A Security Architecture for Fault-Tolerant Systems*

Michael Reiter
Kenneth Birman
Robbert van Renesse

TR 93-1354
(replaces TR 93-1325)
June 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported under DARPA/ONR grant N00014-92-J-1866, and by grants from GTE, IBM, and Siemens, Inc. Any opinions, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views or decisions of the ONR.

A Security Architecture for Fault-Tolerant Systems*

Michael Reiter
reiter@cs.cornell.edu

Kenneth Birman
ken@cs.cornell.edu

Robbert van Renesse
rvr@cs.cornell.edu

Department of Computer Science
Cornell University
Ithaca, New York 14853

June 3, 1993

Abstract

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 novel techniques. We detail the design and implementation of these services and the secure process group abstraction they support. We also give performance figures for some common group operations.

*This work was supported under DARPA/ONR grant N00014-92-J-1866, and by grants from GTE, IBM, and Siemens, Inc. Any opinions, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views or decisions of the ONR.

1 Introduction

There exists considerable experience with addressing the needs for security and fault-tolerance individually in distributed systems. However, much less is understood about how to simultaneously address these needs in a single, integrated solution. Indeed, the goals of security and availability have traditionally been viewed as being in conflict, because the only generally feasible technique for making data and services highly available, namely replicating them, also makes them inherently harder to protect [HT88, LABW92].

In this paper we present the design and implementation of a security architecture for fault-tolerant systems, which 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 [CZ85, BJ87b, KT91, ADKM92]—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 integrated this architecture into a new version of the Isis distributed programming toolkit [BJ87b, BSS91] called Horus, thereby securing Horus' *virtually synchronous* process group abstraction. An earlier paper [RBG92] 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 architecture provides a framework that enables the user to balance this tradeoff for each application individually. The framework consists of *secure process groups* within which a user can efficiently replicate applications in a protected fashion. Authentication and access control mechanisms enable the group members to prevent untrusted processes from joining. And, provided that the members admit only processes on trustworthy sites, the members will enjoy secure communication and correct group semantics among themselves. Thus, the conflict between security and availability at this level of the system is passed to the user by enabling him or her to control where and how widely each application is replicated.

The second level at which this conflict is addressed is in the core security services that underlie these abstractions, 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.

In section 2 we present this approach and compare it to other possible designs. Because these techniques are applicable in a wide range of systems, in section 2 we treat them outside the context of our larger security architecture. A description of their use in our system is deferred until section 3, where we detail the design and implementation of the secure process group abstraction and provide performance numbers for some common operations. Finally, we summarize and discuss related and future work in section 4.

2 Fault-tolerant key distribution

In open networks, an intruder can attempt to initiate spurious communication in two ways [VK83]: 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 (e.g., [NS78, DS81, OR87, SNS88]). 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 communication can be protected or a shared key can be negotiated.

Authentication protocols typically employ a trusted service, commonly called an *authentication service* [NS78], 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. Public key authentication services of this form are also called *certification authorities* [BAN89].

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 [DS81]. Timestamp-based replay detection has been used in several systems (e.g., [SNS88, TA91]) and is often preferable to challenge-response techniques [NS78], 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 [GZ84, Mil89].

The dependence of authentication protocols on authentication and time services raises troubling security and availability issues. First, the assurances provided by authentication protocols directly rely 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, in many environments this is dangerous, 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.

2.1 The time service

The security risks of clock synchronization failures in authentication protocols are well-known [DS81, Gon92], and the need for a *secure* time service has been recognized in several systems (e.g., [Mil89, BM90]). 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, this in itself 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.

2.1.1 The algorithm

Clients interact with our time service by the simple RPC-style protocol shown in figure 1. We assume that the time server possesses a private key $K_{\mathcal{T}}$ whose corresponding public key is well-known. (There is a similar shared-key protocol.) At regular intervals, a client queries the time service with a *nonce identifier* N [NS78], a new, unpredictable value. When the time server receives this request, it immediately generates a timestamp T equal to its current local clock value and replies with $\{N, T\}_{K_{\mathcal{T}}}$, i.e., the nonce and the timestamp, both signed with $K_{\mathcal{T}}$. The client considers the response valid if it contains N and can be verified with the public key of the time service.

