \mathit{reqdata})$$

where *mid* is a message identifier that c has never before attached to a request signed using *chankey*. The signature is:

$$\mathit{Hash}(\mathit{reqbody}, \mathit{chankey}, \mathit{key})$$

where *Hash* is a cryptographic hash function (a one-way message digest function such as MD5). Upon receipt of the request, s verifies that *mid* is not a duplicate and checks the signature.

Like the request, the reply has a body and a signature. The body, *repbody*, is:

$$(\texttt{Reply:}\ c, s, \mathit{mid}, \mathit{repdata})$$

The signature is:

$$Hash(repbody, chankey, key)$$

Upon receipt of the reply, the client verifies that c, s, and *mid* match those of the request, and checks the signature.

Both *reqdata* or *repdata* may contain capabilities and identity objects. These are treated specially:

- On the wire, capability keys are encrypted under *chankey*.
- The wire representation of an identity object is simply a placeholder. When c (or s) receives such a placeholder, it replaces the placeholder with an identity object constructed locally; the methods of this identity object call the local authentication agent to obtain the user name and the host name associated with s (or c, respectively).

For an object of type `SecNetObj.T`, the protocol is the same, except that both the request and the reply are completely encrypted under the channel key *chankey*.

So far we have ignored readers and writers, because their discussion is independent of that of the protocol and because security for them is obtained by reducing them to secure network objects. In the existing implementation of network objects, passing a reader or writer `rw` amounts to passing a network object called a *voucher*. The voucher has a type with special marshaling routines; it mediates communication over `rw`. For security, we arrange that the type of the voucher be a subtype of `AuthNetObj.T` or `SecNetObj.T`.

We can now justify the guarantees listed in section 2:

1. The signatures provide integrity. Since only c and s have *chankey*, only they could have generated $Hash(reqbody, chankey, key)$. If s never generates a request of the form (`Request`: $c, s, \ldots$), which appears to come from c, then s knows that only c could have generated $Hash(reqbody, chankey, key)$, and hence that c must have endorsed *reqbody*. Similarly, c knows that s must have endorsed *repbody*. Moreover, the request and reply are matched because they both include *mid*.
2. At-most-once semantics is guaranteed because s checks that at most one request from c includes *mid* and is signed using *chankey*; and because c accepts replies only for outstanding calls.
3. c and s demonstrate knowledge of the capability key *key* by transmitting $Hash(reqbody, chankey, key)$ and $Hash(repbody, chankey, key)$. Thus, *key* does not appear in clear on the wire. Any capability keys for other secure network objects are protected from eavesdroppers by encryption under *chankey*.
4. Since only c and s have *chankey*, encryption under *chankey* protects the secrecy of the invocation against eavesdroppers.
5. When an address space sends an identity object, the receiver constructs an identity object that reflects the true identity of the sender. An address space that runs our code will never pass an identity other than its own; but even if an address space runs other code and succeeds in passing an identity other than its own, it will not be believed.
6. When a reader or writer `rw` is passed in the invocation of a method of an object of type `AuthNetObj.T` (or `SecNetObj.T`), the voucher for `rw` has type `AuthNetObj.T` (or `SecNetObj.T`, respectively). Therefore, since operations on `rw` are mediated by the voucher, they have the same security guarantees as the invocation.

4.3 Implementation in the Run-time System

In order to implement our design, we made only a few changes to the existing network objects code. Most of these changes were in the implementation of `StubLib`, the interface that the network objects run-time system provides to stubs. (Stubs are machine-generated subroutines that are responsible for marshaling and unmarshaling arguments and results of remote invocations.) Some changes were also required in the `StubLib` interface itself, but they were minor; we added only two lines of code to the stub generator to accommodate them.

The network objects system allows the use of multiple transport protocols (such as TCP or X.25). Our security implementation is independent of the choice of transport protocol, so one can add support for new protocols without altering or writing security code.

In the network objects system, when two address spaces a_1 and a_2 communicate, each of them has a *connection object* consisting of a reader and a writer. Data written into a_1's writer appears in a_2's reader, and vice versa. We embed the cryptography necessary for the protocol of section 4.2 in our implementation of connection objects. We do so by defining readers and writers that apply cryptography to specified byte ranges in the data they transmit. These readers and writers have methods for computing cryptographic digests, for encryption, and for decryption.

As described in section 4.2, each message of our protocol includes an identifier *mid* for protection against replays. Were all address spaces single threaded, a simple sequence number would make a convenient message identifier. In order to accommodate multi-threaded address spaces, we associate each message with a *secure channel*, and let *mid* include a secure channel identifier. Next we explain secure channels and their use.

