Secure Communication Using Remote Procedure Calls

ANDREW D. BIRRELL
Xerox Corporation

Research on encryption-based secure communication protocols has reached a stage where it is feasible to construct end-to-end secure protocols. The design of such a protocol, built as part of a remote procedure call package, is described. The security abstraction presented to users of the package, the authentication mechanisms, and the protocol for encrypting and verifying remote calls are also described.

CR Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; D.4.6 [**Operating Systems**]: Security and Protection—*cryptographic controls*

General Terms: Design, Experimentation, Security

Additional Key Words and Phrases: Remote procedure calls, transport layer protocols.

1. INTRODUCTION

Many computing environments now exist in which frequent and substantial parts of the activities involve communication among computers linked by open networks. A user may well spend most of his time at a personal computer and use networks for transferring data to and from other personal computers or shared-server computers such as printers, file servers, and mail servers. Most of the networks (and internetworks) used for these activities are *open* in the sense that they are readily vulnerable to eavesdropping and interference from unauthorized *intruders*. Such an architecture presents security problems much different from the ones traditionally faced in monolithic time-sharing systems. In particular, it is clear that security must be based on the use of encryption in the communication protocols. Fundamentally, encryption permits the establishment of a data channel that is less open than the underlying internetwork, by arranging that only authorized parties can create, inspect, and/or modify some or all of the data. Establishing, using, and maintaining such a secure data channel requires the resolution of multiple problems. First, it is necessary to identify the authorized parties (traditionally called *principals*). Second, it is necessary to convince each principal that the others are indeed who they claim to be. (This step is tradition-

Author's present address: Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301.

ally termed *authentication*.) Third, it is necessary to transfer the actual data in a manner that is not vulnerable to various known threats. The second and third of these are inevitably interdependent, since a recipient may require convincing that each particular datum did indeed come from the asserted sender.

There are several discussions in the public literature about designing communication protocols to achieve various forms and levels of security. Much of the published material is concerned with particular aspects of the overall problem, such as the design and improvement of authentication protocols [1, 4, 10]. There is less material available describing how to construct a complete secure communication protocol. A recent report by Voydock and Kent [11] gives a thorough description of one such design, including substantial description of the supporting arguments for their design. There are disappointingly few real implementations of secure protocols. The purpose of this paper is to describe the construction of such a protocol.

It is possible to include secure communication at various levels in a communication protocol hierarchy. At the physical layer, security can be achieved by various noncryptographic techniques that prevent tampering with the communication medium itself. At the data link layer, it is possible to encrypt all traffic on each link using a code whose key is shared among all nodes directly connected to that link. This is termed *link* encryption; it protects against intruders from outside the community that shares that data link, but does not distinguish principals within that community. When a communication path is formed from a network consisting of multiple data links, link encryption allows intrusion by members of the trusted community of *every* data link traversed by the path. The lowest layer at which we can provide an end-to-end guarantee is the network layer, where we introduce direct node-to-node addressing of packets. But in most communication architectures (including ours) it is not until the transport layer that end-to-end security is feasible. The transport layer is the lowest level at which enough state information is kept to establish the authenticity of incoming data in successive packets of an interaction. It is the transport layer where we are first concerned with the relationship between successive packets. Here we introduce code that handles packet sequencing, detects missing or repeated packets, and retransmits to recover from lost or malformed packets. These mechanisms are similar to those needed to implement a secure protocol, and so we have chosen to introduce secure communication as an aspect of the transport layer protocol.

One could also introduce security facilities at higher levels. However, doing so would reproduce many of the checking mechanisms already present in the transport layer. These mechanisms have always been difficult to design and implement and often significantly reduce efficiency. Implementing them twice seems undesirable. It would also reduce the utility of the secure protocol, since the easiest way to communicate would likely be by using the transport layer directly. This is particularly true of remote procedure calls, where a major purpose is to simplify the task of communicating by providing a single simple and widely shared mechanism: procedure calls. If security were something that required extra programming beyond the procedure invocation itself, then it would intrude on the aim of easy communication.

Large parts of our security design are derived from previous work. A moderate understanding of previous work is needed for proper appreciation of the remainder of this paper; the report by Voydock and Kent [11, 12] is a good introduction. As discussed in Section 3, we use the federal Data Encryption Standard (DES) for our encryption [8]. This choice is dictated largely by the availability of very fast (and cheap) hardware for DES. Hence, our schemes are based on the use of *private keys* (instead of public keys [6]). For our purposes it would be impracticable to have each pair of principals that want to communicate share a private key, so our scheme is based on the use of an *authentication service* (also known as a *key distribution center*). Thus each principal has a single private key known only to the principal and the authentication service. When two principals wish to communicate, they negotiate with the authentication service to obtain a shared *conversation key.* This conversation key is used to encrypt subsequent communication between the two principals.