Figure 1: Protocol by which client C interacts with time service $\mathcal{T}$

$$\begin{aligned} C \rightarrow \mathcal{T} &: \; N \\ \mathcal{T} \rightarrow C &: \; \{N, T\}_{K_{\mathcal{T}}} \end{aligned}$$

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,

$$(1-\rho)(t-t') \leq H(t) - H(t') \leq (1+\rho)(t-t'). \tag{1}$$

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.[1]

2. *The time server's clock is perfectly synchronized to real time.* We could incorporate time server drift into our formulas, although 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 via a dedicated bus. Alternatively, the time service could be viewed as *defining* time in the system.

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.* 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 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 $0 \leq \alpha_2 \leq R - min_1 - min_2$, and so after the client verifies the response, real time $\hat{t} = T + min_2 + \alpha_2$ is in the range

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

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

[1] Keith Marzullo has suggested the possibility of dynamically measuring ρ on a per-client basis [Mar93]. However, we do not pursue this here.

By (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* [SS75], in the following sense:

Theorem 2 *A 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, each client periodically 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 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 continue to operate without the time service, calculations in our system indicate that this bound is not very tight. As an example, consider two principals P_1 and P_2, each of whose clocks are 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 is 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 .5 seconds. Then, even if the clocks of P_1 and P_2 drift away from one another 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 still be 20.4 hours 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 in [DS81]. In addition, because clocks typically do not drift away from one another at the maximum possible rate, tests in our system have shown that in reality the unavailability of the time service 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. More comprehensive testing of this hypothesis is currently underway.

2.1.2 Comparison to alternative designs

We derived our algorithm from that presented in [Cri89] for implementing a time service. The primary difference between ours and that in [Cri89] 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 $\hat{t}$, as 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 in such a way that it would provide a correct service despite some server failures and corruptions. For instance, a client could use the robust averaging algorithm of [Mar90] to obtain an interval of bounded inaccuracy containing real time from a set of n time servers, provided that fewer than $\lfloor n/3 \rfloor$ servers are faulty or corrupt. On the other hand, such approaches place larger burdens on the administrator of the service than does ours, because the administrator must protect multiple servers, instead of only one, to ensure the integrity of the service. Since the availability of the time service is not crucial, this burden and the additional costs of replication are difficult to justify.

Numerous other approaches to clock synchronization have been proposed, but for brevity, we do not discuss them all here. Unlike ours, however, most assume upper bounds on message transmission times, and to our knowledge, none provide a fail-safe algorithm for estimating time in authentication

[2]This is a simplification of the algorithm in [Cri89]; the actual algorithm also takes measures to ensure that client clocks are continuous and monotonic. These features, however, are unimportant for our purposes.

protocols. We thus believe that our approach is unique in providing this property under relatively few assumptions.

2.2 The authentication service

Like those in [TA91, LABW92], our authentication service is of the public key variety, that produces public key *certificates* for principals. Each certificate $\{P, T, K_P^{-1}\}_{K_A}$ contains the identifier P of the principal, the public key K_P^{-1} of the principal, and the *expiration* time T of the certificate, all signed by the private key K_A 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 [LABW92]. 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, as its unavailability would prevent certificates from being 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 (e.g., [LABW92, Gon93]). We first describe our method of balancing this tradeoff, and then compare it to other alternatives.

2.2.1 The algorithm

In [RB92], we describe a technique for securely replicating any service that can be modeled as a state machine. The technique is similar to *state machine replication* [Sch90], 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 in [RB92] 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 response, just as if the service were 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:

1. 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, and
2. 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.

The replication technique does not rely on any particular threshold signature scheme. For our authentication service, we have implemented the one in [DF92], which is based upon RSA [RSA78].

Given a (k,n)-threshold signature scheme, we build our authentication service as follows. Let $\mathcal{A} = \{AS_1, \ldots, AS_n\}$ be the set of servers. We first choose a threshold k and create n shares from the private key $K_{\mathcal{A}}$ of the authentication service. Each authentication server AS_i, when started, is given the i-th share of $K_{\mathcal{A}}$, its own private key K_{AS_i}, the public key corresponding to the private 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 2.1.1.

