A Security Architecture for Fault-Tolerant Systems

MICHAEL K. REITER
AT & T Bell Laboratories
and
KENNETH P. BIRMAN and ROBBERT VAN RENESSE
Cornell University

Process groups are a common abstraction for fault-tolerant computing in distributed systems. We present a security architecture that extends the process group into a security abstraction. Integral parts of this architecture are services that securely and fault tolerantly support cryptographic key distribution. Using replication only when necessary, and introducing novel replication techniques when it was necessary, we have constructed these services both to be easily defensible against attack and to permit key distribution despite the transient unavailability of a substantial number of servers. We detail the design and implementation of these services and the secure process group abstraction they support. We also give preliminary performance figures for some common group operations.

Categories and Subject Descriptors: C.2.0 [**Computer-Communication Networks**]: General—*security and protection*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.5 [**Operating Systems**]: Reliability—*fault tolerance*; D.4.6 [**Operating Systems**]: Security and Protection—*authentication*; *cryptographic controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*authentication*

General Terms: Security, Reliability

Additional Key Words and Phrases: Key distribution, multicast, process groups

1. INTRODUCTION

There exists much experience with addressing the needs for security and fault tolerance individually in distributed systems. However, less is under-

Because the Editor-in-Chief of ACM Transactions on Computer Systems is a coauthor of this paper, he played no role in the review process or acceptance decision for the manuscript. This work was performed while the first author was at Cornell University. This work was supported under DARPA/ONR grant N00014-92-J-1866, and by grants from GTE, IBM, and Siemens, Inc. Any opinions or conclusions expressed in this document are the authors' and do not necessarily reflect those of the ONR.
Authors' addresses: M. K. Reiter, AT & T Bell Laboratories, Holmdel, NJ 07733; email: reiter@research.att.com; K. P. Birman and R. van Renesse, Department of Computer Science, Cornell University, Ithaca, NY 14853; email: {ken;rvr}@cs.cornell.edu.

stood about how to address these needs simultaneously in a single, integrated solution. Indeed, the goals of security—or more precisely, the goals of secrecy and integrity—have traditionally been viewed as being in conflict with goals of availability, because the only generally feasible technique for making data and services highly available, namely replicating them, also makes them harder to protect [Herlihy and Tygar 1988; Lampson et al. 1992; Turn and Habibi 1986]. This conflict is particularly pertinent to services that are involved in enforcing security policy, or in other words, that are part of the *trusted computing base* (TCB) [Department of Defense 1985] of a system. Prudence dictates that the TCB should be kept as small and localized as possible, in order to facilitate its protection. Distribution of TCB components makes its protection more difficult.

We have designed a security architecture for fault-tolerant systems that illustrates that this conflict need not result in an unreliable or insecure system. The architecture supports *process groups*—a common paradigm of fault-tolerant computing [Amir et al. 1992; Birman and Joseph 1987b; Cheriton and Zwaenepoel 1985; Kaashoek 1992; Peterson et al. 1989]—as its primary security abstraction, and provides tools for the construction of applications that can tolerate both benign component failures and advanced malicious attacks. We have implemented this architecture as part of a new version of the Isis distributed programming toolkit [Birman and Joseph 1987b; Birman et al. 1991] called Horus, thereby securing Horus' *virtually synchronous* process group abstraction. An earlier paper [Reiter et al. 1992] presents the design rationale and an overview of the architecture. Here we emphasize how the security mechanisms have been built to be fault tolerant and efficient.

The tradeoff between security and availability is addressed in two ways in our architecture. At the level of user applications, the *secure process group* abstraction supported by the architecture provides a framework within which the user can balance this tradeoff for each application individually. In a secure group, applications can be efficiently replicated in a protected fashion: authentication and access control mechanisms enable the group members to prevent untrusted processes from joining, and if the members admit only processes on trustworthy sites, the members will enjoy secure communication and correct group semantics among themselves. These protection mechanisms limit how attackers can interfere with applications and, in particular, enable the user to control exactly where and how widely each application is replicated.

The second and more critical level at which this conflict is addressed is within the core security services in the TCB that underlie these mechanisms, and indeed the security of *all* process groups. As do other security architectures, ours uses cryptography to protect communication, and this in turn requires that a secure means of key distribution exist. Most key distribution mechanisms employ trusted services whose corruption or failure could result in security breaches or prevent principals from establishing secure communication; it is in these services that the conflict between security and availability is most apparent. We have developed an approach to reconciling this

conflict that exploits the semantics of these services and novel replication techniques to achieve secure, fault-tolerant key distribution.

The implementation of our security architecture as part of Horus has brought performance and user interface issues to the forefront of our work, as well. By using caching extensively and moving costly operations off critical paths whenever possible, we have achieved reasonable performance in the secure version of Horus without resorting to network-wide cryptographic hardware. This is particularly true for group communication, which we expect to account for the vast majority of group operations in most applications. Moreover, the implementation of the security mechanisms in Horus resulted in minimal changes to the Horus process group interfaces. So, tools and applications designed for the Horus interfaces should port easily to secure groups.

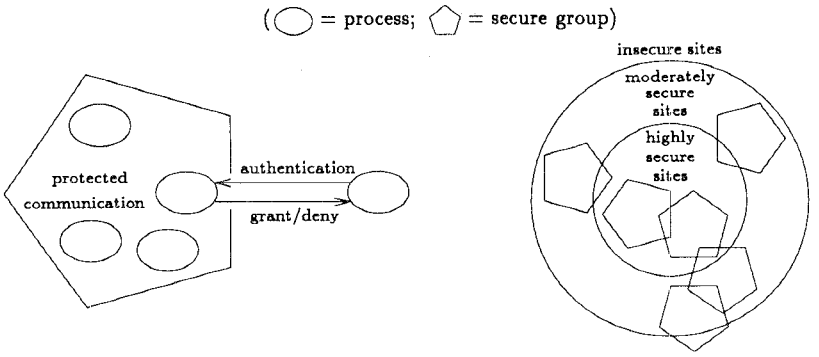
We present here the security architecture as realized in Horus and elaborate on the contributions just described. In Section 2 we present the programming model of secure process groups, with an emphasis on the security features that augment the Horus process group abstraction. Here we also discuss some implementation challenges posed by the programming model. In Section 3 we present our method of fault-tolerant key distribution, which we use to support secure process groups. In Section 4, we detail the implementation of secure process groups and give performance numbers for common group operations. We conclude and discuss related work in Section 5.

2. SECURE PROCESS GROUPS

The basic abstraction provided by Horus is the *process group*, which is a collection of processes with an associated *group address*. Groups may overlap arbitrarily, and processes may create, join, and leave groups at any time. Processes communicate both by point-to-point methods (e.g., RPC) and by *group multicast*, i.e., by multicasting a message to the entire membership of a group of which it is a member. Further, Horus supports a *virtually synchronous* [Birman and Joseph 1987a] execution model, so that message deliveries and changes to the *group view* (i.e., the membership of the group) appear atomically and in a consistent order at all group members, even when failures occur.

Our security architecture makes the Horus programming model robust against malicious attack, while leaving the model itself unchanged. First, during group joins, the group and the joining process are mutually authenticated to one another. More precisely, the group members are informed of the site from which the process is attempting to join, as well as the owner of the process according to that site. Any effort by an intruder to replay a previous join sequence or to forge the apparent site from which a request is sent will be detected. And, the joining process knows that responses apparently from the group are actually from the group that it is trying to join.

Second, a group member must explicitly grant each group join before the join is allowed to proceed. If the group members choose not to admit the joiner, they can deny the request, in which case the joiner will not be



(a) A process requesting to join is authenticated to the group members, who either grant or deny the request. Inside the group, communication is protected cryptographically from active and, if requested, from passive network attacks.

(b) Applications can be built from many secure groups to enforce internal security policies and to limit damage from site corruptions. A group can span sites secured to different levels, but is only as secure as the least secure site admitted.

Fig. 1. Secure process groups.

admitted provided that all group members are trustworthy. In particular, if the requesting site is not trusted by the group members to have properly authenticated the owner of the process requesting to join, the members may choose to deny the request.

Third, the integrity of these mechanisms and the Horus abstractions, as well as the authenticity of group communication, are guaranteed within each group that has admitted no processes on corrupt sites, i.e., sites at which an intruder has tampered with the hardware or operating system. (The requirement that a group not admit processes on corrupt *sites* does not imply that member *processes* need not be trusted, since an untrustworthy process could admit a corrupt site to the group.) Members can also request secrecy, in which case their messages will be encrypted before being sent, in an effort to prevent the disclosure of those messages to a network intruder. Secure point-to-point communication is also supported within and outside of groups, and in particular between group members and *clients* of groups [Birman et al. 1991], although in this article we focus on secure group communication.

The programming model thus presented to the programmer is one in which each process group can be viewed as a "fortress," where admission is regulated by the group members themselves (see Figure 1(a)). A setting to which this style of secure group is particularly well suited is one in which a fault-tolerant service must be provided to a larger, untrustworthy system against which the service must protect itself [Reiter et al. 1992]. Such an application could be composed of a single secure group located on a small "island" of trustworthy sites. Alternatively, a larger application in which

greater internal control is required could be implemented using many secure groups, arranged to enforce security policies within the application and to limit the damage to the overall application from the corruption of a site (see Figure 1(b)). While the groups could span sites with different levels of trustworthiness, each group is only as secure as the least secure site or process it contains.

When implementing secure process groups in Horus, we were faced with many issues in fault tolerance, performance, and integration, including the following:

—Because process groups are a fault tolerance tool, it was important that the integration of our security mechanisms not increase the sensitivity of the process group abstraction to failures. This was most difficult to achieve in authenticating the origin of join requests, since all known techniques for authenticating principals in open networks rely on trusted services whose unavailability could inhibit authentication but whose replication can make them more difficult to protect. Thus, we were forced to devise new techniques for achieving fault-tolerant authentication and key distribution to support authenticated group joins.

—In Horus, a process seeking to contact a group to obtain a service, or to join, will generally not know the current membership of the group and does not need to. Moreover, requiring this knowledge would involve substantial changes to Horus and significant overheads in the system. So, it was important that an outsider's ability to authenticate group members not rely on accurate knowledge of the group membership.