The design presented here arose as part of a project to implement remote procedure calls (RPC) on the Xerox research internetwork. The overall design of this RPC package has been reported in [3]. Prior to the construction of this RPC package, there were no encryption-based protocols in the internetwork. Previous protocols transmitted passwords as clear text whenever any authentication was desired. Part of the design of this RPC package included a new transport layer protocol, and this seemed like an ideal opportunity to include security features at the correct level in the protocol hierarchy. An additional factor that enabled a secure protocol to be introduced was that most software using the research internetwork had recently converted to using Grapevine [2] as the primary authority for naming and authenticating individuals and services. This allowed us to envisage using Grapevine as the mediator in the negotiation to establish the authenticity of the principals involved in secure communication.

2. THE SECURITY ABSTRACTION OFFERED TO CLIENTS

Clients of our RPC package interface to its security facilities by dealing in *conversations.* A conversation represents a communicating pair of security principals; during secure communication, one of these principals is an implementor of a remote procedure, and the other is a caller on that procedure. A client can create a conversation by presenting the RPC run-time system with his name and private key and the name of the other principal. Subsequently, if that conversation is an argument of a remote procedure call, the RPC run-time system ensures that the call is performed securely using a conversation key known only to those two principals. We guarantee to the caller and callee that they are the two principals nominated when the conversation was created. (More precisely, we guarantee that the caller and callee are each trusted by one of those principals, to the extent of having been told the conversation key by one of them.) When a server is invoked for an incoming call with a conversation as argument, the server may ask the RPC run-time system for the name of the other principal in the conversation. Thus, our clients never deal explicitly with encryption, but they get the appropriate guarantees. Creating a conversation involves an interaction between the principal who wishes to create it and the authentication service (as described in Section 4). Using a conversation to make a remote call (described

in Section 5) involves only the two principals—the authentication service is not concerned with this.

For example, if principal *A* wishes to communicate securely with principal *B*, then *A*'s program would include a call on the RPC run-time system of the form:

```
conv ← R.P.C. Create Conv [from: nameOfA, to: nameOfB, key: privateKeyOfA]
```

Principal *A* could then make remote calls to a procedure P.Q implemented by *B* such as:

```
x ← P.Q[thisConv: conv, arg: y]
```

Inside the implementation of P.Q, principal *B* could find the identity of his caller by a call on the RPC run-time system of the form:

```
caller ← RPC.GetCaller[thisConv]
```

This concept of conversations is orthogonal to the other abstractions involved in a call. Multiple processes can participate in a single conversation; there may be multiple simultaneous calls in a conversation; calls can be made through multiple remote interfaces but still be part of the same conversation. Calls may be made in either direction in a conversation, independent of which principal is the caller and which is the callee. Indeed, it would be consistent for many machines (with the same two principals) to participate in a single conversation, although we have not implemented this.

Note that we restrict a secure conversation to a *pair* of principals. We do not directly support multiparty conversations, although they may be emulated by pairwise two-party conversations. Nor do we support third party operations. For example, if a user *A* calls a server *B* to perform some operation, the server *B* cannot communicate securely with a third principal *C* (on a third machine) to perform some action on behalf of *A* merely by providing the authentication information that *B* obtained from *A*. To support such interaction, it would be necessary for *A* to establish a conversation between himself and *C*, then give *B* enough information (particularly, the authenticator and conversation key) to allow *B* to participate in the conversation. Such interactions can be made securely and are not ruled out by our package, but we provide little aid for them.

When building a secure system of any sort, it is important to be clear about the threats that are being countered. We guarantee to the caller that his call will be performed only by a callee whose name the caller has nominated. We will tell the callee the true name of the caller. Calls cannot be observed in transit, to the extent that an intruder cannot determine which procedure is being called, nor any information about the arguments or results (except their length). An intruder cannot make undetected modifications to calls and results while they are in transit. An intruder cannot cause the invocation (or replay) of a call. We do not attempt any protection against traffic analysis or against denial of service (although clearly a caller will notice if his remote call does not complete because of a denial of service attack). It is also important to be aware that these guarantees

are not absolute. The best that can be offered is that we make it prohibitively expensive for an intruder to violate these guarantees. The aim is to make that expense greater than the value to the intruder. The communicating principals should trust these security guarantees only to the extent that they trust the authentication service, the encryption algorithm, and each other.