The protocol by which clients obtain certificates from the authentication service is shown in figure 2. 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 how C chooses T below. 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 $pr_i(P, T+\Delta, K_P^{-1})$ for the contents $(P, T+\Delta, K_P^{-1})$ of P's certificate, where Δ is the predetermined lifetime of the certificate. AS_i then sends $pr_i(P, T+\Delta, K_P^{-1})$ 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 $\{P, T+\Delta, K_P^{-1}\}_{K_{\mathcal{A}}}$, it sends the certificate to C. C accepts the first properly signed certificate for P with an expiration time sufficiently far in the future (see below), and ignores any other replies.

It is easy to see why this protocol provides the Integrity and Availability guarantees stated above. 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.

[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.

Figure 2: Protocol by which client C obtains a certificate for principal P

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

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 2.1.1), 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 $L(t)$, where t is the time at which the server receives the request; so, 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.

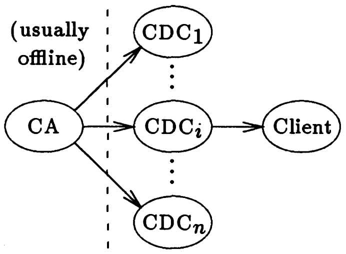
In practice, it works well to have a client, when sending a request at time t, to set T to its own value of $L(t)$ minus a small offset $\delta \geq 0$, and to increase δ on subsequent requests if prior attempts to obtain a certificate failed. Because an unavailability of the time service will generally 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.

2.2.2 Comparison to alternative designs

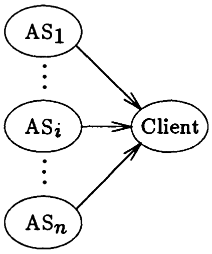
As previously mentioned, we are not the first to notice the conflict between security and availability in the construction of authentication services. In [Gon93], Gong proposed a method for dealing with this tradeoff in shared key authentication services such as Kerberos [SNS88]. Lampson, et.al., [LABW92] described a different solution that is appropriate for a public key authentication service similar to ours, which they call a *certification authority*.

In the latter solution, which is also implemented in SPX [TA91], the certification authority is usually offline, where it can be more easily protected by physical means (figure 3a). To reduce the impact of its limited availability, it produces long-lived certificates that are stored in an online *certificate distribution center* (CDC), which can be replicated for high availability [TA91]. Certificates

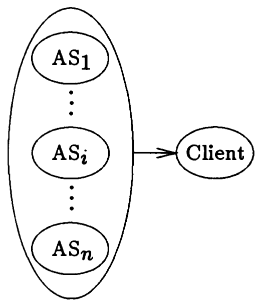
Figure 3: Design alternatives for a fault-tolerant public key authentication service



(a) Offline certification authority (CA) [TA91, LABW92]. CA deposits long-lived certificates in an online, replicated certificate distribution center (CDC).



(b) State machine replication [Sch90]. Client authenticates certificate from each server and accepts key, if any, that occurs in a majority of certificates.



(c) Transparent state machine replication [RB92]. Client authenticates single certificate that could have been created only by a majority of servers.

are obtained only from CDC replicas, so if necessary, a certificate can be revoked by deleting it from all replicas. Thus, a client accepts a certificate only if both the highly secure certification authority and a CDC replica endorse it. The disadvantage of this scheme, noted in [LABW92], is that the corruption of a CDC replica could delay the revocation of a certificate.

This problem could be addressed by using our technique described in [RB92] to *securely* replicate the CDC. However, our approach of securely replicating the authentication service itself (figure 3c) 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 vulnerability 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. 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 that public key are produced.

