Exploiting Fine-Grain Configurability for Secure Communication

Matti A. Hiltunen   Sumita Jaiprakash   Richard D. Schlichting

TR 99-08

Exploiting Fine-Grain Configurability for Secure Communication[1]

Matti A. Hiltunen, Sumita Jaiprakash, and Richard D. Schlichting

TR 99-08

Abstract

Current protocols such as IPSec and TLS that provide communication security for network applications allow customization of certain security attributes and techniques, but in limited ways and without the benefit of a single unifying framework. Here, the design of a highly-customizable extensible service called SecComm is described in which attributes such as authenticity, privacy, integrity, and non-repudiation can be customized in arbitrary ways. With SecComm, applications can open secure communication connections in which only those attributes selected from among a wide range of possibilities are enforced, and are enforced using the strength or technique desired. SecComm is being implemented using the Mach MK 7.3 operating system and Cactus a system for building configurable communication services. In Cactus different properties and techniques are implemented as software modules called micro-protocols that interact using an event-driven execution paradigm. This design approach has a high degree of flexibility, yet provides enough structure and control that it is easy to build collections of micro-protocols realizing a large number of diverse properties.

May 3, 1999

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by the Office of Naval Research under grant N00014-96-0207, the Defense Advanced Research Projects Agency under grant N66001-97-C-8518, and the National Science Foundation under grant CDA-9500991.

1 Introduction

A significant trend in network security design is the use of communication protocols that allow customization of security attributes for explicitly managing the cost/benefit tradeoff. For example, IPSec [KA98], a set of protocols being developed by the IETF to support secure packet exchange at the IP layer, provides two security options. The *authentication header* (AH) option does not encrypt the data contents of the packet, but provides optional authenticity, integrity, and replay prevention by adding an AH that contains a cryptographic message digest. The *encapsulating security payload* (ESP) option provides privacy by encrypting the data contents of the packet and optional authenticity, integrity, and replay prevention using a message digest. Other protocols such as SSL [FKK96], S-HTTP [RS98], and TLS [DA99] offer similar features.

While this trend is positive, we argue that current efforts do not go far enough, and that it should be possible to customize essentially all security attributes related to communication and to customize them in essentially arbitrary ways. To support this argument, we present the design of a highly-customizable extensible secure communication service called SecComm. With SecComm, applications can open secure communication connections in which the security attributes and the strength of guarantees associated with each attribute can be customized at a fine-grain level. For example, SecComm allows an attribute to be guaranteed using arbitrary combinations of security algorithms, and it supports extensibility by allowing the addition of new algorithms as separate modules. At another level, our approach can also be viewed as a technique for implementing protocols such as IPSec, SSL, S-HTTP, and TLS in a modular and extensible fashion.

The customization and extensibility attributes of SecComm derive from the use of Cactus as the underlying implementation platform [HSH$^+$98]. Cactus is a framework for constructing highly-configurable network services, where each service attribute or variant is implemented as an independent module called a *micro-protocol*. Micro-protocols are executed using an event-driven execution paradigm and can share variables and data structures, both of which enhance independence between micro-protocols. A customized version of the service is then constructed by choosing micro-protocols based on the desired properties. Several prototype implementations of Cactus have been constructed, including one written in C that runs on Mach version MK 7.3 from OpenGroup [Rey95], another written in C++ that runs on Solaris and Linux, and a third written in Java that runs on multiple platforms. SecComm is being implemented using the MK version of Cactus on a cluster of Pentiums. Other prototype services that have been successfully implemented using Cactus or the predecessor Coyote system [BHSC98] include group RPC [HS95], membership [HS98], and a real-time channel abstraction [HSH$^+$98].

This paper has several goals. The first is to argue that fine-grain configurability and extensibility are valuable characteristics for realizing security attributes in future communication services. The second is to describe a realization of this philosophy in the form of the SecComm service. The last is to demonstrate specifically how Cactus can be used to build services such as SecComm.

2 Configurable Security

2.1 Overview

Our system model consists of a set of machines connected by a local- or wide-area communication network. Application level processes communicate by using a *protocol graph* that typically consists of IP and some transport level protocol such as TCP or UDP. The SecComm protocol may be inserted into the protocol stack on top of the transport protocol or on top of IP, as illustrated in Figure 1. (The internal structure of SecComm is explained further in section 3.) The SecComm service is generally independent of the choice of the lower-level communication protocol. The method used to structure the protocols hierarchically is an independent issue. For example, in our prototype implementation, the *x*-kernel [HP91] is used for this purpose, but other approaches could be used as well. It is also easy to structure SecComm as a middleware service built on top of TCP sockets.

For each separate application level communication connection, a *session* is created through the SecComm service to allow each connection to have customized security attributes. The SecComm service described in this paper focuses on client/server style communication, where the communicating principals have distinct roles in the communication and each communication direction may require different security guarantees. However, the approach and API support other paradigms such as group communication.