—Group communication can offer substantial performance benefits if the underlying network supports broadcast or multicast [Kaashoek 1992]. We felt that it was necessary to retain these potential benefits as much as possible, without requiring that special-purpose cryptographic hardware be deployed on all sites. This goal was particularly crucial to Horus, since experience with Isis suggests that group communication will be very common in Horus.

—Horus offers a variety of ordering guarantees on the delivery of group multicasts. One of these, the *causal* ordering property, raises security issues when causal relationships exist between multicasts in different overlapping groups [Reiter and Gong 1993; Reiter et al. 1992]. It was important to identify potential security threats to applications that employ this ordering property and to provide defenses against these threats whenever possible.

The following sections detail how we addressed these and other issues in the implementation of our security architecture. Section 3 presents techniques to achieve fault-tolerant key distribution, which we use in our architecture to support authenticated group joins fault tolerantly. However, these techniques are also of interest outside of the context of our security architecture and could be useful in a wide range of systems, and so in Section 3 we present them in a general light. A discussion of their use in our security

architecture is deferred until Section 4, where we focus on the implementation of secure process groups.

3. FAULT-TOLERANT KEY DISTRIBUTION

In open networks, an intruder can attempt to initiate spurious communication in two ways [Voydock and Kent 1983]: it can try to initiate communication under a false identity, or it can replay a recording of a previous initiation sequence. Many *authentication* or *key distribution* protocols have been proposed to protect against these attacks (see Denning and Sacco [1981], CCITT [1988], Kent [1993], Needham and Schroeder [1978], Steiner et al. [1988]). These protocols allow principals (e.g., computers, users) initiating communication to verify each others' identities and the timeliness of the interaction. Most also arrange for the involved principals to share a secret cryptographic key by which subsequent communication can be protected, or to possess each others' public keys, by which either communication can be protected or a shared key can be negotiated.

Authentication protocols typically employ a trusted service, commonly called an *authentication service* [Needham and Schroeder [1978]], to counter the first type of attack. In shared-key protocols, the authentication service normally shares a key with each principal and uses these keys to distribute other shared keys by which principals communicate. In public-key protocols, the authentication service usually has a well-known public key and uses the corresponding private key to certify the public keys of principals.

A predominant technique to detect replay attacks in authentication protocols is to incorporate into each protocol message the time at which the message was generated; the message is then valid for a certain *lifetime*, beyond which it is considered a replay if received [Denning and Sacco 1981]. Timestamp-based replay detection has been used in several systems (e.g., Steiner et al. [1988], Tardo and Alagappan [1991], Wobber et al. [1993]) and is often preferable to challenge-response techniques [Needham and Schroeder [1978]], because it results in fewer protocol messages and less protocol state. However, using timestamps requires that all participants maintain securely synchronized clocks. In practice, clock synchronization is usually achieved via a *time service*, as in Gusella and Zatti [1984] and Mills [1989].

The dependence of authentication protocols on authentication and time services raises troubling security and availability issues. First, the assurances provided by authentication protocols rely directly on the security of these services, and thus these services must be protected from corruption by an intruder. Second, the unavailability of these services may prevent principals from establishing secure communication, or even open security "holes" that could be exploited by an intruder. For instance, the unavailability of a time service could result in clocks drifting far apart, thereby exposing principals to replay attacks. To increase the likelihood of these services being available, they could be replicated. However, as already noted in Section 1, this is dangerous in some environments, because replicating data or services makes them inherently harder to protect.

We have developed techniques to reconcile the conflict between security and availability in these services. By using replication only when necessary, and introducing novel replication techniques when it was necessary, we have constructed these services to be easily defensible against attack. And, the transient unavailability of even a substantial number of servers does not hinder key distribution between principals or expose protocols to intruder attacks. Client interactions with the services are simple and efficient, and the services can be used with many different authentication protocols.

3.1 The Time Service

The security risks of clock synchronization failures in authentication protocols are well known [Denning and Sacco 1981; Gong 1992], and the need for a *secure* time service that cannot be tampered with or impersonated has been recognized in several systems (see Bellovin and Merritt [1990] and Mills [1989]). We claim, however, that the case for a *highly available* time service is not as clear. It is true that an extended period of unavailability might cause principals to have increasingly disparate views of real time. But, in itself this need not result in security weaknesses or inhibit communication too quickly. In evidence of this, the algorithm we propose by which clients estimate real time allows key distribution to proceed securely even during a lengthy unavailability of the time service. This has allowed us to explicitly *not* replicate our time service so that it will be easier to protect, and to achieve resilience to a time service unavailability through the client algorithm for estimating time.

We describe this algorithm in Section 3.1.1 and discuss alternatives to our approach in Section 3.1.2. As will be discussed in Section 3.1.2, the algorithm of Section 3.1.1 is heavily influenced by previous work in clock synchronization. As such, its contribution lies mainly in how clock synchronization techniques can be adapted for use in our setting to achieve simple, fail-safe time estimation in key distribution protocols with an easily defensible, centralized time service.

3.1.1 *The Algorithm.* Clients interact with our time service by the simple RPC-style protocol shown in Figure 2. We assume that the time server possesses a private key $K_{\mathcal{T}}^{-1}$ whose corresponding public key $K_{\mathcal{T}}$ is well known. (There is a similar shared-key protocol.) At regular intervals, a client queries the time service with a *nonce identifier* N [Needham and Schroeder 1978], a new, unpredictable value. When the time server receives this request, immediately it generates a timestamp T equal to its current local clock value and replies with $\{N, T\}_{K_{\mathcal{T}}^{-1}}$, i.e., the nonce and the timestamp, both signed with $K_{\mathcal{T}}^{-1}$. The client considers the response valid if it contains N and can be verified with the public key of the time service. The method by which a client uses this response rests on the following additional assumptions:

(1) *The client has access to a local hardware clock H that measures the length $t - t'$ of a real time interval $[t', t]$ with an error of at most $\rho(t - t')$ where ρ is a known constant satisfying $0 \leq \rho < 1$.* That is,

$$(1 - \rho)(t - t') \leq H(t) - H(t') \leq (1 + \rho)(t - t'). \quad (1)$$

$$C \rightarrow \mathcal{T}: \quad N$$
$$\mathcal{T} \rightarrow C: \quad \{N, T\}_{K_{\mathcal{T}}^{-1}}$$

Fig. 2. Protocol by which client C interacts with time service $\mathcal{T}$.

If ρ is estimated too optimistically so that the actual drift rate of the client is outside of $[-\rho, \rho]$, then the client may be subject to replay attacks or may subject others to replays of its messages. So, the value of ρ should be estimated conservatively; e.g., a ρ on the order of 10^{-5} should be sufficiently conservative for most types of quartz clocks [Cristian 1989].[1]

(2) *The time server's clock is perfectly synchronized to real time.* Actually, it suffices for the time server's clock simply to make progress at the rate of real time, although assuming that the time server's clock is identical to real time simplifies the following discussion. Moreover, we could incorporate time server drift into our formulas, but for all practical purposes, perfect synchronization can be achieved by attaching a WWV receiver or a very accurate clock to the time server's processor by a dedicated bus.

(3) *There are known, minimum real-time delays min_1 and min_2 experienced, respectively, between when a client initiates a request to the time service and when the time server receives that request, and between when the server reads its local clock value and the response is verified as authentic at the client.* The values of min_1 and min_2 can be determined by measurement in the absence of system load, taking into account the minimum signature and verification times for all possible reply and key values. Alternatively, min_1 and min_2 can simply be set to zero, although accurately determining these values tends to increase the duration for which the system can operate without the time service. In our implementation, min_2 is substantially longer than min_1, because it includes the delays for signing and verifying the response.

Under these assumptions, the following theorem holds:

THEOREM 3.1.1.1. *Immediately after a client receives and verifies a response from the time service, the client can characterize the current real time $\hat{t}$ by:*

$$\hat{t} \in [T + min_2, T + r/(1 - \rho) - min_1], \tag{2}$$

where T is the timestamp in the response and r is the round-trip time measured by the client, beginning when it sent the request and ending at time $\hat{t}$, i.e., after it verified the response.

PROOF. Let $min_1 + \alpha_1$ and $min_2 + \alpha_2$ be the *real* time delays experienced, respectively, between when the client sent the request and the server received it, and between when the server read its local clock and the client had verified the response as authentic. Then, $R = min_1 + min_2 + \alpha_1 + \alpha_2$ is the real round-trip time. Since $\alpha_1, \alpha_2 \geq 0$, we have that $0 \leq \alpha_2 \leq R - min_1$

[1] Keith Marzullo has suggested the possibility of dynamically measuring ρ on a per-client basis (personal communication, Feb. 1993). However, we do not pursue this here.

$- min_2$, and so after the client verifies the response, real time $\hat{t} = T + min_2 + \alpha_2$ satisfies

$$\hat{t} \in [T + min_2, T + R - min_1]. \tag{3}$$

By (1), it follows that $R \leq r/(1 - \rho)$, and by combining this with (3) we get the desired result. □

Combining (1) and (2), the client can characterize any later time $t \geq \hat{t}$ by:

$$t \in [L(t), U(t)], \tag{4}$$

where

$$L(t) = (H(t) - H(\hat{t}))/(1 + \rho) + T + min_2$$

and

$$U(t) = (H(t) - H(\hat{t}))/(1 - \rho) + T + r/(1 - \rho) - min_1.$$

To estimate the time, the client uses either $L(t)$ or $U(t)$, depending on which is more conservative. In particular, to detect replays of authentication protocol messages, principals use the following rules for estimating time:

(1) When timestamping an authentication protocol message to allow others to detect a later replay of that message, the sender sets the message timestamp to $T = L(t)$.
(2) A recipient accepts an authentication protocol message with timestamp T as valid at time t only if $T + \Delta > U(t)$, where Δ is the predetermined lifetime of the message.