At each point in time, a secure channel from an address space a_1 to an address space a_2 determines a channel key and a sequence number. (Two secure channels may have the same key.) Both a_1 and a_2 know the secure channel identifier and keep track of the key and the sequence number associated with the secure channel. With each method invocation from a_1 to a_2 that uses a secure channel, both a_1 and a_2 increment the corresponding sequence number. When the key for a secure channel is half way to its expiration, a_1 asks its authentication agent for a new key; on seeing a new key identifier, a_2 obtains the new key from its own agent.

In the normal case, a secure invocation from a_1 to a_2 proceeds as follows. First a_1 chooses a secure channel from a_1 to a_2 on which there is no outstanding invocation (and sets up a new secure channel if none is available). Then a_1 constructs the request using the channel key for the secure channel; *mid* is the concatenation of the sequence number and the secure channel identifier. When a_2 receives the request, it compares the sequence number in *mid* against its version of the sequence number. The reply uses the same key and message identifier as the request. When a_1 receives the reply, it verifies that the key is still current and checks *mid*.

In our current implementation *mid* is tied to a specific channel key and is expected to be constant over the course of an invocation. It is therefore awkward to deal with key expiry during the following circumstances: invocations that return a reply after a long wait, large streams of data, and intermittent streams. It might be possible to negotiate long-lived security associations between address spaces and use an association identifier to identify a single space of sequence numbers for multiple keys. This, however, requires that either the sequence number space is large enough for the security associations, or that there exists a way for renegotiating security associations.

We prefer a different scheme in which we make use of uni-directional keys. In this scheme the authentication agent negotiates a key, a key identifier, and its expiry time for communication from address space a_1 to a_2. A sender can simply attach a stream of sequence numbers to a new key when it is first used. When the sequence number space is exhausted, a new key can be negotiated. Furthermore, address spaces can negotiate a key per thread of control. In this structure, re-keying in the midst of an invocation is straightforward since sequence numbers are not specific to an invocation.

In the new scheme the *mid* would be changed to include a key identifier, a sequence number, and a call number. The key identifier and sequence number are chosen by the sender. The recipient can assume that the sequence number are monotonically increasing because different keys are used for different threads of control. The call number is chosen by the invoker. It identifies the method invocation instance but need not to protect against replay. Notice that the *mid* is not the same in the request and reply as in the original scheme.

5 Performance and Applications

Next, we describe our experiments with secure network objects. We discuss the performance of our system and two example applications.

5.1 Performance Measurements

We measured the performance of our system with the same tests used for the original implementation of network objects [2]. Table 1 presents a subset of our measurements that characterizes the performance of secure network objects in comparison with the performance of the original implementation.

We measured round-trip invocation latency for remote invocations. The first column of Table 1 (labelled *Old*) gives the performance of method invocations in the original implementation. The remaining three columns concern method invocations in our implementation, to objects of type `NetObj.T`, `AuthNetObj.T`, and `SecNetObj.T`, respectively. The rows of Table 1 indicate method invocations with different sorts of arguments and with no results. In the null test, the method has no arguments. In the ten integer test, the method takes ten 64-bit integers as arguments. In the existing surrogate test, a network object is marshaled to a receiver that already has a surrogate for it. The type of the argument matches the security level of the invocation, so for instance in the column *Authentic* the argument has type `AuthNetObj.T`. The new surrogate test is similar, except that the argument is unknown to the receiver; therefore, the receiver must create a new surrogate and register it at the owner (for garbage collection).

The tests do not include the establishment of a channel key. The cost of establishing a channel key depends on the implementation of the authentication agent; a

Test	*Old*	*Insecure*	*Authentic*	*Secret*
Null	708	772	870	991
Ten integer	747	830	929	1244
Existing surrogate	751	822	973	1146
New surrogate	1896	2044	2263	2430

Table 1. Performance of secure network objects (in μs/call).

typical cost may be in the order of tens of milliseconds. In most cases, this cost is insignificant when amortized over multiple secure invocations.

We performed all our measurements using DEC 3000/700 workstations, which contain DECchip 21064a processors at 225 MHz. On these workstations, our MD5 implementation runs at over 15 MBytes/sec; our DES implementation at 990 KBytes/sec. We used two workstations connected by an ATM network with a point-to-point bandwidth of over 16 MBytes/sec.

Our measurements show our system adds a non-trivial cost to insecure invocations. This additional cost is largely due to the expanded size of the wire representation of network objects and of packet headers, and to the associated processing. Secure network objects require a 24-byte object representation that identifies both the host and address space of the object owner. The packet header format similarly requires a host and address space identifier as well as a message identifier.