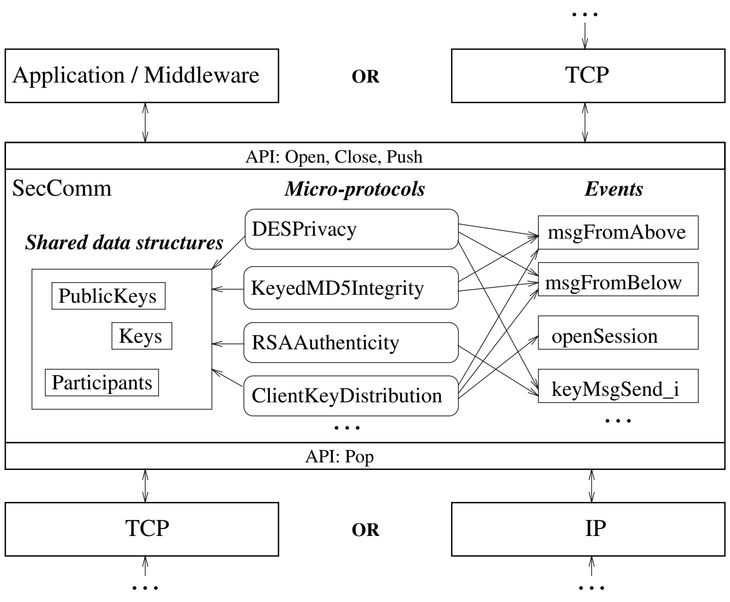

Figure 1: System protocol stack.

The system may also contain centralized security principals that provide authentication and other security services. These principals could include Certification Authorities (CA) that provide certified public keys of other principals in the system and Key Distribution Centers (KDC) that provide authentication using secret key methods as well as session keys.

2.2 Security attributes

As a first step towards exploiting fine-grain configurability, orthogonal security attributes and variants are identified. While not an exhaustive list, these include:

- **Authenticity**. Ensures that a receiver can be certain of the identity of the message sender. Can be implemented using public key cryptography [Hel78], any shared secret, or a trusted intermediary such as Kerberos [SNS88, NT94].

- **Privacy**. Ensures that only the intended receiver of a message is able to interpret the contents. Can be implemented using any shared secret, public key cryptography, or combinations of methods.
- **Integrity**. Ensures that the receiver of a message can be certain that the message contents have not been modified during transit. Some authenticity and privacy methods also provide integrity as a side effect if the message format has enough redundancy to detect violations. Additional redundancy can be provided using message digest algorithms such as MD5 [Riv92]. Integrity can be provided without privacy, but at a minimum, the message digest itself must be protected.
- **Non-repudiation**. Ensures that a receiver can be assured that the sender cannot later deny having sent the message. Relies on authenticity provided by public key cryptography and requires that the receiver store the encrypted message as proof.
- **Replay Prevention**. Ensures that an intruder cannot retransmit an old message unnoticed. Can be implemented using timestamps, sequence numbers, or other such nonces in messages. Typically used in conjunction with authenticity, privacy, or integrity since otherwise it would be trivial for an intruder to generate a new message that appears to be valid.

In addition to these attributes, other aspects of the secure communication service can be customized. In particular, there are numerous techniques for creating and distributing the secret keys required by many of the cryptographic methods that can be varied. These options include use of pre-established keys, use of keys generated by one or all of the communicating parties (e.g. [DH76]), and use of external key distribution centers [SNS88, NT94].

2.3 Cactus implementation platform

A service in Cactus is implemented as a *composite protocol*, with each semantic variant of a security attribute or other functional component within the composite protocol implemented as a *micro-protocol* (Figure 1). A micro-protocol is, in turn, structured as a collection of *event handlers*, which are procedure-like segments of code that are executed when a specified *event* occurs. Events are used to signify state changes of interest, such as "message arrival from the network". When such an event occurs, all event handlers bound to that event are executed. Events can be raised explicitly by micro-protocols or implicitly by the Cactus runtime system. Execution of handlers is atomic with respect to concurrency.

Event handler binding, event detection, and invocation are implemented by the Cactus runtime system that is linked with the micro-protocols to form a composite protocol. Once created, a composite protocol can be composed in a traditional hierarchical manner with other protocols to form the application's protocol graph. For example, as noted above, in the MK prototype, the *x*-kernel is used.

The primary event-handling operations are:

- *bid* = **bind**(*event, handler, order, static_args*): Specifies that *handler* is to be executed when *event* occurs. *order* is a numeric value specifying the relative order in which *handler* should be executed relative to other handlers bound to the same event. When the handler is executed, the arguments *static_args* are passed as part of the handler arguments.
- **raise**(*event, dynamic_args, mode, delay*): Causes *event* to be raised after *delay* time units. If *delay* is 0, the event is raised immediately. The occurrence of an event causes handlers bound to the event to be executed with *dynamic_args* (and *static_args* passed in the **bind** operation) as arguments. Execution can either block the invoker until the handlers have completed execution (*mode* = SYNC) or allow the caller to continue (*mode* = ASYNC).

Other operations are available for unbinding handlers from events, creating and deleting events, halting event execution, and canceling a delayed event.

In addition to the flexible event mechanism, Cactus supports shared data that can be accessed by the micro-protocols in a composite protocol. Shared data structures are easy to use since the atomic execution of event handlers eliminates most concurrency problems.

Finally, Cactus provides a message abstraction that supports configurable services. The main features provided by this message abstraction are named attributes that have scopes corresponding to the composite protocol (*local*), the protocols on a single machine (*stack*), and the peer protocols at the sender and receiver (*peer*). A customizable pack routine concatenates peer attributes to the message body for network transmission, or for operations such as encryption and compression. A corresponding unpack routine extracts the peer attributes from a message at the receiver.

3 SecComm Design

3.1 Application programming interface

The SecComm composite protocol exports the following operations:

- **Open**(participants,keys). Opens a session for a new communication connection, where *participants* is an array identifying the communicating principals. *keys* is an array containing any predefined keys required for the communication connection, such the principal's private key or master key for the KDC.
- **Push**(msg). Passes a message from a higher level protocol or application to the SecComm composite protocol to be transmitted with the appropriate security attributes to the participants.
- **Pop**(msg). Passes a message from a lower level protocol to the SecComm composite protocol to be decrypted, checked, and potentially delivered to a higher level protocol. When the SecComm protocol passes a message to the higher level and authentication is required, it adds a stack attribute AUTH_SENDER that is the ID of the authenticated sender.
- **Close()**: Closes a communication connection.