3. ENCRYPTION ALGORITHMS

As mentioned in Section 1, we use the federal Data Encryption Standard (DES) for our encryption. We made this choice primarily because DES has been implemented in cheap, fast hardware. (The fastest chips run at about 14 megabits per second.) There has been some controversy over the cryptographic strengths and weaknesses of DES, but these are not important to our design. The design would be unaffected by a choice of any other private-key encryption algorithm. Our detailed packet formats allow for multiple encryption algorithms and for key lengths up to 128 bits. Use of a public-key system [6] would have a large impact on the authentication protocol.

Basically, DES maps 64-bit blocks of *plain text* into 64-bit blocks of *cipher text.* That basic mapping hides the data but does not hide patterns (such as repeated blocks of zeros) and does not detect modifications. The *cipher block chaining*, or CBC, mode of DES [9] hides the patterns but still does not guarantee that modifications will be detected. We use the CBC mode with the addition of a 64-bit checksum encrypted at the end of the packet. This checksum is formed by accumulating the 64-bit exclusive-or of the plain text blocks (this is performed by hardware in parallel with the encryption). This technique reduces the probability of most undetected modifications to 2^{-64}. This assertion is based on the observation that from the point of view of an intruder who does not know the conversation key, modifying a block of cipher text produces an unpredictable modification to two blocks of plain text when the cipher text is decrypted. It is fairly simple to show that a random modification to 64-bits of plain-text has probability $1 - 2^{-64}$ of changing the resulting checksum. An alternative modification to CBC mode, which we rejected because the requisite extra hardware would be more complicated, has been proposed by Ehrsam et al. [5]. Remember that an intruder has an a priori probability of 2^{-56} of guessing the conversation key at his first attempt.

Unfortunately, Voydock and Kent have recently pointed out that both of these schemes for detecting modifications to cipher text are inadequate [12]. If an intruder swaps two adjacent cipher text blocks, the change might not be detected. We have not yet modified our protocols to repair this defect.

We assume that users choose (or are issued) sufficiently random private keys [7]. Temporary keys, CBC initialization vectors, and conversation keys should be generated by the authentication servers using a hardware random number generator.

In the descriptions in the following sections, we have omitted some details. These details are quite systematic, being the modifications needed for secure distribution of CBC initialization vectors, for avoidance of transmissions of cipher text for known plain text, and for minimizing the amount of data encrypted with long term keys. All these details are given in full in Section 8.

We use the notation $\{P\}^K$ to indicate the cipher text formed by encrypting

plain text P using encryption key K. In each context, if P is an encryption key, we intend straightforward single-block use of DES, and otherwise we intend encryption using the CBC mode with the checksum described above.

4. AUTHENTICATION

There is substantial literature on protocols for implementing this negotiation [1, 4, 10]. The protocol we use is based primarily on Needham and Schroeder's [10], modified slightly to improve some shortcomings and rearranged to meet our efficiency goals.

This protocol relies on the presence of a trusted authentication service. We use the Grapevine distributed system [2] as our authentication service. Grapevine provides a distributed replicated database indexed by strings known as *RNames.* Several values may be associated with an RName. One such value is used as the private key for security principals.

Our authentication scheme creates an *authenticator.* An authenticator is encrypted data that one principal can use to assure the other of his identity. When principal A passes an authenticator to B, the assurance is based on B's observation that someone who knew B's private key (namely, the authentication service) promises that the imbedded conversation key was given only to principal A. B may as well believe the assurance, because the only alternative is that B's private key has been compromised. The authenticator takes the form

$$\{CK, T, A\}^{KB},$$

where CK is a conversation key, A is A's name, T is the time at which the authenticator was created, and KB is B's private key. T is used to limit the damage potentially caused by a compromised private key, by limiting the lifetime of an authenticator to a few hours.

To obtain an authenticator, A calls the procedure RPC.CreateConv provided by the RPC run-time system on A's host, giving it A's name, B's name, and A's private key. The RPC run-time system calls the authentication service remotely (without additional encryption) giving it

$$[A, B, X],$$

where A and B are the principal names and X is a nonrepeating 64-bit number. (Alternatively, X may be chosen pseudorandomly or randomly.) The authentication service returns

$$\{\text{authenticator}, X, B, CK\}^{KA},$$

where KA is A's private key and CK is the conversation key, also imbedded in the authenticator.

The RPC run-time system on A's host may now obtain the conversation key and authenticator and is assured that it and CK were issued by the authentication service for communication between A and B. Additionally, the RPC run-time system generates a permanently unique identifier for the conversation. Later, when A asks the RPC run-time system to make a remote call using this conversation, it has available the authenticator, the conversation key, and the

unique identifier. In Section 5, the first part of Figure 2 shows the operation of creating a conversation.