The benefit of this scheme is that it is *fail-safe*, in the following sense:

THEOREM 3.1.1.2. *An authentication protocol message with lifetime Δ sent by a (correct) client at time t will never be accepted by another (correct) client after time $t + \Delta$.*

PROOF. Suppose a client sends an authentication protocol message at time t. The timestamp $T = L(t)$ for the message satisfies $T \leq t$. Now consider a recipient at time $t + \Delta$, where Δ is the lifetime of the message. Since at the recipient, $t + \Delta \leq U(t + \Delta)$, it follows that $T + \Delta \leq U(t + \Delta)$. Thus, the message will be rejected as invalid. □

Because the interval (4) grows wider with time, periodically each client resynchronizes with the time service in order to narrow its interval. A successful resynchronization results in new values of $H(\hat{t})$, r, and T for the calculation of $U(t)$ and $L(t)$. Resynchronization attempts can fail, however, when the round-trip time r for the attempt exceeds some timeout value. When this happens, the client continues to attempt to resynchronize with the service at regular intervals, while retaining the values of T, r, and $H(\hat{t})$ obtained in the last successful resynchronization to calculate $L(t)$ and $U(t)$. So, if the service becomes unavailable, clients' intervals will continue to widen. If the service is unavailable for too long, eventually the principals' values of $U(t)$ will exceed their values of $L(t)$ by the protocol message lifetimes, and all messages will be perceived as expired immediately upon creation.

While this bounds the amount of time that the system can operate without the time service, calculations in our system indicate that this bound is not very tight. For example, consider two principals P_1 and P_2, each of whose clocks is characterized by $\rho = 10^{-5}$, and suppose for simplicity that the values of $\hat{t}$ and T corresponding to the last resynchronization for each prior to a time service crash are the same. Moreover, suppose that $min_1 = min_2 = 0$ and that the value of r obtained by P_2 in its last resynchronization is 0.5 seconds. Then, even if the clocks of P_1 and P_2 drift apart at the maximum possible rate—i.e., the clocks of P_1 and P_2 are as slow and as fast as possible, respectively, while still satisfying (1)—it will be over 20.4 hours from $\hat{t}$ before the value of $U(t)$ at P_2 exceeds the value of $L(t)$ at P_1 by 30 seconds, a relatively short message lifetime in comparison to that suggested by Denning and Sacco [1981]. Additionally, the parameters used in the above calculation are very conservative for most settings, and tests in our system show that a time service unavailability can typically be tolerated for much longer. These results lead us to believe that the system, if tuned correctly, should be able to operate without the time service for sufficiently long to restart the time service, even if the restart requires operator intervention.

3.1.2 *Comparison to Alternative Designs.* We derived our algorithm from that presented by Cristian [1989] for implementing a time service. The primary difference between ours and Cristian's lies in how clients use the interval (2). In the latter, the client uses the midpoint of (2) as its estimate of the time at time $\hat{t}$, since this choice minimizes the maximum possible error, and the client estimates future times as an offset, equal to the measured time since the last resynchronization, from this midpoint.[2] However, like any other clock synchronization algorithm in which each client maintains a single clock value, this algorithm is not fail-safe: e.g., if the midpoint of (2) were too low, then the client's future estimates of the time would tend to be low, and thus expired messages may be incorrectly accepted. We feel that our approach, which is fail-safe, is better for our purposes.

A reasonable alternative to not replicating our time service is to replicate it for high availability and to compensate for the increased difficulty of protecting the service by making it tolerant of the corruption of some servers. For instance, a client could use the robust averaging algorithm of Marzullo [1990] to obtain an interval of bounded inaccuracy containing real time from a set of n time servers, if fewer than $\lfloor n/3 \rfloor$ servers are faulty or corrupt. This approach might be attractive if clients are highly transient, and thus a time service unavailability will prevent large numbers of clients from synchronizing initially with the service at client startup. However, this is unlikely to be the case in most systems, where time service clients are sites that do not tend to reboot frequently. Moreover, a replicated time service places a larger burden on the administrator of the service than does ours, since the administrator must protect multiple servers, instead of only one, to ensure the

[2] This is a simplification of the algorithm by Cristian [1989]; the actual algorithm also takes measures to ensure that client clocks are continuous and monotonic. These features, however, are unimportant for our purposes.

integrity of the service. For these reasons and the additional costs of replication (e.g., authenticating and maintaining multiple time servers), we feel that a replicated time service is difficult to justify for our purposes.

Also discussed by Marzullo [1990] are approaches to evaluating a predicate on a physical state variable despite the impossibility of accurately measuring that variable. It is observed that given a range of values that is known to contain the actual physical value, safe evaluation of the predicate may require that *all* values in the range satisfy the predicate, or that only *some* value in the range does. Our approach of estimating time conservatively with the endpoints of (4) can be viewed as an instance of the former approach, where the physical state variable being measured is time; the range containing time is (4); and the predicates of interest relate time to timestamps in authentication protocol messages.

Numerous other approaches to clock synchronization have been proposed (see, e.g., Simons et al. [1990]), but for brevity, we do not discuss them all here. Unlike ours, however, most assume upper bounds on message transmission times or employ greater distribution, thereby increasing the number of components that must be protected in the system. Moreover, to our knowledge none provide a fail-safe algorithm for estimating time in authentication protocols. We thus feel that our approach is unique in providing this property with relatively few requirements.

3.2 The Authentication Service

Our authentication service is of the public-key variety, that produces public-key *certificates* for principals. Each certificate $\{P, T, K_P\}_{K_{\mathcal{A}}^{-1}}$ contains the identifier P of the principal, the public key K_P of the principal, and the *expiration* time T of the certificate, all signed by the private key $K_{\mathcal{A}}^{-1}$ of the authentication service. A principal uses these certificates to map principal identifiers to public keys, by which those principals (who presumably possess the corresponding private keys) can be authenticated; the details are discussed in Lampson et al. [1992]. In general, a principal can request a certificate for any principal from the authentication service.

The need for security in such an authentication service is obvious: as the undisputed authority on what public key belongs to what principal, the authentication service, if corrupted, could create public-key certificates arbitrarily and thus render secure communication impossible. It would also appear that, unlike the time service, the authentication service must be highly available, since its unavailability could prevent certificates from being obtained or refreshed when they expire. Other researchers have also noted that both security and availability, and thus the conflict between them, must be dealt with in the construction of authentication services [Gong 1993; Lampson et al. 1992]. The most common approach to address this conflict in public-key authentication services is to implement the service using two services: a highly secure *certification authority* that creates certificates, and a highly available certificate database that distributes them (see CCITT [1988], Kent [1993], Lampson et al. [1992], Tardo and Alagappan [1991]). Our approach differs in that it performs both of these functions in a single

replicated service, but does so in such a way that the service remains correct and available despite even the malicious corruption of a minority of servers. So, the conflict between security and availability is addressed by replicating the service for availability, but compensating for the increased difficulty of protecting the service by making the service tolerant of successful attacks on servers. We first describe our approach, and then compare it in detail to other alternatives.

3.2.1 *The Algorithm.* Reiter and Birman [1994] describe a technique for securely replicating any service that can be modeled as a state machine. The technique is similar to *state machine replication* [Schneider 1990], in which a client sends its request to *all* servers and accepts the response that it receives from a majority of them. In this way, if a majority of the servers is correct, then the response obtained by the client is correct. The approach of Reiter and Birman provides similar guarantees but differs by freeing the client from authenticating the responses of all servers. Instead, the client is required to possess only one public key for the service and to authenticate only one (valid) response, just as if the service was not replicated.

We have constructed our authentication service using this technique. In its full generality, the system administrator can choose any *threshold value* k and create any number $n \geq k$ of authentication servers such that the service has the following properties:

Integrity. If fewer than k servers are corrupt, the contents of any properly signed certificate produced by the service were endorsed by some correct server.

Availability. If at least k servers are correct, the service produces properly signed certificates.

As indicated above, a natural choice for the threshold value is $k = \lfloor n/2 + 1 \rfloor$, so that a majority of correct servers ensures both the availability and the integrity of the service.

Our technique employs a *threshold signature scheme.* Informally, a (k, n)-threshold signature scheme is a method of generating a public key and n *shares* of the corresponding private key in such a way that for any message m, each share can be used to produce a *partial result* from m, where any k of these partial results can be combined into the private-key signature for m. Moreover, knowledge of k shares should be necessary to sign m, in the sense that without the private key it should be computationally infeasible to

(1) create the signature for m without k partial results for m,

(2) compute a partial result for m without the corresponding share, or

(3) compute a share or the private key without k other shares.

Our replication technique does not rely on any particular threshold signature scheme. For our authentication service, we have implemented the one of Desmedt and Frankel [1992], which is based on RSA [Rivest 1978].

Given a (k, n)-threshold signature scheme, we build our authentication service as follows. Let $\mathscr{A} = \{AS_1, \ldots, AS_n\}$ be the set of authentication servers.

These servers should satisfy the same specification, although they need not be identical; in fact, it may be preferable that they be developed independently, to prevent a (possibly deliberate) design flaw from affecting all of them [Joseph 1987]. We first choose a threshold value k and create n shares from the private key $K_{\mathscr{A}}^{-1}$ of the authentication service. Each authentication server AS_i, when started, is given the ith share of $K_{\mathscr{A}}^{-1}$, its own private key $K_{AS_i}^{-1}$, the public key K_{AS_j} of each server AS_j, and the public keys for all principals. It is also given the public key of the time service to synchronize its clock as in Section 3.1.1.

The protocol by which clients obtain certificates from the authentication service is shown in Figure 3. A client C requests a certificate for a principal P by sending the identifier for P and a timestamp T to the servers. The purpose of T is to give the servers a common base time from which to compute the expiration time of the certificate;[3] we discuss later how C chooses T. When each server AS_i receives the request, it extracts T and tests if T is no more than its current value of $L(t)$. If this is the case, it produces its partial result $pr_i(P, T + \Delta, K_P)$ for the contents $(P, T + \Delta, K_P)$ of P's certificate, where Δ is the predetermined lifetime of the certificate. AS_i then sends $pr_i(P, T + \Delta, K_P)$ to the other servers, signed under its own private key. (Alternatively, partial results can be sent over point-to-point authenticated channels, rather than being authenticated by digital signatures.) When AS_i has authenticated $k - 1$ other partial results from which it can create the certificate for $\{P, T + \Delta, K_P\}_{K_{\mathscr{A}}^{-1}}$, it sends the certificate to C. C accepts the first properly signed certificate for P with an expiration time sufficiently far in the future, and ignores any other replies.