3.2 Shared data structures and events

The design has a clear separation between the cryptographic micro-protocols that use encryption keys and the micro-protocols responsible for creating and distributing these keys. To communicate this information between these two types of micro-protocols, the SecComm design uses the Cactus provisions for shared data structures. In particular, there is a shared table *Keys* accessed by micro-protocols based on indices passed as parameters at startup time. Any predefined keys passed in the **Open()** operation are also stored in *Keys*. Another shared data structure called *PublicKeys* caches known public keys. Finally, array *Participants* stores the identities of the principals involved in the communication connection.

Our prototype implementation of SecComm uses the cryptographic package Cryptlib [Gut98] to provide basic cryptographic functionality. Any cryptolibrary with the necessary functions could be used, however.

The design of SecComm uses a number of events for communication between micro-protocols and to initiate execution when messages arrive. These include:

- msgFromAbove(*msg*). Indicates that *msg* has arrived from a higher level protocol or application.
- msgFromBelow(*msg*). Indicates that *msg* has arrived from a lower level protocol or OS.
- openSession(). Indicates that a new secure communication session has been created, thereby initiating creation and distribution of session keys if needed.
- keyCreate_i(*key,length*). Indicates that a key of size *length* is to be created by the ith key creation micro-protocol.
- keyMsgSend_i(*msg*). Indicates that a message *msg* involved in the distribution of the ith key is about to be sent.
- keyMsgReceive_i(*msg*). Indicates the arrival of a message *msg* involved in the distribution of the ith key.
- keyMiss(*principal*). Indicates that the public key of *principal* is required.
- securityAlert(*msg*). Indicates that a potential security violation related to *msg* has been detected.

3.3 Micro-protocols

3.3.1 Overview

The abstract security attributes described in Section 2.2, as well as key creation and distribution, are implemented by one or more micro-protocols. When a number of micro-protocols implement variations of the same abstract property, we collectively refer to them as a *class* of micro-protocols. For example, the class of privacy micro-protocols includes DESPrivacy and RSAPrivacy micro-protocols that use DES and RSA algorithms, respectively. In many cases, all the micro-protocols in such a class have the same interaction with other micro-protocols, which simplifies the presentation of the service. Figure 2 illustrates the main micro-protocol classes and typical event interactions between them.

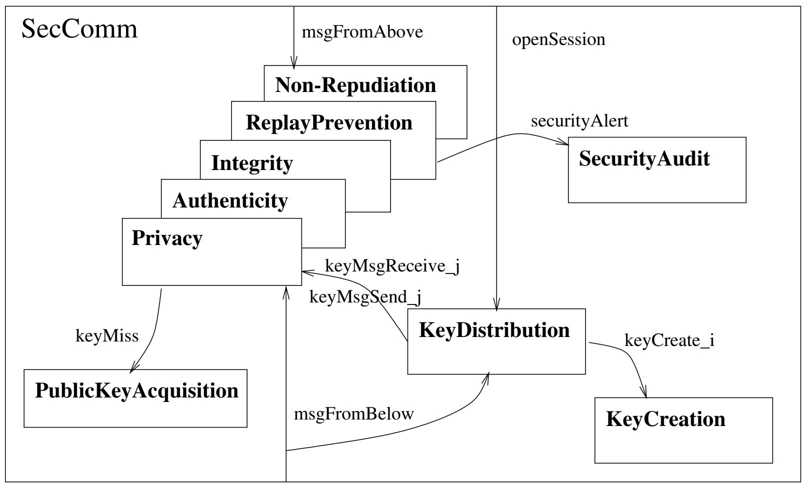

Figure 2: Micro-protocol classes and their interactions.

The privacy, authenticity, integrity, replay prevention, and non-repudiation micro-protocols all react to the same set of events, namely msgFromAbove, msgFromBelow, keyMsgSend_i, and keyMsgReceive_i. This design allows the same security micro-protocols to be used to secure both the normal data messages and the key exchange messages. Note, however, that the security requirements for key exchange are typically different than those for data exchange.

When designing the micro-protocols, it is important to take relative execution order into account for handlers bound to events msgFromAbove, msgFromBelow, keyMsgSend_i, and keyMsgReceive_i so that logical constraints between the attributes are satisfied. For example, replay prevention must be executed before data integrity so that any nonces added will be included in the message digest. Furthermore, it is important that the execution order at the receiver is the reverse of that at the sender, since cryptographic methods are typically not commutative. These constraints are implemented using the *order* argument on the **bind()** operation.

We now give an overview of the major micro-protocol classes.

3.3.2 Privacy

SecComm includes numerous privacy micro-protocols, ranging from those based on standardized cryptographic methods such DESPrivacy and RSAPrivacy, to others based on non-standard methods. The latter include OneTimePadPrivacy that encrypts a stream of messages by xoring it with a secret file that is shared by the sender and receiver. This method can provide very good privacy, but requires a shared file at least as long as the stream of messages to be exchanged. Other trivial privacy micro-protocols use simple substitutions and shifts. For example, the CaesarPrivacy micro-protocol adds a fixed constant passed as initialization parameter to each byte of the message, modulo 256. Such trivial methods do not provide a high level of privacy, but they may be enough the deter a casual observer. Moreover, combinations of the fast trivial methods used in conjunction with a standard method such as DESPrivacy may enhance privacy considerably. Other privacy techniques, such as steganography [JJ98], could easily be added to SecComm if desired.