Our technique also has advantages over state machine replication [Sch90] (figure 3b) of the authentication service (or the CDC of [TA91, LABW92]). First, our approach requires less state at the client: the client needs only a single public key for the service, and need not maintain secure channels to the servers or retain any other replies but the first properly signed one. The last of these is especially beneficial if a principal obtains and forwards its own certificate to its partners in cryptographic protocols, as in the "push" technique described in [LABW92]. If the service were implemented using state machine replication, the principal would need to collect and forward a number of certificates equal to the size of a majority of the servers. Second, our technique requires the client to authenticate less communication, as the client can ignore all replies after the first one it accepts as valid. Third, in our approach the configuration of the authentication 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 in our scheme. As will be illustrated in section 3, 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.)

3 Secure process groups

As discussed in section 1, our authentication and time services have been implemented as part of a comprehensive security architecture for fault-tolerant systems [RBG92]. The architecture was developed in an effort to integrate security into the Horus system. Like its predecessor Isis [BJ87b, Bir93], Horus is a system for building fault-tolerant applications using the process group model of computation. While the security architecture has been implemented in Horus, we expect that its integration into Isis will be straightforward, and in principle it could also be applied in other group-oriented systems such as V [CZ85], Amoeba [KT91], and Transis [ADKM92].

In this section, we discuss the design, implementation and performance of the architecture in Horus. We begin by describing the programming model of *secure process groups* that is supported by the security architecture, with an emphasis on the security guarantees that augment the Horus process group abstraction. We then discuss several aspects of the secure group implementation: the cryptographic keys used in a group, the group join protocol, and secure group communication. As group communication constitutes the vast majority of activity in most Isis applications today (and thus, we expect, in most Horus applications in the future), we also compare the performance of group communication for the secure and insecure versions of the system.

3.1 Programming model

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. Horus further supports the model of *virtual synchrony* [BJ87a], so that message deliveries and notification of group membership changes (i.e., changes to the *group view*) appear atomically and in a consistent order at all group members, even when failures occur.

The 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 elect to not admit the joiner, they can deny the request, in which case the joiner will not be admitted. 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 corrupted 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, as an untrustworthy process could admit a corrupted site to the group.) Secrecy can also be requested by group members, in which case their messages will be encrypted before being sent, in an effort to prevent their disclosure to a network intruder. Secure point-to-point communication is also supported both within and outside of groups, although in this paper we discuss only secure group communication.

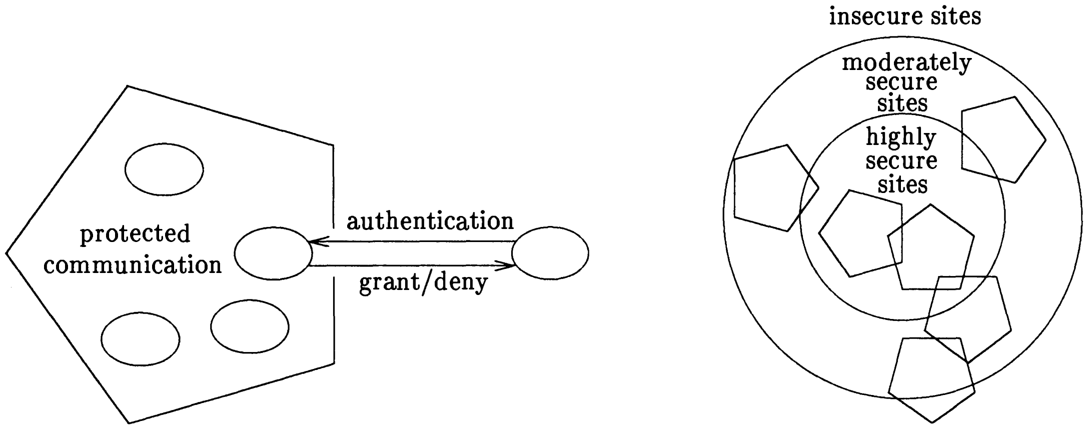
The programming model thus presented to the programmer is one in which each process group can be viewed as a "fortress", where admission to this fortress is regulated by the group members themselves (see figure 4a). Provided that the members admit only processes on trustworthy sites, the normal Horus semantics are guaranteed within the group. Moreover, these guarantees are achieved with minimal changes to the process group interface. So, tools and applications designed for the Horus interfaces should be able to be ported to secure groups easily.