Authentic invocations add a fixed minimum MD5 overhead of nearly 40 μs per invocation. Each call and return packet requires at least two 64-byte MD5 calculations, and both client and server must perform these. The remainder of the incremental cost can be attributed to the generation of a message identifier, and the checking of timestamps, message identifiers, and digests. The calls where an object is used as argument incur the additional overhead of encrypting each object capability under DES.

The difference between authentic and secret calls is primarily due to DES processing. Since the method identifier, return code, and MD5 digest are encrypted as well as all arguments and results, there is a minimum of 96 bytes of DES encryption/decryption ($\simeq$100 μs) per secret invocation. (The encryption of the MD5 digest is not necessary, and we could eliminate it for even better performance.) The ten integer test adds 80 bytes of arguments, to be encrypted and decrypted; this accounts for another 160 μs.

We have also measured the performance of readers and writers. Streams with authenticity are currently limited to about 8 MBytes/sec. We observed that 40% of the CPU cost is attributable to buffering and TCP overhead; this explains why the bandwidth of streams with authenticity is limited to around 55% of the raw MD5 bandwidth. Streams with both authenticity and secrecy are severely limited by the speed of DES. According to our measurements, such streams have a throughput of about 870 KBytes/sec; this implies that nearly 90% of the CPU is dedicated to DES computation.

As these measurements demonstrate, the performance of our system is acceptable for many applications. However, we believe that there remains much room for optimization.

5.2 Example Applications

To date, we have had two preliminary but encouraging experiences in the application of secure network objects.

In the first application, we have implemented a secure version of an answering service written by Rob DeLine. This service is part of a telecollaboration system and implements an answering machine for multi-media messages. The answering machine is a network object with methods `create`, `retrieve`, and `delete`. When a message is created, it is stored under a special name, called a *cookie*; the cookie is e-mailed to the intended recipient of the message. An e-mail reader can then present the cookie to retrieve or to delete the message.

The original answering service is not secure. In particular, cookies are essentially used as capabilities, yet they are communicated in clear and are easy to guess. Therefore, the service has neither authenticity nor secrecy.

Our version of the answering service addresses these problems. Because of the performance limitations of software encryption, we have not provided secrecy but only authenticity. In our version, the answering machine is a network object of type `AuthNetObj.T`; it would be easy to make it a network object of type `SecNetObj.T` instead. We store the names of both the sender and intended recipient with each message; appropriate identity objects must be passed as arguments of calls to the methods `create`, `retrieve`, and `delete`. For example, only the sender and intended recipient of a message can delete the message.

Our second application is a secure version of Obliq [3]. Roughly, Obliq is to Modula-3 as Tcl is to C. Obliq is an untyped, interpreted scripting language that supports distributed object-oriented computation. For example, it can be used to program computing agents that roam over a network.

In our version of Obliq, each object is implemented as a network object of type `AuthNetObj.T`. When an Obliq object o is exported, it is explicitly tagged with a list of the principals that may import it. The Obliq run-time system encapsulates o in a reference monitor of type `AuthNetObj.T`. This reference monitor provides two methods: a method for access to o and a method that returns the identity of the owner of the reference monitor. The former method is responsible for performing access control checks. The latter method allows a client that imports o to verify that the server is the expected one.

We have implemented the secure version of Obliq that we have just described. This implementation provides evidence that we can build useful security mechanisms for Obliq fairly easily. On the other hand, we do not yet have sufficient experience to decide which security mechanism is most appropriate.

6 Related work

There has been much work on capabilities, in various contexts. Communication systems with a pure capability model, like that of Amoeba [15], suffer from the confinement problem. Several restrictions and variations of the capability model have been proposed. For example, Gong suggests adding identities to capabilities [7]; Bacon et al. suggest restricting their lifetime [1]. Karger's dissertation describes several others [9]. The ideas in our use of capabilities can be traced back through a vast literature.

There has also been substantial work in the area of security and network communication systems (e.g., [25, 14, 21]). However, to our knowledge, there is at present no object-oriented network communication system with support for security. Next we discuss two recent efforts in this area.

The Object Management Group has requested technology for integrating security in CORBA [19] and has received submissions [20]; security work is currently in progress in the Object Management Group. A general architecture for security in CORBA has also been proposed, and an implementation of the architecture was announced as future work [5]. In summary, there seems to be significant interest and activity on security in CORBA, but (at the time of the writing of this paper) not yet any resolution or experimental results.