The privacy micro-protocols are very simple, as illustrated in the case of the DESPrivacy micro-protocol in Figure 3. In the parameter list, *key* is an index in the Keys data structure, *order* is the relative order in which this security micro-protocol is to be applied to messages, *dir* indicates in which communication direction this micro-protocol is to be applied (client to server (CS), server to client (SC), or both (BOTH)), *mode* indicates if this micro-protocol is to be used for securing data messages, key exchange messages, or both, and *dist_index* indicates for which key distribution (if any) this micro-protocol is to be used. Note that the privacy micro-protocol packs the message before encryption so that all message attributes to be transmitted over the network will be encrypted. The pack routine combines all the peer attributes into the message body denoted by the DATA attribute tag. The initialization section of the micro-protocol is executed when a new SecComm connection is created (i.e., when a session is opened). The global variable *Role* indicates if this is the client or server end of the secure communication connection. This particular configuration of the micro-protocol uses the CBC mode of DES, but other modes such as ECB, CFB, and OFB [DES81] could easily be implemented.

DESPrivacy, like any other secret key based micro-protocol, requires that a shared secret key be established between the communicating partners in the Keys table. This can be accomplished using a predefined key passed to SecComm at session opening time or by using the KeyDistribution micro-protocols (see section 3.3.6). The basic RSAPrivacy is very similar except that it uses the receiver's public key to encrypt the message.

```
micro-protocol DESPrivacy(key,order,dir,mode,dist_index) {

    handler Encrypt(msg){
        pack(msg); setAttr(msg,DATA)= DES(getAttr(msg,DATA),Keys[myKey],ENCRYPT,CBC);
    }
    handler Decrypt(msg){
        setAttr(msg,DATA)= DES(getAttr(msg,DATA),Keys[myKey],DECRYPT,CBC); unpack(msg);
    }
    initial {
        myKey = key;
        if mode == DATA or mode == BOTH {
            if (dir == CS and Role == CLIENT) or (dir == SC and Role == SERVER) or (dir == BOTH)
                bind(msgFromAbove,Encrypt,order);
            if (dir == CS and Role == SERVER) or (dir == SC and Role == CLIENT) or (dir == BOTH)
                bind(msgFromBelow,Decrypt,1000-order);
        }
        if mode == KEYS or mode == BOTH {
            myKeyMsgSendEvent = get event corresponding to string "keyMsgSend_"&dist_index;
            myKeyMsgReceiveEvent = get event corresponding to string "keyMsgReceive_"&dist_index;
            if (dir == CS and Role == CLIENT) or (dir == SC and Role == SERVER) or (dir == BOTH)
                bind(myKeyMsgSendEvent,Encrypt,order);
            if (dir == CS and Role == SERVER) or (dir == SC and Role == CLIENT) or (dir == BOTH)
                bind(myKeyMsgReceiveEvent,Decrypt,1000-order);
        }
    }
}
```

Figure 3: DESPrivacy micro-protocol.

The design of the SecComm service also allows any combinations of privacy micro-protocols to be used together. While it is often difficult to determine the exact improvement in security achieved by combining more than one encryption method, it has been argued that proper use of multiple encryption can increase privacy [MH81]. Multiple encryptions schemes considered secure, including triple DES, are discussed in [Sch94].

3.3.3 Authenticity and integrity

Authenticity and integrity are discussed together since integrity can be considered a subset of authenticity. That is, a message can truly be considered to be authentic only if there is confidence that it was sent by its claimed sender and that it has not been modified since being transmitted. Authenticity and integrity can be achieved either by encrypting the whole message or by using a message digest generated by a cryptographic message digest or hash function such as MD5 [Riv92] or SHA [SHA95]. If a standard message digest function is used, the digest itself must be protected by either encrypting it or by calculating the digest over the data and a secret key (e.g., keyed MD5 [MS95a] and SHA [MS95b]).

SecComm currently includes two basic message digest micro-protocols, MD5Integrity and SHAIntegrity, and their keyed counterparts, KeyedMD5Integrity and KeyedSHAIntegrity. Each of these micro-protocols creates a message digest as a peer attribute with tag DIGEST at the sender, and checks it at the receiver. If the non-keyed integrity micro-protocols are used, they are executed before the corresponding cryptographic protocol so that the message digest is protected.

Two authenticity micro-protocols based on public keys are also included. RSAAuthenticity encrypts the entire message with the sender's private key, while RSADigestAuthenticity encrypts only the message digest. Figure 4 gives pseudo code for RSAAuthenticity. The RSADigestAuthenticity micro-protocol is very similar except it only encrypts and decrypts the DIGEST attribute. Note that the integrity micro-protocols may be used with RSAAuthenticity. In this case, the whole message including the message digest will be encrypted and decrypted. Note also that in practice, it would make sense to combine the RSAAuthenticity and RSADigestAuthenticity into one micro-protocol with an extra argument indicating if the whole message or just the digest is to be encrypted. In this paper, we consider them to be separate micro-protocols to clarify the presentation and to keep the micro-protocols as simple as possible.

```
micro-protocol RSAAuthenticity(key,order,dir,mode,dist_index) {

    handler Encrypt(msg){
        pack(msg); setAttr(msg,DATA) = RSA(getAttr(msg,DATA),Keys[myKey]);
    }
    handler Decrypt(msg){
        sender = getAttr(msg,SENDER);
        if (publickey = PublicKeys.get(sender) == NULL) {
            raise(keyMiss,sender,SYNC);
            publickey = PublicKeys.get(sender);
        }
        setAttr(msg,DATA)= RSA(getAttr(msg,DATA),publickey); unpack(msg);
        setAttr(msg,AUTH_SENDER,STACK) = sender;
    }
    initial {
        . . . similar to DESPrivacy . . .
    }
}
```

Figure 4: RSAAuthentication micro-protocol.