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. Such a service could be composed of a single secure group located on a small "island" of trustworthy sites. Alternatively, in a larger service where greater internal control is also required, the service could be implemented using many secure groups, arranged to enforce security policies within the service and to limit the damage to the overall service from the corruption of a site (see figure 4b). 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.

3.2 Implementation

The security architecture as implemented in Horus is illustrated in figure 5. On each site, the core Horus functionality is implemented in a transport layer entity called MUTS and a session layer entity

Figure 4: Secure process groups
(= process; = secure group)



(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, passive network attacks.

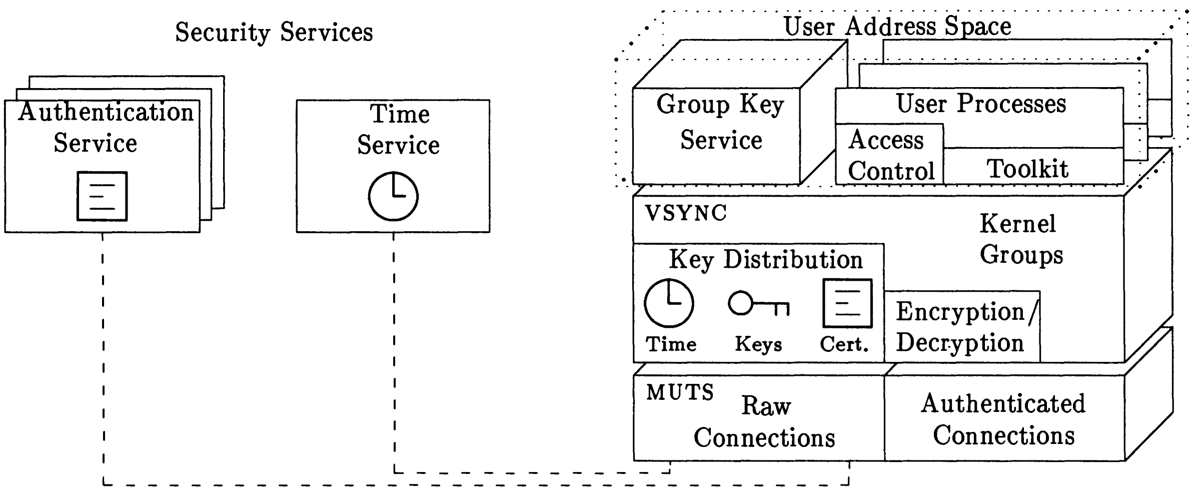
(b) Applications can be built from many secure groups to enforce internal security policies and to limit damage from site corruptions. A group can span sites secured to different levels, but is only as secure as the least secure site admitted.

called VSYNC, both of which reside in the operating system kernel on most platforms [vRBC+92]. 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 a set of user-level libraries and tools to be linked with user programs to facilitate primary-backup computations, support replicated data, etc. While security mechanisms are included at all levels of the system, the core functionality for the security architecture lies in the MUTS and VSYNC layers. These layers have been augmented with protocols for distributing and using *group keys*.

3.2.1 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 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

Figure 5: The Horus security architecture



in the VSYNC layer and are never held by user processes.

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, which is intended to be used for the entire lifetime of the group. In our implementation, the communication key consists of two DES [DES77] keys (112 bits), and encryption is performed with it using the triple encryption technique of [Tuc79].

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 called the *identification key* (or "group private key") of the group, because it is primarily used to identify, or 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. Again, identification keys are held within VSYNC, although group members can obtain signatures on messages through a VSYNC interface. Like the communication key, the identification key is intended to be used for the lifetime of the group. In the present implementation, the size of group private keys is a compile-time constant (we typically use 512-bit RSA moduli), although in the future we intend to 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. In addition, VSYNC creates and returns the group's address, which contains, among other things, the public key and a location hint for the group. Overall latency of the group creation is minimal unless the VSYNC cache is empty, which could happen, e.g., if the creation is requested immediately 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, replicated, 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 altering 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 in [RBG92].

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 [Sch90] or [RB92]. 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. In addition, we permit multiple name services within our architecture, opening the possibility that a highly secure name service could exist side-by-side with less secure name services.