The Spring object-oriented operating system [13] supports both ACLs and capabilities. An accurate comparison with secure network objects is difficult, as there has not been a complete description of security in Spring. Spring has more expressive capabilities than our system; for example, several capabilities for an object can give different privileges. Perhaps for this reason, the combination of identities and capabilities is less important in Spring, and has been explored less [8]. To date, Spring security has been implemented only on a single processor; all network communication is unprotected and capabilities are passed in clear.

7 Conclusions

Secure network objects behave like insecure network objects, but preserve the intended semantics even in the presence of an active attacker; optionally, secure network objects provide secrecy from eavesdroppers. In addition, identity objects form the basis for authentication in our system. The combination of secure network objects with identity objects leads to a simple programming model and a simple implementation, both in the spirit of the original network objects. Thus we have integrated security and network objects.

Overall, we felt that object-orientation was helpful. Not surprisingly, we found that objects give rise to natural units of protection. We also took advantage of the object type system to specify security properties, directly and economically.

In our description, we have tried to be precise, but not formal. We believe that a more formal study would be interesting. In particular, it would be worthwhile to give notations and rules for reasoning about secure network objects.

Acknowledgments

We would like to thank Andrew Birrell, Greg Nelson, and Luca Cardelli for many helpful discussions; and Roger Needham, Cynthia Hibbard, Tim Mann, Rustan Leino, Andrew Tanenbaum, Philip Homburg, Raoul Bhoedjang, Tim Ruhl, and Greg Sharp for helpful comments on the paper.

References

[1] J. Bacon, R. Hayton, S. L. Lo, and K. Moody. Extensible access control for a hierarchy of servers. *ACM Operating Systems Review*, 28(3):4–15, July 1994.

[2] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages

217–230, Dec. 1993. To appear in *Software: Practice and Experience.*

[3] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Jan. 1995.

[4] W. Cheswick. An evening with Berferd, in which a hacker is lured, endured, and studied. In *Proceedings of the Usenix Winter '92 Conference*, 1992.

[5] R. Deng, S. Bhonsle, W. Wang, and A. Lazar. Integrating security in CORBA based object architectures. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 50–61, May 1995.

[6] J. Dennis and E. van Horn. Programming semantics for multiprogrammed computation. *Communications of the ACM*, 9(3):143–155, Mar. 1966.

[7] L. Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, May 1989.

[8] G. Hamilton. Personal communication, 1994 and 1996.

[9] P. A. Karger. *Improving Security and Performance for Capability Systems.* PhD thesis, Cambridge University, Oct. 1988.

[10] B. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, Oct. 1973.

[11] B. Lampson. Protection. *ACM Operating Systems Review*, 1(8):18–24, Jan. 1974.

[12] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.

[13] J. Mitchell, J. Gibbons, G. Hamilton, P. Kessler, Y. Khalidi, P. Kougiouris, P. Madany, M. Nelson, M. Powell, and S. Radia. An overview of the Spring system. In *IEEE Compcon Spring 1994*, Feb. 1994.

[14] R. Molva, G. Tsudik, E. van Herreweghen, and S. Zatti. Kryptoknight authentication and key distribution system. In *Proceedings of the European Symposium on Research in Computer Security*, Nov. 1992.

[15] S. J. Mullender, A. S. Tanenbaum, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th IEEE conference on Distributed Computing Systems*, June 1986.

[16] National Bureau of Standards. Data encryption standard. FIPS 47, 1977.

[17] R. Needham. Names. In S. Mullender, editor, *Distributed Systems*, chapter 12, pages 315–327. Addison-Wesley, second edition, 1993.

[18] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, 1991.

[19] Object Management Group. Common object request broker architecture and specification. OMG Document number 91.12.1.

[20] Object Management Group. OMG documents. See URL: `http://www.omg.org/`.

[21] Open Software Foundation. Introduction to OSF DCE. Revision 1.0, 1992.

[22] R. Rivest and S. Dusse. RFC 1321: The MD5 message-digest function. Internet Activities Board, 1992.

[23] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *IEEE International Conference on Distributed Computer Systems*, May 1986.

[24] E. H. Spafford. The Internet worm program: An analysis. *Computer Communication Review*, 19(1):17–57, Jan. 1989.

[25] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An authentication service for open network systems. In *Usenix 1987 Winter Conference*, pages 191–202, Jan. 1988.

[26] Sun Microsystems. RFC 1057: RPC: Remote procedure call protocol specification: Version 2. Internet Activities Board, June 1988.

[27] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, Feb. 1994.