Unlike methods based on public keys, with shared secrets, one micro-protocol can be used to provide both privacy and authenticity by encrypting the whole message. Similar to RSADigestAuthenticity, however, we can develop variants of the privacy protocols that only encrypt the message digest. Since these micro-protocols do not provide privacy, we consider them exclusively authenticity micro-protocols. Thus, the DES based micro-protocol is called DESDigestAuthenticity.

If message digests are not used, simply decrypting a message at the receiver does not actually prove the authenticity of the message. However, it does guarantee with a high probability that the result will be syntactically and/or semantically incorrect if the message is not authentic. The detection of such an incorrect message can either be left to the higher level protocols or implemented in the SecComm service. One approach to the latter is to introduce redundancy into the message by adding one or more peer message attributes at the sender before the message is encrypted, and then checking the attributes at the receiver after the message is decrypted. Such an additional check is implemented by the ConsistencyCheck micro-protocol.

Although all the micro-protocols presented here are very simple, the overall design of SecComm allows powerful combinations of authenticity and integrity checks. For example, a message digest may be keyed, be protected using multiple encryptions, or both. Also, multiple message digests can be included in a message, where each digest is protected using different combinations of encryption methods.

3.3.4 Replay prevention and non-repudiation

There are numerous methods for preventing accidental or malicious message replay. For example, the sender can add a timestamp to the message that the receiver checks to determine if it is too old. This method is implemented by the TimeReplayPrevention micro-protocol shown in Figure 5. Since this method may miss messages replayed quickly, it could be augmented to maintain a small cache of old messages that is checked when new messages arrive. Note that this micro-protocol sets message attribute STATUS to STOP to indicate that the message should not be delivered to the higher level. All other micro-protocols that check the correctness of a message at the receiver use a similar technique.

Other methods for replay prevention are based on message sequence numbers. Specifically, if the underlying communication provides a FIFO ordering guarantee, it is sufficient to verify that a message's sequence number is larger than the largest sequence number seen so far. This method is implemented by the SeqReplayPrevention micro-protocol. Both of these methods are good for streams of messages flowing in one direction. For handshake style protocols such as key distribution, replay prevention is often implemented by the sender creating a random number that it requires the receiver to return in its reply. Such replay prevention methods are discussed with the key distribution micro-protocols in section 3.3.6.

```
micro-protocol TimeReplayPrevention(order,dir,mode,dist_index) {

    handler AddTimeStamp(msg){
        setAttr(msg,TIMESTAMP,PEER) = time();
    }
    handler CheckTimeStamp(msg){
        if getAttr(msg,TIMESTAMP) < time() - MaxDelay {
            raise(securityAlert,msg,SYNC); stopEvent(); setAttr(msg,STATUS,LOCAL) = STOP;
        }
    }
    initial {
        MaxDelay = maximum network delay + maximum clock drift;
        . . . similar to DESPrivacy . . .
    }
}
```

Figure 5: TimeReplayPrevention micro-protocol.

Note that if a message is not protected from tampering either using integrity or privacy methods, it is easy for an intruder to circumvent replay prevention micro-protocols by simply modifying the time stamp or the sequence number. However, replay prevention can still be useful without these methods to detect accidental message replays.

Non-repudiation is based on public key authentication, with the receiver storing the message encrypted using the sender's private key as a proof of having received the message. In SecComm, the Non-Repudiation micro-protocol simply stores the encrypted message in a file with a timestamp indicating when it was received. To make it easier to decrypt the stored message at a later time, the public key authentication should be the first encryption done at the sender and thus, the last decryption done at the receiver. Non-Repudiation implements this by storing the message just before this last decryption, which means it can be re-decrypted at some later time knowing only the sender's public key at the time the message was received. Otherwise, decryption might require using session keys created just for the particular communication connection.

3.3.5 Key creation micro-protocols

Key creation micro-protocols are used to generate new secret keys for use as session keys for communication connections. The design decouples key creation from key distribution and thus, allows different key creation methods to be easily configured into the system. Such configurability can considerably strengthen the security of the service, since knowing how session keys are created makes it easier to break a cryptosystem.

The structure of a key creation micro-protocol is simple, as illustrated in Figure 6.

```
micro-protocol RandomKeyCreate(creation_index,seed) {

    handler CreateKey(key,length){
        key = generate next random key of size length using previous; previous = key;
    }
    initial {
        myKeyCreateEvent = get event corresponding to string "keyCreate_"&creation_index;
        bind(myKeyCreateEvent,CreateKey); previous = seed;
    }
}
```

Figure 6: RandomKeyCreate micro-protocol.

3.3.6 Key distribution micro-protocols

If the keys used by the secret key based cryptographic methods are not agreed upon *a priori*, they must be distributed and agreed upon at the time a communication session is opened. Among the potential options for basic key creation and distribution are:

- The client creates the key and distributes it to the server(s).
- The server creates the key and distributes it to the client.
- Both the client and server create a key and the session key is calculated from these two keys using some method (e.g., Diffie-Hellman).
- Some external security principal creates the session key and distributes it to the client and server(s) (e.g., Kerberos).