The permanently unique identifier of a conversation is created by concatenating the unique identifier of this processor with a sequence number. When the run-time system is first started, this sequence number is initialized from a one-second real-time clock, and the values used for unique identifiers never exceed the current value of that clock. This restricts the rate of generation of new conversations on a single processor to a long-term average of one per second, although the burst rate may occasionally exceed one per second.

Note that in order to return the authenticator, the authentication service uses the private keys of both principals. Since the Grapevine database is distributed, both of these keys might not be known by any single Grapevine host. So to respond to the request a Grapevine host may need to communicate in a secure fashion with another Grapevine host. The Grapevine servers are capable of communicating securely among themselves, since they are themselves security principals registered in a part of the database that is replicated on *every* Grapevine host.

It is important to remember that the entire security of this scheme depends on the security of the authentication service's database. Ultimately, this must depend on the physical protection of the hosts maintaining this database.

5. MAKING SECURE CALLS

The structure employed for our RPC package (as we have described in [3]) is as follows. A caller initiates a remote call by making a local call to a specially constructed *user stub* module. This stub takes the arguments of the call and an identification of the desired procedure and places them in one or more packets that it passes to the RPC run-time system. The run-time system is responsible for transmitting the packets reliably to the remote host and waiting for a response. In the remote host, the packets are received and are passed to the appropriate *server stub* module (also specially constructed). The server stub unpacks the arguments and makes an ordinary local call to the appropriate procedure. When this local call returns, the server stub takes the results and places them in one or more packets, and the RPC run-time system communicates them to the caller machine, where they are given to the user stub. The user stub then takes the results and returns from the original local call. This structure is depicted in Figure 1, and is described in much more detail in [3]. This earlier paper also describes our binding mechanism, whereby a caller determines which host implements a desired remote procedure.

The stub modules are generated mechanically by a program known as *Lupine*. This program gives special treatment to procedures that have an argument whose data type is that used for conversations. In the user stub generated for such a procedure, the code for transmitting the call packets passes the conversation in to the RPC run-time system when asking for the packets to be transmitted. Thus the RPC run-time system knows to encrypt the packets and has access to the appropriate information to do so. Similarly, the RPC run-time system tells the server stub for such a call which conversation is being used, so it is passed as an

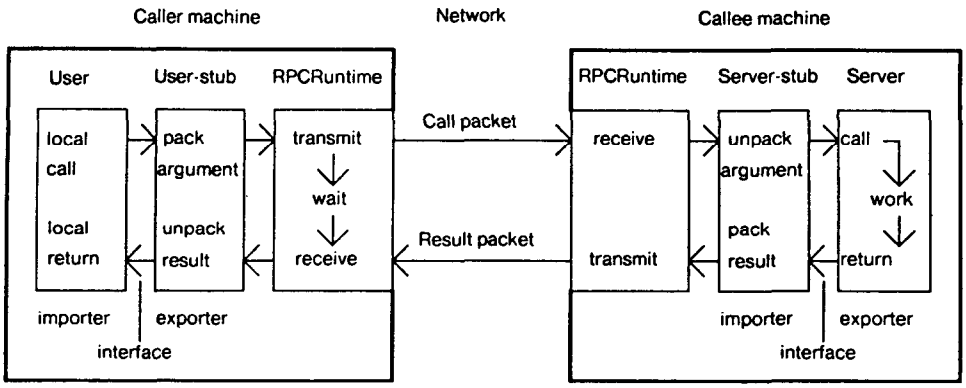

Fig. 1. The components of the system and their interactions for a simple, nonsecure call.

argument to the server implementation of the procedure and can be used to obtain the name of the principal who is the caller.

When making a secure call, the RPC run-time system must take extra steps to encrypt the call; this is easy, since the conversation passed as argument is, in fact, a pointer into the RPC run-time system's data structure, giving it the conversation key and conversation identifier. When receiving such a call, the RPC run-time system must ensure that it has (or obtains) the information about that particular conversation. To describe how this is achieved, we will describe first the steady state, and then we will describe how we reach that state.

To support secure calls, the RPC run-time system on each machine maintains a hash table mapping the unique identifier of a conversation into a data object giving the principal's names and the conversation key. Remember that this identifier is unique over all hosts and all time. To allow secure calls, we have added a field to our packet headers to contain (as plain text) the conversation identifier. Other plain text fields are the internetwork source and destination host identifiers and an identifier of the calling process. The remainder of each packet is encrypted using CBC with the checksum described in Section 3. The encrypted part of the packet contains additional protocol information, particularly the sequence number of this call (relative to the calling host and process), the identifier of the calling process, and the sequence number of this packet relative to this call. We use the term *call identifier* for the set of fields

[calling-host, calling process, call-sequence-number].