It is easy to see why this protocol provides the Integrity and Availability guarantees just stated. Informally, Integrity holds because if only fewer than k servers are corrupted by an intruder, then the corrupt servers do not possess enough shares to sign a certificate; i.e., they need the help of a correct server. Availability holds because if at least k servers are correct, then the correct servers possess enough shares to sign a certificate and can do so using this protocol.

Because each correct server produces a partial result only if T is no more than its value of $L(t)$, where t is the time at which it receives the request, any certificate produced from its partial result has an expiration timestamp of at most $t + \Delta$. A principal accepts a certificate as valid at some time t only if the certificate expiration time is greater than the principal's value of $U(t)$, which ensures that the certificate expiration time has not been reached. So, like authentication protocol messages (see Section 3.1.1), a certificate will never be considered valid for longer than its intended lifetime.

A client's choice for T is constrained by two factors. On the one hand, for a certificate to be produced, each of k different servers must find T to be at most $L(t)$, were t is the time at which the server receives the request; so,

[3] In a prior version of this protocol, each server used its value of $L(t)$ when the request was received as the base to compute the expiration time. This version was more sensitive to clock drifts and variances in request delivery times.

$$C \rightarrow \mathscr{A}: \quad P, T$$
$$(\forall i)\ AS_i \rightarrow \mathscr{A}: \quad \{P, T + \Delta, pr_i(P, T + \Delta, K_P)\}_{K_{AS_i}^{-1}}$$
$$(\forall i)\ AS_i \rightarrow C: \quad \{P, T + \Delta, K_P\}_{K_{\mathscr{A}}^{-1}}$$

Fig. 3. Protocol by which client C obtains a certificate for principal P.

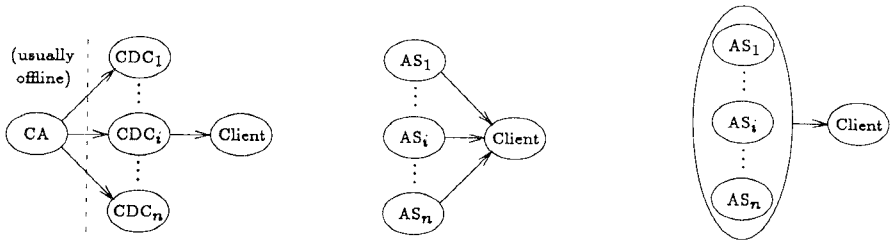
choosing T too high prevents a certificate from being produced. On the other hand, since the certificate's expiration time is $T + \Delta$, the client shortens the effective lifetime of the certificate by choosing T too low. So, a client should choose T to be close to, but less than, what it anticipates will be the correct servers' values of $L(t)$ when they receive the request.

In practice, it works well to have a client, when sending a request at time t, to set T to its own value of $L(t)$ minus a small offset $\delta \geq 0$, and to increase δ on subsequent requests if prior attempts to obtain a certificate failed. Because an unavailability of the time service will generally cause clients' values of $L(t)$ to drift from those of the servers, during a lengthy unavailability a client may need to set δ to several seconds to obtain a certificate, at the cost of reducing the effective lifetime of the certificate by that amount. However, since certificate lifetimes are typically at least several minutes, this would normally reduce the effective lifetime by only a small fraction.

3.2.2 *Comparison to Alternative Designs.* As previously mentioned, we are not the first to notice the conflict between security and availability in the construction of authentication services. Gong [1993] proposed a method for dealing with this tradeoff in shared-key authentication services such as Kerberos [Steiner et al. 1988]. Lampson et al. [1992] also discussed this tradeoff and described a different solution that is appropriate for a public-key authentication service similar to ours.

In the latter solution, which is also implemented in SPX [Tardo and Alagappan 1991], certificates are created by a highly secure *certification authority*. The certification authority is not replicated and can even be taken offline, to make it easier to protect (Figure 4(a)). To reduce the impact of its limited availability, it produces long-lived certificates that are stored in and distributed from an online *certificate distribution center* (CDC), which can be replicated for high availability [Tardo and Alagappan 1991]. Because certificates are long lived, however, there must be some way to revoke them securely. For this reason, certificates are obtained only from CDC replicas, so if necessary, a certificate can be revoked by deleting it from all replicas. That is, a client accepts a certificate only if both the highly secure certification authority and a CDC replica endorse it. A disadvantage of this scheme, noted by Lampson et al. [1992], is that the corruption of a CDC replica could delay the revocation of a certificate.

This problem could be addressed by using the technique described in Reiter and Birman [1994] to replicate the CDC *securely*. However, our approach presented in Section 3.2.1 of securely replicating the authentication service itself (Figure 4(c)) addresses this problem more directly. Since the authentication service is online and highly available, it can refresh certificates frequently and create them with short lifetimes. Thus, the window of vulnerabil-



(a) Offline certification authority (CA) [CCITT 1988; Lampson et al. 1992; Tardo and Alagappan 1991]. CA deposits long-lived certificates in an online, replicated certificate distribution center (CDC).

(b) State machine replication [Schneider 1990]. Client authenticates certificate from each server and accepts key that occurs in a majority of certificates.

(c) Transparent state machine replication [Reiter and Birman 1992]. Client authenticates single certificate that could have been created only by a majority of servers.

Fig. 4. Design alternatives for a fault-tolerant public-key authentication service.

ity between the disclosure of a principal's private key and the expiration of the principal's certificates can be greatly shortened, making revocation of existing certificates less crucial. If the ability to revoke certificates is still desired, however, our authentication service could easily be adapted to produce certificate revocation lists [CCITT 1988; Kent 1993]. And, of course, once the disclosure of a principal's private key is discovered, the principal's public key can be removed from the authentication servers so that no more certificates containing it are produced.

Our technique also has advantages over state machine replication [Schneider 1990] (Figure 4(b)) of the authentication service (or the CDC of Lampson et al. [1992] and Tardo and Alagappan [1991]). First, in our approach a client's ability to verify the validity of a public key depends only on its knowledge of a single public key for the authentication service, rather than on knowledge of the identities of all servers and an ability to authenticate each of them individually. Second, our approach requires less computation at the client, since the client need not authenticate or retain any other replies from the service but the first one it accepts as valid. This is especially beneficial if a principal obtains and forwards its own certificate to its partners in cryptographic protocols, as in the "push" technique described by Lampson et al. If the service were implemented using state machine replication, the principal would need to authenticate, collect, and forward a number of certificates equal to the size of a majority of the servers, and the destination would need to authenticate all of these certificates, presumably on the critical path of the key distribution protocol. Third, in our approach the configuration of the service is largely transparent to clients, and so servers can be added or removed more easily.

There is, however, at least one disadvantage of our scheme with respect to the others mentioned here: due to the round of server communication, a client is likely to wait longer for a response from the authentication service in our scheme. As illustrated in Section 4, though, in many situations communication with the authentication service can be performed in the background, off

the critical path of any other protocol or computation, and in advance of any actual need for a certificate. (The certificate so obtained is then cached until the need for it arises.)

4. SECURE GROUP IMPLEMENTATION

As discussed in Sections 1 and 2, the services of Section 3 were motivated by the need for fault-tolerant authentication and key distribution in our security architecture. In this section, we detail the implementation of secure process groups in the Horus system, and in doing so, elaborate on the use of the services of Section 3 in our architecture. We also describe the mechanisms used to address other integration and performance concerns, such as those enumerated in Section 2.

The implementation of our architecture in Horus is shown in Figure 5. On each site, the core Horus functionality is implemented in a transport layer entity called MUTS and a session layer entity called VSYNC, both of which will reside in the operating system kernel on most platforms [van Renesse et al. 1992]. The purpose of MUTS is to provide reliable, sequenced multicast among sites; VSYNC then implements the process group and virtual synchrony abstractions over this service. Horus will also provide user-level libraries and tools to facilitate primary-backup computations, replicated data, etc. While security mechanisms are included at all levels of the system, the core functionality of the security architecture exists in MUTS and VSYNC. These layers have been augmented to distribute and use *group keys*.

4.1 Group Keys

Group keys form the foundation of security within each process group. The group keys are a pair of cryptographic keys that are replicated in volatile memory at each site in the group. While the security dangers of replication also apply to group keys, we view this replication as acceptable at this level of the system for two reasons. First, the user has complete control over where the group members, and thus the group keys, reside. This gives the user the opportunity to determine a prudent degree of replication for each group, depending on the nature of the application and the environment in which it will run. Second, the damage resulting from imprudent replication of a group's keys is limited, because the disclosure of a group's keys corrupts only that group, not the entire system. Group keys are managed in the VSYNC layer and are never held by user processes. Moreover, group keys are intended to be used for the entire lifetime of the group, although the members of a group can effectively change their group keys simply by forming a new group with the same membership. To minimize the risk of exposing a group's keys, only those sites that need them—that is, the sites of current group members—should possess them at any given time. Thus, if a site leaves a group, it erases its copy of the group keys for that group in an effort to prevent the subsequent corruption of this site from disclosing the group keys to an outsider. In the event of a site failure, we rely on the loss of volatile storage to erase the keys from memory.

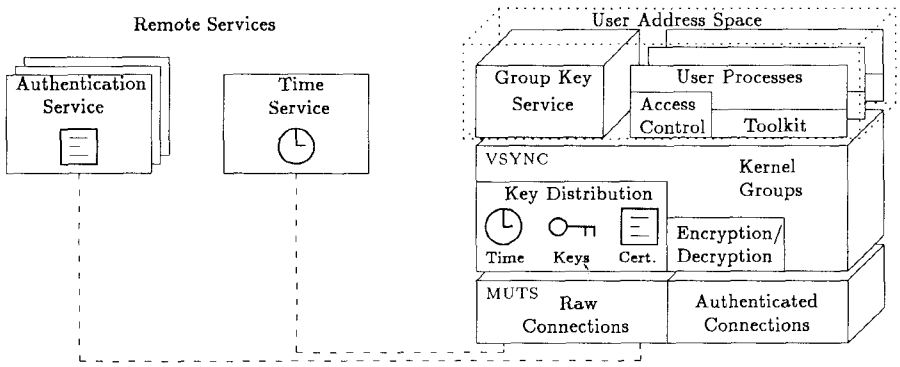


Fig. 5. The Horus security architecture.