Key distribution has security risks analogous to data communication, but with greater potential impact since the compromised key will likely be used for a period of time. Some key distribution methods such as Kerberos have fixed built-in methods for providing privacy, authenticity, integrity, and replay prevention for the key distribution process. A Kerberos style key distribution scheme can be implemented in our framework as a micro-protocol, but a key objective of our design is to allow the security guarantees required for key distribution to be as customizable as those for normal data communication. This flexibility is provided by implementing the above key distribution strategies as simple micro-protocols that do not by themselves provide any security guarantees except replay prevention. Instead, they raise events keyMsgSend_i and keyMsgReceive_i, which allows any combination of the security property micro-protocols to be configured to secure the key distribution.

Key distribution micro-protocols are executed at session creation time. When a session is opened, all the micro-protocols in the session are first initialized by invoking their initialization section. Then, the

SecComm composite protocol raises event openSession, for which all key distribution micro-protocols are registered. For simplicity, we assume that each key distribution micro-protocol creates and distributes one key in the Keys data structure. Furthermore, the key distribution micro-protocols are designed so that they execute in sequence—i.e., one key is created and distributed before the next one is created—with the key index used as the order of key distribution. This sequential execution is achieved through event manipulation as illustrated in Figure 7. In particular, the first key distribution micro-protocol to be executed stops the openSession event, that is, prevents the other key distribution micro-protocols from being notified of its occurrence. When the distribution of the first key is completed, the key distribution micro-protocol removes its handler binding for the openSession event and raises the openSession event again. The ReplaceKey event handler and replaceKey_i events are for dynamically replacing a session key during a communication connection.

Note that SecComm must not accept new messages from the higher level protocols while key distribution is in progress or keys are being changed. Thus, the composite protocol provides micro-protocols with operations **disablePush()** and **enablePush()** to tell the composite protocol to disable or enable push operations. If multiple micro-protocols disable the push operation, they all must enable it before communication can resume.

Other key distribution micro-protocols such as ServerKeyDistribution and DiffHellKeyDistribution have the same general structure. Key distribution based on an external security principal naturally has a somewhat different structure, but the same general principles apply.

3.3.7 Public key acquisition

The system may contain one or more central certification authorities (CAs) that store public keys of users and other security principals. The public key acquisition micro-protocols communicate with these authorities to retrieve the desired keys. Since there are multiple protocols for acquiring certificates (e.g., X.509, PGP) and multiple, potentially incompatible, implementations of these protocols (e.g., X.509 implementations by Netscape, Microsoft, and RSA), a number of different PublicKeyAcquisition micro-protocols may be required.

A CA would typically not understand the Cactus message format or use the same protocol stack, so these micro-protocols have to use TCP sockets directly, and construct and manipulate messages that conform to the particular protocol and implementation of the CA. If an implementation of the protocol is available in a cryptographic library (e.g., SSL), the micro-protocol implementation can utilize these routines to facilitate the process.

3.3.8 Configuration constraints

As indicated in the descriptions above, constraints sometimes exist between micro-protocols that restrict which combinations are feasible. Figure 8 summarizes these restrictions in a graphical form. Each node represents a micro-protocol and each edge represents a dependency between micro-protocols. A micro-protocol depending on a choice of at least one other micro-protocol is represented by a dotted line box around the set of micro-protocols. For example, the digest based authenticity micro-protocols require that some integrity micro-protocol create the digest. A potential dependency between micro-protocols is represented by a dashed line edge. In particular, all micro-protocols that use keys may require a key distribution micro-protocol if the keys are not predefined.

```
micro-protocol ClientKeyDistribution(key_index, creation_index, length){

    handler DistributeKey() {
        stopEvent(); raise(myKeyCreateEvent,key,myKeyLength,SYNC); Keys[myKey] = key;
        msg = new Message; addAttr(msg,KEY,PEER) = key;
        raise(myKeyMsgSendEvent,msg,SYNC); bind(msgFromBelow,ReceiveAckMsg,0);
        send msg to server;
    }
    handler ReceiveKeyMsg(msg) {
        stopEvent(); unbind(msgFromBelow,ReceiveKeyMsg);
        setAttr(msg,STATUS,LOCAL) = FORWARD; raise(myKeyMsgReceiveEvent,msg,status,SYNC);
        if getAttr(msg,STATUS) == FORWARD { Keys[myKey] = getAttr(msg,KEY);
            raise(myKeyMsgSendEvent,msg,SYNC); send msg to client; enablePush(); }
    }
    handler ReceiveAckMsg(msg) {
        stopEvent(); unbind(msgFromBelow,ReceiveAckMsg);
        setAttr(msg,STATUS,LOCAL) = FORWARD; raise(myKeyMsgReceiveEvent,msg,status,SYNC);
        if getAttr(msg,STATUS) == FORWARD and getAttr(msg,KEY) == Keys[myKey]{
            unbind(openSession,DistributeKey); raise(openSession,ASYNC); enablePush(); }
    }
    handler ReplaceKey(mode) {
        if mode == CLIENT DistributeKey();
        else bind(msgFromBelow,ReceiveKeyMsg,myKey);
    }
    initial {
        myKey = key_index; myKeyLength = length; disablePush();
        if Role == CLIENT bind(openSession,DistributeKey,myKey);
        else bind(msgFromBelow,ReceiveKeyMsg,myKey);
        myKeyCreateEvent = get event corresponding to string "keyCreate_"&creation_index;
        myKeyMsgSendEvent = get event corresponding to string "keyMsgSend_"&myKey;
        myKeyMsgReceiveEvent = get event corresponding to string "keyMsgReceive_"&myKey;
        myReplaceKeyEvent = get event corresponding to string "replaceKey_"&myKey;
        bind(myReplaceKeyEvent,ReplaceKey);
    }
}
```

Figure 7: ClientKeyDistribution micro-protocol.