On receipt of a packet participating in a secure call, the conversation identifier is looked up in this hash table to find the conversation key, and the remainder of the packet is decrypted. The decrypted packet is checked to ensure that the CBC checksum is valid. Failure of this check indicates that the packet is not genuine. We do not distinguish between errors caused by an intruder and transmission, although this could be done quite easily by adding another checksum layer around the entire packet. (In practice, transmission errors are quite rare at this level in our communication environment, so the occurrence of frequent errors would be cause for investigation.) The RPC run-time system can then consider the remainder of the packet to determine whether the packet starts

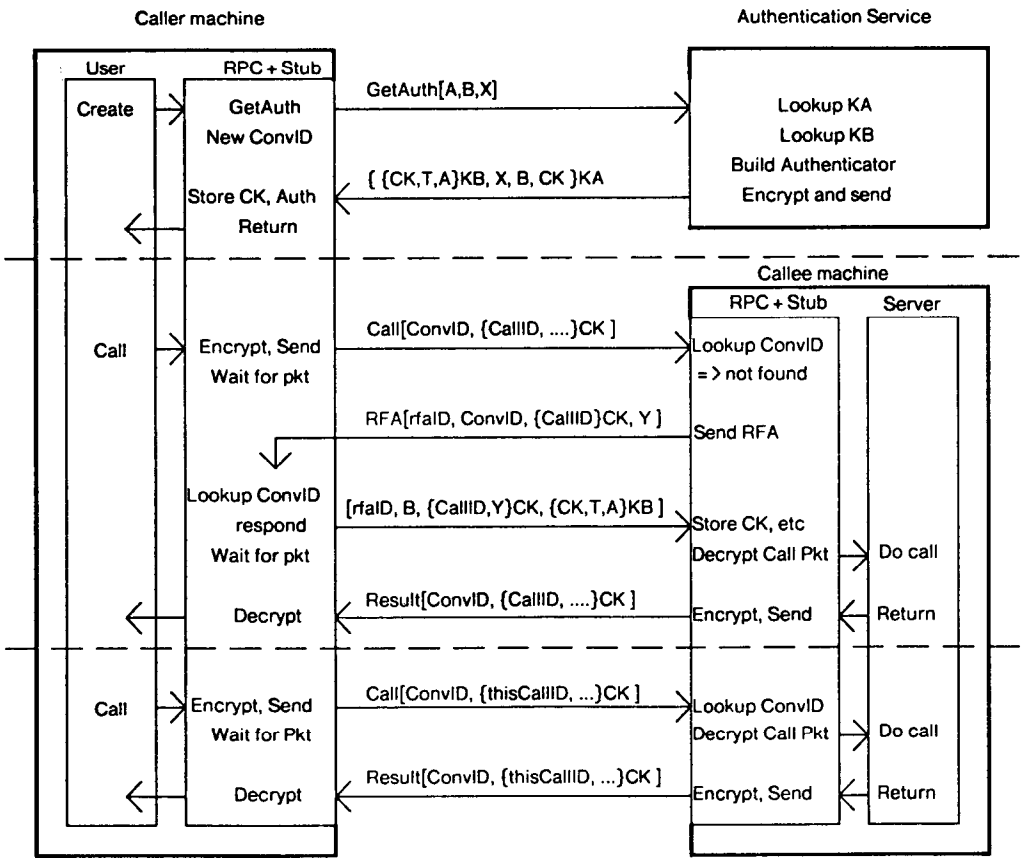

Fig. 2. Communication for secure calls. The first part of the figure shows the creation of a conversation and its corresponding authenticator (*auth*) and conversation identifier (*ConvID*). The last part shows a normal secure call. The middle part shows the interactions when the RFA protocol is needed for the callee to establish or re-establish his connection state.

a new call, is part of an ongoing call, or is a duplicate of some old call. This determination is just as it would be for nonsecure calls and uses the same data structures. Note that we are using the mechanisms for eliminating duplicates that may happen in any transport protocol to simultaneously eliminate replays maliciously injected by an intruder. (This is discussed further in Section 6.) A typical secure call is shown at the end of Figure 2.