To minimize the risk of exposing a group's keys, only those sites that need them—i.e., the sites of current group members—should possess them at any given time. Thus, if a site leaves a group, it destroys 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.

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

3.2.2 The group join protocol

Once a process has obtained the group address for a group, it can contact the group either to request a service or to join the group. In the former case, the process can become a *client* of the group via a user-level protocol that forces a group member to prove its membership in the group. VSYNC does not know about this protocol or the notion of clients of groups.

To join a group, however, VSYNC must be involved to obtain the group keys. In preparation for group joins, each site A is booted with the public keys of the authentication and time services described in section 2, and its own private key K_A; the authentication service possesses the corresponding public key K_A^{-1}. VSYNC periodically synchronizes 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 [SG92].)

Figure 6: VSYNC protocol by which site A joins group G, containing site B

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

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 56-bit DES *reply key* RK, both encrypted under the public key of the group (i.e., $[RP, RK]_{K_G^{-1}}$); 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]

[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.

All of this information is signed with K_A and preceded by A's certificate. A sends this message to a site B that, based on the location hint in the group address, it believes to be in the group. (Safety of this protocol does not rely on this hint being accurate. If it is incorrect, however, the requesting process will need to obtain a more current address for the group, e.g., through the name service mentioned in section 3.2.1. For the remainder of this discussion, we assume that site B actually is in the group.)

2. When B 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 2.1.1 and 2.2.1, respectively. If the message is determined to be valid, B uses the group private key K_G 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.

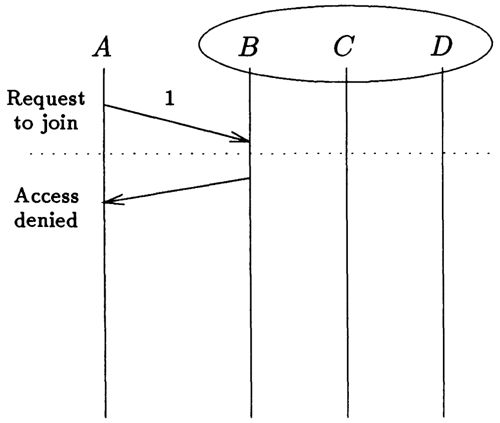
3. The local member can take any measures including, e.g., executing protocols with other group members or the requesting process, to determine whether the join should be granted. In particular, if the member does not trust A to have authenticated the owner of the process properly, then it should possibly 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 ($\otimes$), and the group private key K_G 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.

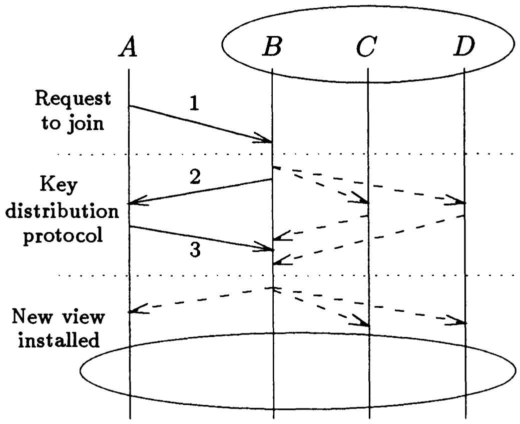
5. After A authenticates B's reply and notes its own identity in the message, it acknowledges the keys by returning N signed with RK. Note that an attempt by a network intruder to replay a message in place of B's reply will be detected because RK is a new (i.e., "fresh") key.

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^{-1} and K_G for these operations. First, the cryptographic operations with RP and RK are more efficient 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^{-1}, the corruption of A would reveal K_A 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.

Figure 7: Overview of the VSYNC group join protocol



(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.

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 [TA91, LABW92], the CDC) *is* on the critical path of authentication protocols. Second, the transparency of replication in the authentication service simplifies the protocol. If, e.g., state machine replication were used, each site would need to maintain certificates from a majority of servers to prepend to its requests. This would also result in a substantial computational overhead for authenticating these certificates.