The first of the group keys is a key to a symmetric, or shared-key, cipher. This key is called the *communication key* of the group, because it is used by MUTS and VSYNC to communicate securely with other group members. Group communication is not protected under the communication key directly. Rather, the communication key is used by MUTS to establish authenticated connections within the group, which preserve the authenticity and order of all internal group communication, and by VSYNC to distribute keys for the encryption of user messages. This indirect use of the communication key is done to limit the amount of communication protected under, and thus the amount of exposure of, the communication key. In our implementation, the communication key consists of two DES [National Bureau of Standards 1977] keys (112 bits), and encryption is performed with it using the triple-encryption technique of Tuchman [1979].

The second of the group keys is an RSA private key whose corresponding public key is incorporated into the group address. This private key is primarily used to authenticate sites and processes in the group to outsiders that possess the group address. Authentication of group members is necessary whenever a process needs to communicate with a group member, e.g., because the group is providing a service that the process desires. While group private keys are held within VSYNC, group members can obtain signatures on messages through a VSYNC interface. In the present implementation, the size of group private keys is a compile-time constant (we typically use 512-bit RSA moduli), but in the future we will allow the user to choose from a set of sizes when each group is created.

The group keys for a group are created by a user-level service on the site where the group is created. This service, called the *group key service*, generates sets of group keys in the background and caches them in VSYNC. This is done to remove the costly generation of an RSA key pair from the critical path of group creations. (With the implementation we presently use, generation of a 512-bit RSA modulus on a 33MHz Sparc ELC workstation usually costs at least several seconds.) When a local process requests to

create a group, VSYNC removes a set of group keys from its local cache and associates them with the group. Additionally, VSYNC creates and returns the group's address, which contains, among other things, the public key corresponding to the group private key and a location hint for the group (i.e., a transport address). Latency of a group creation is minimal unless the VSYNC cache is empty, which could happen, for instance, if the creation is requested shortly after the site is booted and before the group key service has produced a set of group keys.

Because other processes must use this group address to contact the group, the user process would typically distribute this address in some fashion. How this is done is up to the process, which has available to it secure group and point-to-point communication that it can use for this purpose. However, this distribution will often occur through the Horus *name service*, a user-level distributed service that implements a hierarchical name space, much like a file system. In this case, the user process (communicating over an authenticated channel) registers the group address under a name in the name space, from which other processes can read the address. To prevent an intruder from overwriting the address (and thus the public key) for a group, we have enhanced the name service to enforce access controls that restrict what processes may write addresses to a name. A more detailed discussion of the name service and access control is presented by Reiter et al. [1992].

We should note that this use of the name service requires that it be protected from tampering.[4] Since it plays a central role for many applications, it might be appropriate to replicate it using the techniques of Reiter and Birman [1994] as we did with the authentication service of Section 3.2. However, we have not yet done this in the present implementation, and in fact, the cost of doing so may be prohibitive for many general-purpose uses. We hope to explore alternative implementations of name services to address these issues in the future. We stress, however, that applications are not bound to use any particular name service or even to name their groups at all. Also, we permit multiple name services in our architecture, opening the possibility that highly secure name services could exist side-by-side with less secure ones.

4.2 The Group Join Protocol

Once a process has obtained the group address for a group, it can request to join the group. (Alternatively, it can request to become a *client* of the group; this is discussed in Section 4.4.) In this section we detail the protocol by which a process joins the group.

[4]The need to protect the name service can be reduced by storing group address "certificates" in it, that bind group names to group addresses with the signature of some authority (which could be application specific). Our name service will support this possibility but will not require it, because doing so implies that for each name, there must be some well-known principal(s) trusted by other principals to certify the group address for that name. Instead of requiring this for all groups, we have chosen to leave this matter of policy up to each application.

In preparation for group joins, each site A is booted with the public keys of the authentication and time services described in Section 3, and its own private key K_A^{-1}; the authentication service possesses the corresponding public key K_A. VSYNC synchronizes periodically with the time service. It also obtains a certificate $CERT_A$ for its site from the authentication service and refreshes this certificate periodically, well before its value of $U(t)$ surpasses the certificate expiration time.

When a process on site A desires to join a group, it provides to VSYNC the group address of the group it would like to join. It can also specify a message m to be sent to the group members. VSYNC then follows the protocol shown in Figures 6 and 7. In Figure 6, encryption and signature under key K are denoted by $[\cdot]_K$ and $\{\cdot\}_K$, respectively. (A message is "signed" with a shared key K by encrypting a one-way hash of the message with K [Stubblebine and Gligor 1992].)

(1) Site A creates a request containing

- —a global identifier G for the group that is obtained from the group address;
- —a timestamp T equal to the site's current value of $L(t)$;
- —the user id U of the apparent owner of the requesting process (although more generally U could be a representation of the user that the group members could verify themselves);
- —a new, secret 112-bit *reply pad* RP and new, secret 56-bit DES *reply key* RK, both encrypted under the public key of the group (i.e., $[RP, RK]_{K_G}$); and
- —the user-specified message m, encrypted under RK if the user so requested (e.g., because m is a password or capability for admission to the group).[5]

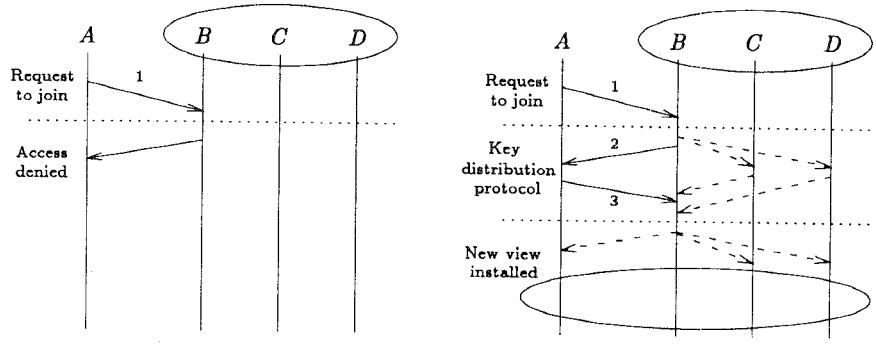
All of this information is signed with K_A^{-1} and preceded by A's certificate. A sends this message to the group using the location hint in the group address. Safety of this protocol does not rely on the hint being accurate. If it is incorrect, however, the requesting process must obtain a more current location hint for the group, e.g., through the name service mentioned in Section 4.1. For the rest of this discussion, we assume that A's message reaches a site in the group.

(2) When a site B in the group receives this message, it authenticates the requesting site by checking the authentication service's signature on the certificate, extracting the public key from the certificate, and then checking the signature on the request. Moreover, it verifies the timeliness of the message by comparing the timestamps in the request and the certificate to its current value of $U(t)$, as described in Sections 3.1.1 and 3.2.1,

[5]Long-term capabilities or passwords for admission to the group should not be protected under RK, because otherwise admission to the group, and thus CK, could be obtained simply by breaking the relatively weak key RK. If RK is needed to protect long-term information, it should be of strength comparable to that of CK.

$$A \rightarrow B: \quad CERT_A, \{G, T, U, [RP, RK]_{K_G}, [m]_{RK}\}_{K_A^{-1}}$$
$$B \rightarrow A: \quad \{A, N, CK \oplus RP, [K_G^{-1}]_{CK}\}_{RK}$$
$$A \rightarrow B: \quad \{N\}_{RK}$$

Fig. 6. VSYNC protocol by which site A joins group G, containing site B.



(a) A process on site A requests to join the group containing (processes on) sites B, C, and D. A sends the request to B, which delivers the request to its local member. Here, the member denies access, and B replies accordingly.

(b) In this case, access is granted. The group keys are securely sent to A in message 2, in parallel with a group synchronization protocol. After the keys are acknowledged (message 3), the new group view is installed.

Fig. 7. Overview of the VSYNC group join protocol.

respectively. If the message is determined to be valid, B uses the group private key K_G^{-1} to decrypt RK and, if necessary, RK to decrypt m. Then, it delivers an upcall to a local member, indicating the group G, site A, apparent owner U, and message m.

(3) The local member can take any measures including, for example, executing protocols with other group members or the requesting process, to determine whether the join should be granted. In particular, if the member does not trust A to have authenticated the owner of the process properly, then it should possibly not allow the process to join. Eventually it informs VSYNC of its decision.

(4) If access is granted, B returns the group keys to A as shown in Figure 6: the group communication key CK is encrypted with the reply pad RP by bitwise exclusive-or ($\oplus$), and the group private key K_G^{-1} is encrypted under CK (which is done off the critical path of this protocol). B also includes the identity of A and a nonce N in the message, which it signs with RK. Note that an attempt by a network intruder to replay this message to A in the future will be detected because each RK is a new (i.e., "fresh") key.

(5) After A authenticates B's reply and notes its own identity in the message, it acknowledges the keys by returning N signed with RK. A sends

this message solely for synchronization purposes, to inform B that it has received the group keys and thus that B can proceed in installing the new group view containing A (see Figure 7). If B does not receive this message after retransmitting the group keys several times, it proceeds in installing the new view anyway. If A did not receive the group keys, it will later be unable to communicate to group members and thus will be removed from the group as if it had failed.

There are two reasons that RP and RK are used to encrypt the group communication key and to sign B's response, respectively, versus using K_A and K_G^{-1} for these operations. First, the cryptographic operations with RP and RK are faster than the corresponding operations with the RSA keys. Second, in the event that A leaves the group and is later corrupted, the use of RP prevents this corruption from disclosing CK. That is, if the group keys were communicated to A encrypted under (only) K_A, the corruption of A would reveal K_A^{-1} and thus CK if the intruder had previously recorded the protocol by which A joined the group. Using RP to encrypt CK prevents this because after the join protocol completes, A and B destroy RP and any state that could be used to reconstruct it.

Two points are worth emphasizing about our use of the authentication and time services. First, neither service is on the critical path of the group join protocol. This is more notable in the case of the authentication service, because in most systems, the authentication service (or, in Lampson et al. [1992] and Tardo and Alagappan [1991], the CDC) *is* on the critical path of authentication protocols. Second, the transparency of replication in the authentication service simplifies the protocol. If the authentication service were replicated using state machine replication, each site would need to maintain certificates from a majority of servers to prepend to its requests. And, the destination of a request would need to authenticate these certificates on the critical path of the protocol, possibly resulting in a significant performance impact.