We have assumed the existence in the server of a mapping from conversation identifier to caller and conversation key. Further, the table used by the server to eliminate duplicate calls (which gives the sequence number of the last call from each process on each host) is part of the security arrangements, since it is this that prevents an intruder replaying an old call. Clearly, these mappings must be established initially in a secure way, by some form of connection establishment protocol. In our package, this is achieved by a technique that also permits the server to discard this information when the connection is idle. This is achieved by a form of call-back, known as a *request for authenticator*, or RFA. When a server receives a call packet whose conversation identifier is unknown to the

server (and which, therefore, the server is unable to decrypt), the server sends an RFA packet back to the calling machine. The RFA packet contains

$$[\text{conversation-identifier}, \{\text{call-identifier}\}^{CK}, Y],$$

where Y is a 64-bit number chosen by B such that this value of Y is not predictable by an intruder. For example, B could generate Y by using DES as a random number generator with a seed known only to B. Since the server does not yet know the conversation key, it cannot perform any encryptions or decryptions yet, but $\{\text{call-identifier}\}^{CK}$ is available to the server from the initial part of the (still encrypted) packet. (This encryption is defined to use purely CBC, with no checksum.) On receiving the RFA, the caller (who is A) returns a packet containing

$$[B, \{\text{call-identifier}, Y\}^{CK}, \text{authenticator}].$$

This allows the server B, to obtain the conversation key and A's name from the authenticator. The call identifier in the response assures the server of the current call sequence number for the calling process contained in the call-identifier. The server decrypts the original call packet and verifies that its call identifier matches that in the RFA response. The number Y bound in with the call identifier in the RFA response assures the server that the RFA response is not a retransmission and hence that the call is not being replayed by an intruder. The inclusion of B's name in this response is purely to enable the RPC run-time system to decrypt the authenticator without consulting higher-level software; an incorrect name here would cause the packet decryption to fail. The use of this RFA protocol for the first call of a conversation is shown in the middle of Figure 2.

The server now has the information it needs for accepting steady-state calls. This has cost two extra packets, which is minimal for any form of connection establishment. The server may discard this information after a suitable period with no calls from A, since the information can be obtained by the server whenever it wishes. Thus we do not require a connection termination protocol.

Additionally, the server should limit the lifetime of the conversation by using the time given in the authenticator to reduce the damage caused by a compromised private key or conversation key.

6. PREVENTION OF REPLAYED CALLS

The most complicated of the threats we are preventing is that we do not let an intruder cause the server to invoke a call more than once. Basically, this is achieved by the mechanism that eliminates duplicate calls on the basis of the securely transmitted call identifier. This mechanism is initialized securely (and restored securely if the server discards it) by the RFA mechanism. However, there are several subtleties in this.

Note that we specified the *permanent* uniqueness of the conversation identifier. If somehow an intruder caused a principal to reuse a conversation identifier from some previous conversation, the only possible adverse effect with probability more than 2^{-56} is denial of service. This would happen if the server's table still contained an entry for that conversation identifier: in that case, the server would incorrectly believe that it knew the conversation key, and so the checksum would

look wrong when the caller's packets were decrypted. A replayed call would be accepted only if the conversation identifier was reused with the same conversation key. Since conversation keys are allocated securely and randomly on the caller's behalf by the authentication service, the replay is accepted with probability 2^{-56}.

A similar argument applies to the identifier of the calling machine. We used this when looking up the conversation identifier in the server's table, and we did so by believing information transmitted unencrypted in the packet header. The only effect of the intruder modifying the packet header would be that we would decrypt the packet with the wrong conversation key (except for a 2^{-56} probability), and this would be detected by our checksum arrangements. Because of this, we can optimize by not including the caller's machine identifier in the secure part of the packet at all.

We also are relying on the uniqueness of the call identifiers. Each call identifier has three parts: a machine-relative monotonic sequence number, a machine-relative process identifier, and a global machine identifier. The sequence number does not need to be *permanently* unique, since, for a call to be considered at all, its permanently unique conversation identifier must be approved by the caller responding to the RFA. However, within a conversation, the sequence number must be nonrepeating: since we use a 32-bit field this limits us to 2^{32} calls per conversation. Other security considerations restrict the reasonable lifetime of a conversation to less than this. The caller can straightforwardly ensure the machine-relative (nonpermanent) uniqueness of the process identifier. The machine identifier is not transmitted with the call identifier, since it may be picked up from the packet header and verified while looking up the conversation identifier. Again, the possibility of an intruder causing a caller to use a duplicate machine identifier is not a problem (beyond the 2^{-56} probability of identical keys), since it would cause the server to decrypt the packet using the wrong conversation key.

The period for which a server maintains the connection-state information must be guaranteed to be longer than the maximum length of time for which A is willing to continue retransmitting a call packet. Otherwise, an intruder could wait until a call has been invoked, suppress all further packets between B and A, wait until B has discarded the state information, and then allow a retransmission and subsequent packets (including the RFA) to get through. This would have the effect of causing the call to be invoked twice. This remarkably unlikely event is depicted in Figure 3. It is prevented by the server keeping its state information for a long enough period. Note that this only requires clocks that run at approximately the same rate, not synchronized clocks.