This configuration graph illustrates which combinations of micro-protocols result in valid services. At the time the protocol graph is constructed, a two step process is followed. First, those micro-protocols that enforce the desired security attributes using the desired techniques are selected. Then, the transitive closure of those micro-protocols is taken within the graph to satisfy dependencies. A graphical configuration tool called the CactusBuilder has been constructed that automates this process and generates the appropriate configuration files [Hil98].

4 Related Work

The need for customizing security guarantees for data communication has been noted in the development of the IPSec protocol [KA98], as well as SSL [FKK96] and TLS [DA99]. As noted in the Introduction, IPSec support two security options, the AH (authentication header) option and the ESP (encapsulating security payload) option. The security option and the specific cryptographic algorithms and keys used for a connection are specified in a *security association* (SA). A SA may be created manually or negotiated automatically

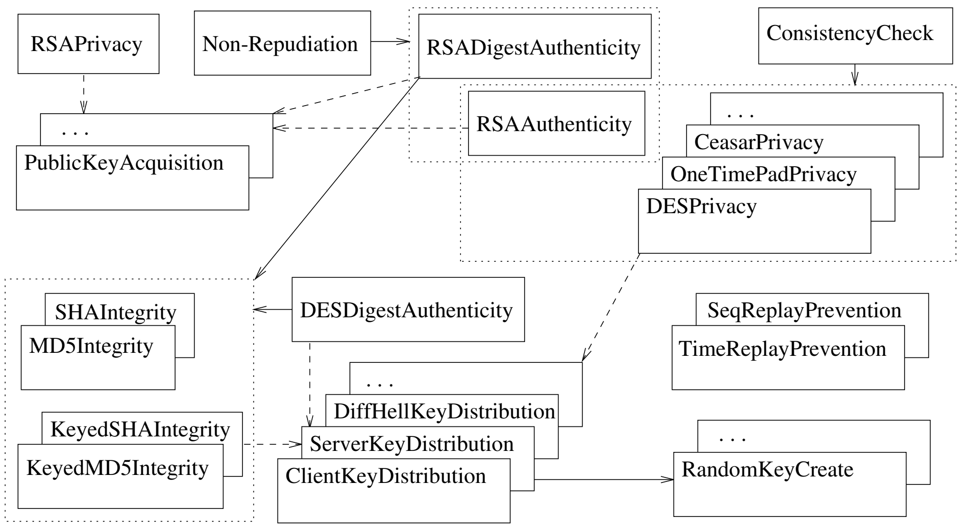

Figure 8: Configuration constraints.

using key management protocols such as IKE [HC98]. Multiple security methods may be specified for a connection by giving multiple SAs, called a *SA bundle* [KA98].

TLS also provides customizable security guarantees for communication between processes. The TLS protocol consists of two layers. The lower level protocol, the TLS Record Protocol, provides privacy using encryption (e.g., DES or RC4) and integrity using keyed message digests (e.g., SHA or MD5). Messages can also be compressed using a chosen compression algorithm. A higher level TLS Handshake Protocol is used to authenticate the communicating partners and negotiate cryptographic algorithms and keys for the TLS Record Protocol. The handshake protocol typically uses X.509 certificates to authenticate the server and potentially the client. A number of key exchange options are supported, including RSA and Diffie-Hellman. A similar type of customization is provided in other Internet protocol proposals, including the Secure HyperText Transfer Protocol (S-HTTP) [RS98] and Privacy-Enhanced Mail (PEM) [Lin93].

IPSec, TLS, and the SecComm approach presented here have a similar goal of customizable secure communication, but with different constraints on the solution and different approaches for achieving this goal. For example, TLS is designed for communication between a client and server that do not have any prior agreement on security configuration. It also does not directly support multiple encryptions or multiple message digests, nor does it provide non-repudiation. In addition, both IPSec and TLS are optimized for the case where a connection is used to send a large number of messages rather than single messages, which means that good performance is “hard-wired” to be a high priority rather than something that can be customized in the context of the performance/security tradeoff. Moreover, although IPSec is relatively flexible, the approach presented in this paper offers a simpler design—for example, no need for separate AH and ESP options and SA bundles—with unlimited flexibility in combining security techniques. Similarly, it appears that our design easily surpasses the flexibility of TLS.

Finally, note that IPSec and TLS are protocol specifications and therefore do not specify how an IPSec or TLS compliant service must be implemented internally. Our approach could be used to configure, in essence, an instance of SecComm that is IPSec or TLS compliant. Of course, the message packing routines would have to be customized to generate IPSec and TLS compliant message formats, but this is easily done using the customization facilities provided by the Cactus runtime system. Although IPSec and TLS can naturally be implemented in a modular manner without the Cactus approach, the benefits of the approach become more prominent when arbitrary combinations of methods are desired, when key distribution must be customizable, and, in particular, if the security mechanisms must change adaptively at runtime.

Communication security has been addressed in a number of other prototype systems as well. For example, the Rampart [RBR94] and SecureRing [KMMS98] systems use public key authentication to tolerate Byzantine failures. The Ensemble system [RBH$^+$98] provides a degree of customization for secure group communication. In particular, Ensemble uses a model where the communication subsystem is constructed as a hierarchy of modules, with security provided by a number of optional modules. The *Signing Router* module adds a keyed MD5 signature to messages, and the *Encrypt* module provides privacy using RC4. Both modules use the same agreed group key, which is established by the *Exchange* and *Rekey* modules using PGP for authentication. Other cryptographic methods could easily be substituted for RC4, MD5, and PGP. The potential level of security customization using Ensemble is comparable to IPSec, but not comparable to that provided by SecComm.