The latency of a group join in the present implementation, measured over SunOS 4.1.1 on moderately loaded 33MHz Sparc ELC workstations, is approximately 2.26 seconds on average. This cost is independent of group size, except for very large groups. Over 90% of this cost can be attributed to the modular exponentiation routines of the (software) RSA implementation we use presently:[6] with the modular exponentiation operations removed, the cost of a group join drops to 195ms on average. Clearly hardware support for

[6] In these tests we used the C implementation of RSA provided with the RSAREF toolkit, licensed free of charge by RSA Data Security, Inc. The RSAREF toolkit was developed to support privacy-enhanced electronic mail, not interprocess communication, and much faster software implementations of RSA exist. For example, Cryptolib performs RSA operations reportedly, with a 512-bit modulus and an 8-bit public exponent, at the rate of 160ms per signature and 30ms per verification on a Sparc 2 [Lacy et al. 1993]. With such an implementation, the cost of a group join should drop to roughly 600ms on such a platform. However, with any software implementation of which we are aware, the cost of RSA operations would continue to be the limiting factor in the performance of the group join protocol.

modular exponentiation would be of value for applications in which group membership is very dynamic. However, experience with Isis (see Birman et al. [1991]) leads us to believe that most Horus applications will employ largely static groups, and so we do not expect that most applications will require such hardware to use the security architecture effectively.

4.3 Secure Group Communication

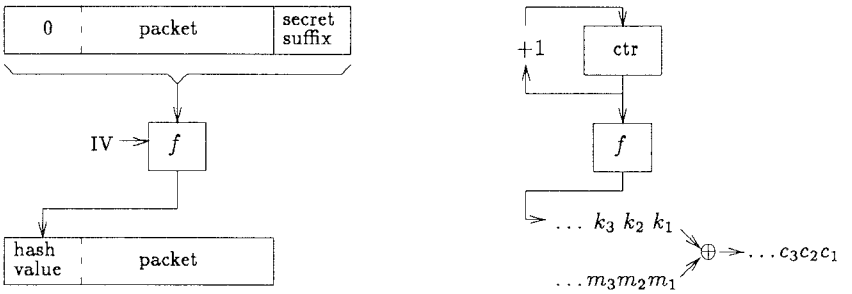
As discussed in Section 2, all communication within a secure group is protected cryptographically from tampering by a network intruder. And, if requested, group members' messages will be encrypted before being sent on the network. The mechanisms for performing these functions are decoupled in our system: verification of message authenticity is performed by MUTS, whereas encryption to prevent the release of message contents is performed in VSYNC.

The decision to decouple the mechanisms for preserving authenticity and secrecy is supported by several factors. First, maximum benefit from the authenticity mechanisms is achieved by incorporating them at the lowest layer of the system, within MUTS. This enables the VSYNC protocols to rely on the abstractions provided by MUTS, because all MUTS messages, and thus the MUTS abstractions, are protected from tampering by a network intruder. On the other hand, because only user messages need to be encrypted (we make no effort to address *traffic analysis* attacks [Voydock and Kent 1983]), the encryption mechanisms are incorporated at a much higher level, in VSYNC. This enables the encryption mechanisms to exploit the abstractions provided by the lower layers in its algorithms. Finally, decoupling these mechanisms allows us to use faster algorithms for each.

The algorithms for preserving authenticity and secrecy both rely on the cryptographic strength of a *one-way hash function*. Informally, a one-way hash function f has the properties that it is computationally infeasible to produce two inputs m_1 and m_2 such that $f(m_1) = f(m_2)$ or to produce any input m such that $f(m) = h$ for a given, prespecified value h. In recent years, several fast one-way hash functions have been proposed; our implementation presently offers the use of either MD4 [Rivest 1991] or MD5 [Rivest 1992]. Both of these functions process inputs of arbitrary length in 64-byte blocks, maintaining a 16-byte state between blocks, to produce a 16-byte hash value. MD5 is conjectured to be stronger than MD4, but in our tests is also approximately 30% slower.

4.3.1 *MUTS Packet Authentication.* Messages, or more accurately, MUTS packets, are authenticated on a per-connection basis. A MUTS connection is a logical data path from one MUTS instance to others. Only the instance that opened the connection can send data across it, although recipients on a connection also acknowledge packets over that connection.

The first packet sent on a connection includes a new, unpredictable *connection key*, encrypted under the communication key of the group in which the



(a) A MUTS packet is "signed" by appending to it a secret suffix and applying a one-way hash function f, initialized with a secret initialization vector (IV).

(b) VSYNC encrypts a message by XORing it with a key stream that is produced by applying a one-way hash function f to a counter. The counter is incremented between applications.

Fig. 8. Cryptographic operations.

connection is opened. The connection key consists of a 16-byte *initialization vector* (IV) and a *suffix* whose size is a compile-time constant (typically 16 bytes). This packet also contains a timestamp to allow the recipients to detect replay attacks, as described in Section 3.1.1.

To send a subsequent packet on the connection, the sender initializes the internal state of the hash function f to IV and then applies f to the packet and the suffix. (Alternatively, a secret *prefix*, that is processed by f before the packet, can be used in lieu of the IV [Tsudik 1993].) The result is placed in the packet header, as shown in Figure 8(a). A recipient of a packet verifies it by copying the hash value out of its header, clearing the hash field, applying f (initialized with the IV) to the packet and the suffix, and comparing the result to that copied from the packet. If the hash values match, then the recipient considers the packet authentic. Replays of packets are detected using sequence numbers in the packet headers. Acknowledgments on the connection are authenticated in the same manner.

To our knowledge, this form of message authentication was first proposed by Tsudik, although a variation of this approach was employed by Galvin et al. [1991], and a similar use of one-way hash functions in authentication protocols was proposed by Gong [1989]. Its security relies on the assumption that it is infeasible to determine the initialization vector and the suffix from packets and their hash values, and that the secrecy of the initialization vector and the suffix prevents a network intruder from forging proper hash values on packets. A substantial advantage of this approach over techniques that use encryption is speed, because fast one-way hash functions are typically much faster than encryption functions (e.g., MD4 in software is over three times as fast as the fastest software DES implementations [Lampson et al. 1992]).

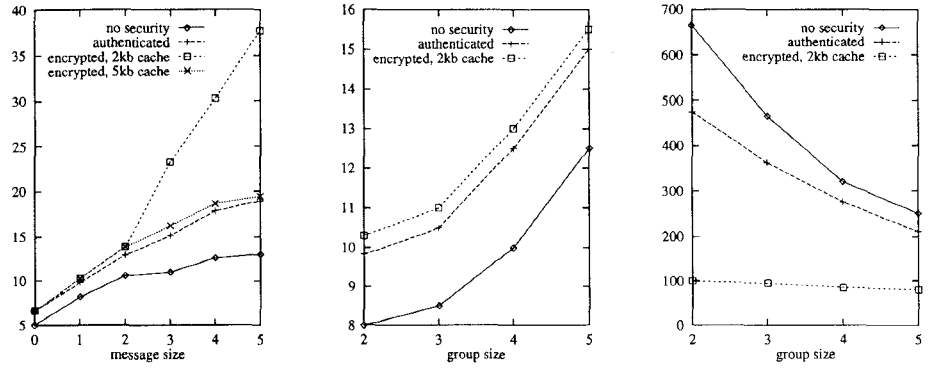
4.3.2 *VSYNC Message Encryption.* Because MUTS assures authentic, sequenced multicast in groups containing no corrupt sites, the encryption algorithm we employ in VSYNC to encrypt user messages need not attempt to protect the authenticity of those messages. This allows us to use faster encryption algorithms that are generally not suitable for protecting message authenticity in open networks. The form of cipher we have chosen is typically known as a *synchronous stream cipher* [Diffie and Hellman 1979].

In a stream cipher, the ciphertext of a message $m = m_1 m_2 \ldots$ is obtained by enciphering the ith element m_i with the ith element k_i of a secret *key stream* $k_1 k_2 \ldots$ A stream cipher is said to be *synchronous* if the key stream is generated independently from the message stream. In our implementation, each m_i and k_i is a single bit. The key stream is generated using the *counter method* of Diffie and Hellman: 16 bytes of the key stream are generated by applying a one-way hash function f to a 16-byte integer counter, and the counter is then incremented before the next application of f to obtain the next 16-bytes of the key stream (see Figure 8(b)). If f is sufficiently strong, then future portions of the key stream will be unpredictable to those not knowing the value of the counter, even if previously used portions of the key stream are discovered. So, the initial value of the counter is the key for producing the key stream. The ith bit c_i of the ciphertext is simply the exclusive-or of k_i and m_i (i.e., $c_i = k_i \oplus m_i$).

We use this cipher as follows. On each message installing a new group view containing n sites, the VSYNC instance sending the message includes a list of n new, unpredictable 16-byte integers encrypted under the communication key for the group. Each site in the group decrypts this list and initializes n integer counters to the n values in the list. The ith site in the group encrypts a message to the group using the key stream generated from the ith counter. Sites decrypt messages from the ith site by exclusive-oring the next portion of the ith key stream against the received message.

One requirement in using a synchronous stream cipher is that the key streams of the sender and receivers remain synchronized with one another. We have implemented these encryption mechanisms at a level within VSYNC that allows us to exploit some process group semantics for this purpose. In particular, the level at which encryption is performed within VSYNC ensures that a message encrypted while the group is in one view will be decrypted at all recipients in the same view. This makes view changes an appropriate time to reset the integer counter for each site in the group to a new value, which should be done occasionally to make cryptanalysis more difficult. Additionally, the level at which encryption is performed also ensures that messages from the same site are decrypted in the order they were encrypted, which automatically keeps the key stream for each site synchronized at all sites in the group between view changes.

An advantage of synchronous stream ciphers over other encryption methods is that they allow much of the work for encryption to be done in the background. In our system, we precompute and cache portions of the key stream for each site in a group so that the key stream is immediately available for use. This reduces the encryption latency of messages smaller



(a) Latency of member-to-group RPC (ms) as a function of message size (kb). Group of size two.

(b) Latency of member-to-group RPC (ms) as a function of group size. Message of size 1kb.

(c) Member-to-group bandwidth (kb/s) as a function of group size.