7. COSTS OF SECURITY

When embarking on this project, we had naively believed that the major cost of including security facilities would be the encryption itself. In practice, this is far from true. The hardware we plan to use for DES performs encryption at up to 14 megabits per second (faster than any of our communication networks), but making our protocols secure has added significant complexity and cost to them. The discussion of the prevention of replayed calls showed how the amount of information needed to identify a packet is increased by the security requirements.

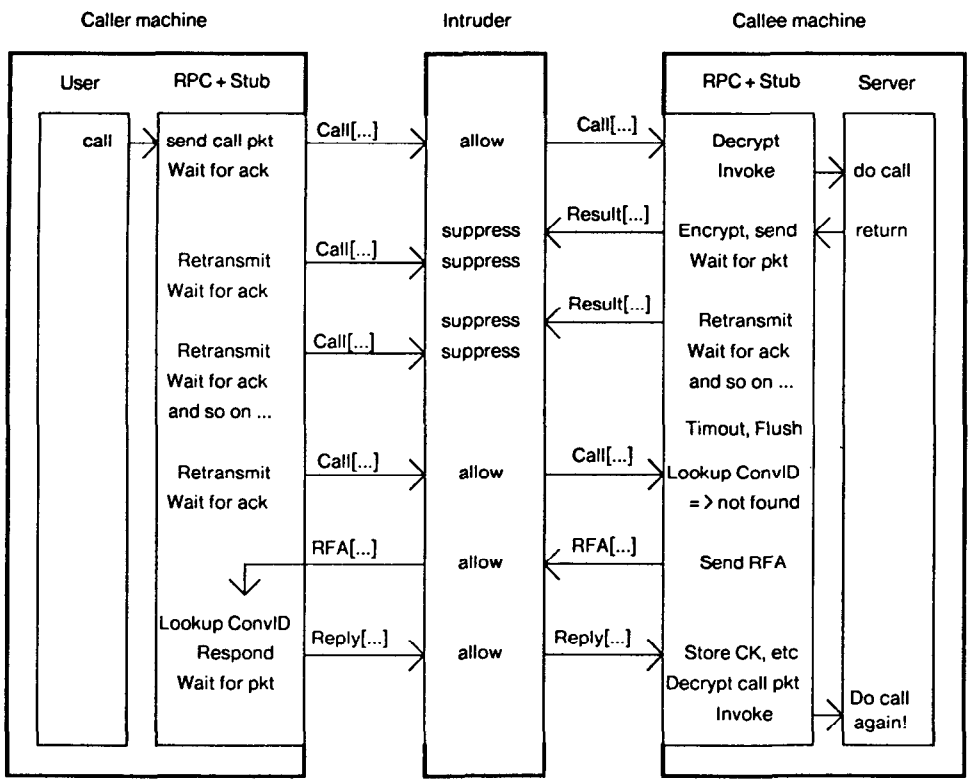

Fig. 3. A security loophole if a server discards its connection state too early. Merely by suppressing appropriate packets, the intruder could cause a call to be invoked twice. This is described in Section 6.

In nonsecure protocols, much smaller identifiers are adequate because we are confident that we will not encounter day-old packets. In the secure protocol the possibility of an active intruder causes us to make our packets permanently uniquely identified. In total we use 14 bytes to identify a call: 2 bytes for the machine identifier, 4 bytes for the conversation identifier, 4 bytes for the call sequence number, 2 bytes for the process identifier in the clear text, plus another 2 in the cipher text. This adds significantly to the minimum packet transmission time and to the time required to construct or interpret the packet. We could perform adequate duplicate elimination for nonsecure calls by using a 2-byte machine identifier, 2-byte call sequence number, and 2-byte process identifier. The size could be reduced significantly by more subtle encodings, but only at the cost of processing time in interpreting and verifying it. We also must maintain and look up the table in the server giving information about each conversation. The packet exchange of the RFA mechanism is required solely for secure calls.

We are not unhappy about these costs, though. The cost of security is not very high, and we are happy to pay it in order to get away from our previous completely unprotected state. Of course, there is no need for all clients to pay the cost of security. Those making nonsecure calls can happily use the simpler protocol we

described in [3]. We also offer an intermediate style that uses secure authentication but does not encrypt the calls themselves.

8. ADDITIONAL DETAILS OF THE SECURITY PROTOCOL

Cryptologists are aware that a cipher is often more easily broken if the cipher text for known plain text is available or if large amounts of cipher text are available encrypted with a single key. With DES it is not clear how important these threats are, but since the preventative measures are quite simple, we have included them in our security protocols. Similarly, the initialization vector used by the CBC mode of DES must be randomly chosen and securely distributed if the first plain text block is known or has only a small amount of unknown information. Some encryption hardware encourages a style of usage where the principal's private key is loaded into the unit only once (possibly manually) and encryption is always by *working keys*, which are presented to the unit encrypted with the private key; the working keys and the private key need never occur outside the encryption hardware as plain text. In the following, KX and KY are temporary keys and J is an initialization vector, randomly chosen by the authentication service.