5 Conclusions

The ability to customize security attributes at a fine-grain level allows users to pick the most appropriate point along the security spectrum based on the characteristics of their particular application and security environment. SecComm is a security service designed to support this type of customization for the communication needs of networked applications. While similar in spirit to existing protocols such as IPSec and TLS, SecComm goes beyond these to support more attributes and more variants, all within a general flexible and extensible implementation framework based on micro-protocols and events. The design also decouples to a large extent the security aspects and the communication aspects of the problem. This allows, for example, SecComm to be used with multiple transport protocols and at multiple locations in the protocol hierarchy with little or no change.

SecComm is currently being implemented using the Cactus system on a cluster of Pentiums running Mach MK 7.3. Once completed, we will experiment with the service in the context of various applications, including a configurable distributed file system that is also being built using Cactusİn addition, we will explore altering security attributes and techniques within the service dynamically using the adaptive facilities provided by CactusÓur ultimate goal is to use this fine-grain configurability and fast adaptation ability as the basis for an *inherently survivable system architecture* that can automatically react to threats in the execution environment [HSU98].

Acknowledgments

Gary Wong implemented the Cactus framework used for the SecComm implementation and ported the Cryptlib to the MK 7.3 platform. Gary and Carlos Ugarte provided excellent comments and suggestions that improved the paper.

References

[BHSC98] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, Nov 1998.

[DA99] T. Dierks and C. Allen. The TLS protocol, version 1.0. Request for Comments (Standards Track) RFC 2246, Certicom, Jan 1999.

[DES81] *DES Modes of Operation*. National Bureau of Standards, FIPS PUB 81, U.S. Department of Commerce, Washington, D.C., 1981.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[FKK96] A. Freier, P. Karlton, and P. Kocher. The SSL protocol, version 3.0. Internet-draft, Netscape Communications, Nov 1996.

[Gut98] P. Gutmann. Cryptlib. http://www.cs.auckland.ac.nz/∼pgut001/cryptlib/, 1998.

[HC98] D. Harkins and D. Carrel. The internet key exchange (IKE). Request for Comments (Standards Track) RFC 2409, Cisco Systems, Nov 1998.

[Hel78] M. Hellman. An overview of public-key cryptography. *IEEE Transactions on Communications*, 16(6):24–32, Nov 1978.

[Hil98] M. Hiltunen. Configuration management for highly-customizable software. *IEE Proceedings: Software*, 145(5):180–188, Oct 1998.

[HP91] N. Hutchinson and L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[HS95] M. Hiltunen and R. Schlichting. Constructing a configurable group RPC service. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 288–295, Vancouver, BC, Canada, May 1995.

[HS98] M. Hiltunen and R. Schlichting. A configurable membership service. *IEEE Transactions on Computers*, 47(5):573–586, May 1998.

[HSH$^+$98] M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. Technical Report 98-06, Department of Computer Science, University of Arizona, Tucson, AZ, Jul 1998. To appear in *IEEE Transactions on Parallel and Distributed Systems*.

[HSU98] M. Hiltunen, R. Schlichting, and C. Ugarte. Survivability issues in Cactus. In *Proceedings of the 1998 Information Survivability Workshop*, pages 85–88, Orlando, FL, Oct 1998.

[JJ98] N. Johnson and S. Jajodia. Exploring steganography: Seeing the unseen. *IEEE Computer*, 31(2):26i–34, Feb 1998.

[KA98] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments (Standards Track) RFC 2401, BBN Corp, Home Network, Nov 1998.

[KMMS98] K. Kihlstrom, L. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st Hawaii International Conference on System Sciences*, pages 317–326, Kona, HI, Jan 1998.

[Lin93] J. Linn. Privacy enhancement for internet electronic mail: Part I: Message encryption and authentication procedures. Request for Comments RFC 1421, Feb 1993.

[MH81] R. Merkle and M. Hellman. On the security of multiple encryption. *Communications of the ACM*, 24(7):465–467, Jul 1981.

[MS95a] P. Metzger and W. Simpson. IP authentication using keyed MD5. Request for Comments RFC 1828, Piermont, Daydreamer, Aug 1995.

[MS95b] P. Metzger and W. Simpson. IP authentication using keyed SHA. Request for Comments RFC 1852, Piermont, Daydreamer, Sep 1995.

[NT94] B. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, Sep 1994.

[RBH$^+$98] O. Rodeh, K. Birman, M. Hayden, Z. Xiao, and D. Dolev. The architecture and performance of security protocols in the Ensemble group communication system. Technical Report TR98-1703, Department of Computer Science, Cornell University, Dec 1998.

[RBR94] M. Reiter, K. Birman, and R. van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, Nov 1994.

[Rey95] F. Reynolds. The OSF real-time micro-kernel. Technical report, OSF Research Institute, 1995.

[Riv92] R. Rivest. The MD5 message-digest algorithm. Request for Comments RFC 1321, MIT and RSA Data Security, Inc., Apr 1992.

[RS98] E. Rescorla and A. Schiffman. The secure hypertext transfer protocol. Internet-draft, Terisa Systems, Inc., Jun 1998.

[Sch94] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., New York, 1994.

[SHA95] *Secure Hash Standard*. National Institute of Standards and Technology, U.S. Department of Commerce, Washington, D.C., Apr 1995.

[SNS88] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Conference Proceedings*, pages 191–202, Dallas, TX, Winter 1988.