Fig. 9. Preliminary performance of group communication.

than the cache, as encrypting these messages requires simply exclusive-oring the precomputed key stream against the message (provided that the cache is full). The size of the cache for each site in a group is presently a compile-time constant, although this size could be dynamically adjustable.

4.3.3 *Performance.* Preliminary performance figures for group communication are pictured in Figure 9. In each graph, the line labeled "no security" indicates performance when no security mechanisms are employed. The line labeled "authenticated" indicates performance when packets are authenticated by MUTS. The lines labeled "encrypted, 2kb cache" and "encrypted, 5kb cache" indicate performance when packets are authenticated by MUTS and when user messages are encrypted by VSYNC. In the former case, each VSYNC instance maintained a 2 kilobyte (kb) cache of precomputed bits for each key stream; in the latter case, these caches were of size 5kb. These tests used MD4 for f and were performed between user processes on distinct, moderately loaded 33MHz Sparc ELC workstations running SunOS 4.1.1. The implementation of MD4 that we used in these tests is able to process up to 1.75 megabytes (Mb) of input per second on this platform.

In parts (a) and (b) of Figure 9 are average latencies for member-to-group RPC interactions. In a member-to-group RPC, one member multicasts a single message to the group, and all other members acknowledge (with a null message). The latency is the measured time at the sender between sending the initial message and receiving all acknowledgments. Part (a) illustrates this latency in milliseconds (ms) as a function of message size, and part (b) illustrates this latency as a function of group size, with a constant message size of 1kb. Part (c) of Figure 9 indicates member-to-group bandwidth, i.e., how much data can be pushed from a single member to the other group members per second. Each data point was obtained by performing a 1Mb

member-to-group RPC and dividing 1Mb by the time required for this RPC to complete.

Two items are worth noting about the graphs in Figure 9. First, in part (a) the rapid rise of the "encrypted, 2kb cache" curve relative to the "encrypted, 5kb cache" curve beginning after the 2kb message is due to the fact that in the 2kb-cache tests, the VSYNC instances each precomputed only 2kb of the key stream. So, while their cache contents sufficed to encrypt and decrypt the 2kb message, the 3kb message exhausted their caches and forced them to generate parts of the key stream before sending or delivering the message. Similarly, in part (c) the large impact of encryption is partly due to the immediate exhaustion of the VSYNC caches when encrypting and decrypting very large (in these tests, 1Mb) messages. Put another way, precomputing portions of the encryption key stream helps very little in increasing bandwidth.

The second item of interest regards the graph in part (b). While not surprising, it is still interesting to note that group size has virtually no effect on the cost of message encryption or packet authentication. The increase in latency as a function of group size is primarily a result of sending the message to increasingly many destinations, and if hardware multicast were available, this increase should virtually disappear. At the time of this writing, however, we have not yet experimented with hardware multicast.

In addition to hardware multicast, we plan to pursue other optimization to the performance of group communication. We are continuing to optimize the cryptographic mechanisms of our implementation. Also, we intend to incorporate flow control mechanisms into MUTS to improve performance. At the present time, MUTS provides little flow control. So, to prevent packets from being dropped by the operating system in the tests of Figure 9, we had to use small packet sizes and frequent acknowledgments, resulting in a large number of packets and thus cryptographic operations at the MUTS layer. We anticipate that flow control mechanisms will improve the performance of MUTS transport substantially.

4.4 Clients of Groups

When a process obtains a group address for a group, it may choose to become a *client* of the group [Birman et al. 1991], rather than requesting to join the group. The client abstraction gives a process a way to communicate efficiently with group members, without exposing the group members to the risk resulting from admitting the process to the group. This abstraction is most useful when a process group is intended to provide a service to a less trusted system.

Clients are a user-level notion managed by user-level toolkit protocols; i.e., VSYNC does not support clients of groups, except to provide generic secure point-to-point channels over which clients and group members communicate. To be more precise, the protocol by which a process becomes a group client begins by the process asking VSYNC to establish a secure channel to the address found in the location hint of the group address. Once this channel is

established, the process uses the public key in the group address to verify that the process at the other end of this channel is a group member, by asking that process to sign a random challenge with the group private key. The process, now a client, communicates to the group via this member: for example, the client can multicast to the group by sending the message to the group member, which forwards the message to the group on behalf of the client.

The role of VSYNC in the above protocol is only to implement the secure point-to-point channel requested by the client process. As a part of this channel establishment, VSYNC informs both the client and the group member of the site at the other end of the secure channel, as well as the owner of the other process according to that site. This enables group members to make application-specific access control decisions based on this information. The key distribution protocol by which VSYNC establishes secure channels uses standard and well-known techniques, and will not be discussed here. Nor will the implementation of these channels be discussed, as it is simply a special case of the multicast channels discussed in Section 4.3.

4.5 Causal Ordering Properties

As mentioned in Section 2, Horus offers several different ordering guarantees on the delivery of group multicasts to group members. One of these is a *causal* ordering guarantee. Informally, if one multicast was initiated prior to another and could have caused the other to occur, then the first multicast is said to be *causally before* the second [Lamport 1978]. If desired, Horus will ensure that if one multicast is causally before another, then the former is delivered before the latter at any common destination. Moreover, these guarantees can be provided even if the multicasts occurred in different (overlapping) groups.

Reiter et al. [1992] discussed how these causal ordering guarantees raise security issues when causally related multicasts occur in overlapping groups, one of which contains a corrupted site. Depending on the causality detection protocol employed, it might be possible for the corrupt site to effect a violation of causality in the sequence of messages delivered to a process on a correct site in a group not containing the corrupt site. For instance, if groups G and H intersect, and H contains a corrupt site not in G, then that site might be able to cause a multicast in H to be delivered to a member in the intersection of G and H before a causally prior multicast in G. Reiter et al. also described how this could result in a violation of an application's security policy.

As part of this research, we have sought to better understand the importance to security of accurately detecting causal relationships despite malicious behavior, as well as possible attacks on attempts to detect causal relationships and ways to defend against them. The results of this effort are reported in Reiter and Gong [1993]; briefly, this article presents a framework within which these attacks can be examined, and several algorithms to prevent them in some situations. A variant of one of these algorithms is

already employed in Horus. A possible direction for future work in the system is to implement additional defenses against such attacks.

5. SUMMARY AND DISCUSSION

We have presented a security architecture for fault-tolerant systems. The architecture provides the programming abstraction of secure process groups, within which users can replicate applications in a protected fashion. The foundation of security in each process group is the group's keys, which are distributed to members during the group join protocol. These keys are used to communicate securely within the group and to authenticate group members to outsiders. The mechanisms for ensuring the authenticity and secrecy of group communication have been decoupled to allow the use of faster algorithms for each, and in particular, to exploit caching to accelerate encryption. Preliminary performance results are encouraging.

The group join protocol relies on authentication and time services that support cryptographic key distribution securely and fault tolerantly. We have chosen to replicate the authentication service, because by making the authentication service highly available, we gain greater flexibility in choosing certificate lifetimes and thus less reliance on a secure revocation mechanism than in approaches that use a nonreplicated service. To compensate for the security risks of this replication, the service is built to tolerate a minority of server corruptions.

The time service is not replicated, so that it is easier to protect. Moreover, its unavailability does not result in security breaches or hinder clients that continue to operate correctly. In fact, it could be temporarily taken offline for further protection if the need arises. While techniques exist for replicating time services so that some number of server corruptions could be tolerated, we have found that the additional costs of replication are difficult to justify.

By integrating our architecture into Horus, we have secured Horus' virtually synchronous execution model within each secure group. We have also identified security concerns that arise when applications employ causal ordering guarantees on the delivery of multicasts in multiple overlapping groups, and have elsewhere provided algorithms to address these concerns in some situations.

Changes to the Horus process group interfaces due to the security mechanisms are minimal, consisting only of additional routines to grant or deny join requests and additional options to some routines to indicate when communication should be encrypted. Thus, applications and group programming toolkits designed for the Horus interfaces should port easily to work over secure groups and thenceforth can be relied upon in a secure group that has not admitted a corrupt site or process. Such toolkits planned for Horus will facilitate primary-backup computations, data replication, automatic synchronization among group members, client-server computations, etc.

5.1 Status of the System

At the time of this writing, all of the mechanisms described in Sections 3 and 4 have been fully implemented, with the exception of the name service

discussed briefly in Section 4.1 and the user-level portion of the group client protocol discussed in Section 4.4. Presently we have a preliminary name service running with limited functionality. A name service with the described functionality is intended for development in the near future, as is the implementation of the user-level toolkit that includes the client protocol.

5.2 Related Work

To our knowledge, consideration of security issues unique to group-oriented systems first occurred in the design of the V kernel [Cheriton and Zwaenepoel 1985]. V supports a notion of process groups, but with weaker semantics than that of Horus. While V is not a security kernel and does not support key distribution or secure communication, V does make efforts to restrict group membership for security reasons. In principle, the security architecture proposed here could be integrated with V to further these efforts, although not without changes to some algorithms for using group keys.

Reiter [1994] documents research in the design and implementation of process groups that are not only secure against attacks from outside the group, but that are also resilient to the malicious corruption of group members. The Reiter [1994] algorithm enables correct members to control and observe changes to the group membership consistently, provided that fewer than one-third of the members in each instance of the group membership are corrupt.

There have also been attempts to address security issues in systems that support fault-tolerant computing using approaches other than process groups. One example is the Strongbox extension to the Camelot distributed transaction processing facility [Tygar and Yee 1991]. Strongbox provides mechanisms for mutually authenticating clients and servers and for encrypting communication between them.

In addition to our work outlined in Section 4.5 and detailed in Reiter and Gong [1993] and Reiter et al. [1992], another effort has identified the detection of causal relationships as being important to security and has attempted to provide defenses against attacks on efforts to detect causal relationships [Smith and Tygar 1993].

Work related specifically to the topic of fault-tolerant key distribution was outlined in Sections 3.1.2 and 3.2.2.

5.3 Future Work

One important area for future work in the area of fault-tolerant key distribution is in adapting the services described in Section 3 to very large systems. These services in their current form cannot scale to very large systems, for both security and performance reasons. In a very large system, the services may become overwhelmed, and there may not be a single authority trusted to protect them. To alleviate this, an instance of these services could be employed per administrative domain, as in Kent [1993], Lampson et al. [1992], and Steiner et al. [1988]. An alternative deployment of the authentication service would be to place each domain in charge of a different server. These directions hold many possibilities for future results.