The authenticator is in fact

$$\{KX\}^{KB}\{\{CK\}^{KB}, T, A\}^{KX}.$$

This CBC encryption uses a zero initialization vector; it is important that $\{CK\}^{KB}$ is the first plain text block, since this value is unknown to an intruder. When the authentication service is sending the authenticator back to A, it actually sends

$$\{KY\}^{KA}\{J, \text{authenticator}, X, B, CK\}^{KY}.$$

This CBC encryption uses a zero initialization vector; it is important that J is the first plain text block, since this value is unknown to an intruder. The packet returned in response to an RFA packet actually contains

$$[B, \{J, \text{call-identifier}, Y\}^{CK}, \text{authenticator}].$$

Again, the CBC encryption uses a zero initialization vector. When packets in calls are encrypted in this conversation, J is used as the initialization vector. Remember that all encryptions other than encrypted keys use CBC mode with an additional checksum to detect modifications.

9. STATUS AND CONCLUSIONS

Our remote procedure call package is fully implemented and is in daily use by a number of applications. The protocol for accessing the Alpine file servers (which provide a file system featuring distributed atomic transactions) uses our security features, as does the control protocol for an ethernet-based voice project. Both of these applications have found the security mechanisms entirely painless to use.

Unfortunately the present implementation is not yet complete. First, most of our computers are not yet equipped with DES hardware, so at present we are

using a trivial exclusive-or scheme in place of a genuine encryption. Second, we have not yet retrofitted the Grapevine servers to support the secure authentication protocol. We are confident that neither of these changes would seriously disturb our implementation, but they do make it impossible for us to measure the performance impact of encryption.

We are happy with our decision to include secure communication in this package. It has enabled us to explore in detail the implications of our previous theoretical designs for secure communication. It has shown that secure communication can be successfully included in our protocol family, and that security can be presented to programmers as a communication facility that is easy and convenient to use. Once we have suitable hardware in place, we will rapidly be able to convert to a situation in which we are relying only on the physical security of the Grapevine servers and of the participating end users' workstations.

REFERENCES

1. BAUER, K. R., BERSON, T. A., AND FEIERTAG, R. J. A key distribution protocol using event markers. Tech. Rep. TR-81060, Sytek Inc., Sunnyvale, Calif., 1981.
2. BIRRELL, A. D., LEVIN, R., NEEDHAM, R. M., AND SCHROEDER, M.D. Grapevine: An exercise in distributed computing. *Commun. ACM 25,* 4 (Apr. 1982), 260–274.
3. BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2,* 1 (Feb. 1984), 39–59.
4. DENNING, D. E., AND SACCO, G. M. Timestamps in key distribution protocols. *Commun. ACM 24,* 8 (Aug. 1981), 533–536.
5. EHRSAM, W. F., MATYAS, S. M., MEYER, C. H., AND TUCHMAN, W. L. A cryptographic key management scheme for implementing the data encryption standard. *IBM Syst. J. 17,* 2 (1978), 106–125.
6. KLINE, C. S., AND POPEK, G. J. Public key vs. conventional key encryption. In *AFIPS Conference Proceedings 48* AFIPS Press, Arlington, Va., 1979, pp. 831–837.
7. MATYAS, S. M., AND MEYER, C. H. Generation, distribution, and installation of cryptographic keys. *IBM Syst. J. 17,* 2 (1978), 126–137.
8. NBS. Data Encryption Standard. FIPS publication 46, National Bureau of Standards, U.S. Department of Commerce, Washington, D.C., 1977.
9. NBS. DES Modes of Operation. FIPS publication 81, National Bureau of Standards, U.S. Department of Commerce, Washington, D.C., 1980.
10. NEEDHAM, R. M., AND SCHROEDER, M. D. Using encryption for authentication in large networks of computers. *Commun. ACM 21,* 12 (Dec. 1978), 993–999.
11. VOYDOCK, V. L., AND KENT, S. T. Security in higher-level protocols: Approaches, alternatives and recommendations. Tech. Rep. 4767, Bolt Beranek and Newman, Inc., (Oct. 1981). Also available as Tech. Rep. ICST/HLNP-81-19. National Bureau of Standards, Washington, D.C., Oct. 1981.
12. VOYDOCK, V. L., AND KENT, S. T. Security mechanisms in high-level network protocols. *ACM Comput. Surv. 15,* 2 Jun., 1983, 135–171.

Received February 1984; revised September 1984; accepted October 1984