A second possible direction for future work is the development of secure audit and recovery procedures that are compatible with our security architecture. In particular, because all group members possess the group keys for a group, it is not presently possible to audit reliably the actions of particular group members within the group. Additionally, in the event of the penetration of a site, the groups containing that site should ideally be informed of that penetration once it is detected. Our system presently provides no automatic measures for this.

There are several other issues that we have not attempted to address in this work but that should be addressed in systems in which this technology is employed. Examples include securely booting sites and user authentication. To aid in booting sites we have built a *boot server*, which provides to each site its initial keys using a secret *boot key* that must be provided to the site by an operator. (Alternatively, these initial keys can be stored on the site in nonvolatile memory [Lampson et al. 1992].) However, we do not take measures to verify the integrity of the operating system or Horus when the machine is booted; for one approach to doing this, see Lampson et al. We view these issues, and other intranode issues such as protected address spaces and local interprocess communication, as more general operating system security issues that have not been goals of this work.

We have also not attempted to address the issue of user authentication. However, the mechanisms described in this article facilitate several solutions to this problem. For instance, the authentication service of Section 3.2 and our boot service could easily be extended to provide user certificates and users' private keys, respectively, similar to the Certificate Distribution Centers and Login Enrollment Agent Facility of SPX [Tardo and Alagappan 1991]. The secure channels provided by our architecture also facilitate simpler password-based user authentication mechanisms.

ACKNOWLEDGMENTS

We thank Brad Glade for commenting on an early version of this article, and especially for suggesting the inclusion of the timestamp in the first message of the protocol of Figure 3. We are also grateful to Tushar Chandra, Fred Schneider, Stuart Stubblebine, Mark Wood, and the anonymous referees for providing helpful comments.

REFERENCES

AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. 1992. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*. IEEE, New York, 76–84.

BELLOVIN, S. M., AND MERRITT, M. 1990. Limitations of the Kerberos authentication system. *Comput. Commun. Rev. 20*, 5 (Oct.), 119–132.

BIRMAN, K. P., AND JOSEPH, T. A. 1987a. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Symposium on Operating Systems Principles*. ACM, New York, 123–138.

BIRMAN, K. P., AND JOSEPH, T. A. 1987b. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 1 (Feb.), 44–76.

BIRMAN, K. P., SCHIPER, A., AND STEPHENSON, P. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9*, 3 (Aug.), 272–314.

CCITT. 1988. The directory—authentication framework, Recommendation X.509. International Telegraph and Telephone Consultative Committee, Geneva, Switzerland.

CHERITON, D. R., AND ZWAENEPOEL, W. 1985. Distributed process groups in the V kernel. *ACM Trans. Comput. Syst. 3*, 2 (May), 77–107.

CRISTIAN, F. 1989. Probabilistic clock synchronization. *Distrib. Comput. 3*, 3, 146–158.

DENNING, D. E., AND SACCO, G. M. 1981. Timestamps in key distribution protocols. *Commun. ACM 24*, 8 (Aug.), 533–536.

DEPARTMENT OF DEFENSE. 1985. Department of Defense trusted computer system evaluation criteria. DOD 5200.28-STD, Washington, D.C.

DESMEDT, Y., AND FRANKEL, Y. 1992. Shared generation of authenticators and signatures. In *Advances in Cryptology—CRYPTO '91 Proceedings*, J. Feigenbaum, Ed. Lecture Notes in Computer Science, vol. 576. Springer-Verlag, New York, 457–469.

DIFFIE, W., AND HELLMAN, M. E. 1979. Privacy and authentication: An introduction to cryptography. *Proc. IEEE 67*, 3 (Mar.), 397–427.

GALVIN, J. M., MCCLOGHRIE, K., AND DAVIN, J. R. 1991. Secure management of SNMP networks. In *Integrated Network Management*. Vol. 2. Elsevier Science Publishers B.V. (North-Holland), Amsterdam.

GONG, L. 1993. Increasing availability and security of an authentication service. *IEEE J. Sel. Areas Commun. 11*, 5, (June), 657–662.

GONG, L. 1992. A security risk of depending on synchronized clocks. *ACM Oper. Syst. Rev. 26*, 1 (Jan.), 49–53.

GONG, L. 1989. Using one-way functions for authentication. *Comput. Commun. Rev. 19*, 5 (Oct.), 8–11.

GUSELLA, R., AND ZATTI, S. 1984. TEMPO—A network time controller for a distributed Berkeley UNIX system. In *Proceedings of the USENIX Summer Conference*. USENIX Assoc., Berkeley, Calif., 78–85.

HERLIHY, M. P., AND TYGAR, J. D. 1988. How to make replicated data secure. In *Advances in Cryptology—CRYPTO '87 Proceedings*, C. Pomerance, Ed. Lecture Notes in Computer Science, vol. 293. Springer-Verlag, New York, 379–391.

JOSEPH, M. K. 1987. Towards the elimination of the effects of malicious logic: Fault tolerance approaches. In *Proceedings of the 10th NBS/NCSC National Computer Security Conference*. NBS/NCSC, Baltimore, Md., 238–244.

KAASHOEK, M. F. 1992. Group communication in distributed computer systems. Ph.D. thesis, Vrije Universiteit, The Netherlands.

KENT, S. T. 1993. Internet privacy enhanced mail. *Commun. ACM 36*, 8 (Aug.), 48–60.

LACY, J. B., MITCHELL, D. P., AND SCHELL, W. M. 1993. CryptoLib: Cryptography in software. In *Proceedings of the 4th USENIX Security Workshop*. USENIX Assoc., Berkeley, Calif., 1–17.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July), 558–565.

LAMPSON, B., ABADI, M., BURROWS, M., AND WOBBER, E. 1992. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst. 10*, 4 (Nov.), 265–310.

MARZULLO, K. 1990. Tolerating failures of continuous-valued sensors. *ACM Trans. Comput. Syst. 8*, 4 (Nov.), 284–304.

MILLS, D. L. 1989. RFC 1119: Network Time Protocol (version 2) specification and implementation. Internet Activities Board. Sept.

NATIONAL BUREAU OF STANDARDS 1977. Data encryption standard. Federation Information Processing Standards Publication 46, Government Printing Office, Washington, D.C.

NEEDHAM, R. M., AND SCHROEDER, M. D. 1978. Using encryption for authentication in large networks of computers. *Commun. ACM 21*, 12 (Dec.), 993–999.

PETERSON, L. L., BUCHHOLZ, N. C., AND SCHLICHTING, R. D. 1989. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst. 7*, 3 (Aug.), 217–246.

REITER, M. K. 1994. A secure group membership protocol. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*. IEEE, New York, 176–189.

REITER, M. K., AND BIRMAN, K. P. 1994. How to securely replicate services. *ACM Trans. Program. Lang. Syst. 16*, 3 (May), 986–1009.

REITER, M. K., AND GONG, L. 1993. Preventing denial and forgery of causal relationships in distributed systems. In *Proceedings of the 1993 IEEE Symposium on Research in Security and Privacy*. IEEE, New York, 30–40.

REITER, M. K., BIRMAN, K. P., AND GONG, L. 1992. Integrating security in a group oriented distributed system. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*. IEEE, New York, 18–32.

RIVEST, R. L. 1991. The MD4 message digest algorithm. In *Advances in Cryptology—CRYPTO '90 Proceedings*, A. J. Menezes and S. A. Vanstone, Eds. Lecture Notes in Computer Science, vol. 537. Springer-Verlag, New York, 303–311.

RIVEST, R. L., AND DUSSE, S. 1992. RFC 1321: The MD5 message-digest algorithm. Internet Activities Board.

RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (Feb.), 120–126.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec.), 299–319.

SIMONS, B., WELCH, J. L., AND LYNCH, N. 1990. An overview of clock synchronization. In *Fault-Tolerant Distributed Computing*, B. Simons and A. Spector, Eds. Lecture Notes in Computer Science, vol. 448. Springer-Verlag, New York, 84–96.

SMITH, S., AND TYGAR, J. D. 1993. Signed vector timestamps: A secure protocol for partial order time. Tech. Rep. CMU-CS-93-116, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa.

STEINER, J. G., NEUMAN, C., AND SCHILLER, J. I. 1988. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Conference*. USENIX Assoc., Berkeley, Calif., 191–202.

STUBBLEBINE, S. G., AND GLIGOR, V. D. 1992. On message integrity in cryptographic protocols. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*. IEEE, New York, 85–104.

TARDO, J. J., AND ALAGAPPAN, K. 1991. SPX: Global authentication using public key certificates. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*. IEEE, New York, 232–244.

TSUDIK, G. 1992. Message authentication with one-way hash functions. In *Proceedings of IEEE INFOCOM '92*. IEEE, New York, 2055–2059.

TUCHMAN, W. 1979. Hellman presents no shortcut solutions to the DES. *IEEE Spectrum 16*, 7 (July), 40–41.

TURN, R., AND HABIBI, J. 1986. On the interactions of security and fault-tolerance. In *Proceedings of the 9th NBS/NCSC National Computer Security Conference*. NBS-NCSC, Baltimore, Md., 138–142.

TYGAR, J. D., AND YEE, B. S. 1991. Strongbox. In *Camelot and Avalon, A Distributed Transaction Facility*, J. L. Eppinger, L. B. Mummert, and A. Z. Spector, Eds. Morgan Kaufmann, San Mateo, Calif., 381–400.

VAN RENESSE, R., BIRMAN, K., COOPER, R., GLADE, B., AND STEPHENSON, P. 1992. Reliable multicast between microkernels. In *Proceedings of the USENIX Microkernels and Other Kernel Architectures Workshop*. USENIX Assoc., Berkeley, Calif.

VOYDOCK, V. L., AND KENT, S. T. 1983. Security mechanisms in high-level network protocols. *ACM Comput. Surv. 15*, 2 (June), 135–171.

WOBBER, E., ABADI, M., BURROWS, M., AND LAMPSON, B. 1993. Authentication in the Taos operating system. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, New York, 256–269.

Received August 1993; revised April 1994; accepted June 1994