The latency of a group join, 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 used[6]: with the modular exponentiation operations removed, the cost of a group join drops to 195ms on average. Clearly hardware support for modular exponentiation would be of value for applications in which group membership is very dynamic. However, experience with Isis (see [BSS91]) leads us to believe that most Horus applications will

[6]In these tests we used the C implementation of modular exponentiation 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 faster software implementations of modular exponentiation are available from RSA Data Security, Inc. and others. However, with any software implementation of which we are aware, the cost of modular exponentiation would continue to be a limiting factor in the performance of the group join protocol.

employ largely static groups, and so we do not expect that most applications will require such hardware to use the security architecture effectively.

3.2.3 Secure group communication

As discussed at the beginning of this section, 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 be encrypted,[7] 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 [Riv91] or MD5 [RD91]. 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.

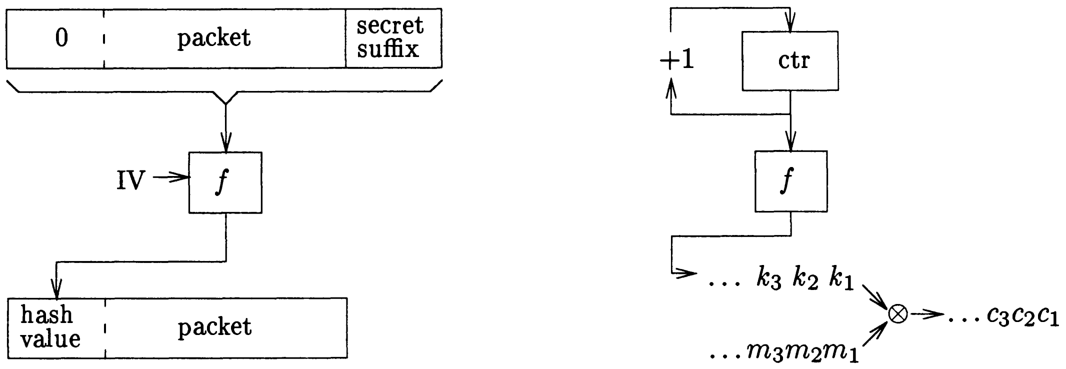
MUTS packet authentication. Messages, or more accurately, MUTS packets, are authenticated on a per-connection basis. A MUTS connection is a logical path from one MUTS instance to others. Only the instance that opened the connection can send data across it, although recipients on a connection also acknowledge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

The first packet sent on a connection includes a *connection key*, encrypted under the communication key of the group in which the connection is opened. (This packet also contains a timestamp to allow the recipients to detect replay attacks, as described in section 2.1.1.) The connection key consists of a 16-byte *initialization vector* (IV) and a *suffix* whose size is a compile-time constant (typically 16 bytes). After the sender prepares a subsequent packet to send over the connection, it initializes the internal state of the hash function f to IV and then applies f to the packet and the

[7] We make no effort to address *traffic analysis* attacks [VK83].

suffix. (Alternatively, the IV can be replaced by a secret *prefix* that is processed by f before the packet [Tsu92].) The result is placed in the packet header, as shown in figure 8a. The 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 valid.

Figure 8: Cryptographic operations



(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 XOR-ing 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.

To our knowledge, this form of message authentication was first proposed in [Tsu92], although a variation of this approach was employed in [GMD91] and a similar use of one-way hash functions in authentication protocols was proposed in [Gon89]. 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 more conventional techniques that use encryption is speed, because fast one-way hash functions are typically much faster than encryption functions (e.g., MD4 in software runs at over three times the rate of the fastest software DES implementations [LABW92]).

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* [DH79].

In a stream cipher, the ciphertext of a message $m = m_1 m_2 \ldots$ is obtained by enciphering the

i-th element m_i with the i-th element k_i of a secret *key stream* $k_1k_2\ldots$ 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 [DH79]: 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 8b). Provided that the counter is initialized to a secret, random value and f is sufficiently strong, knowledge of one portion of the key stream will not reveal other portions. That is, the key for producing $k_1k_2\ldots$ is the initial value of the counter. The i-th bit c_i of the ciphertext is simply the exclusive-or of k_i and m_i (i.e., $c_i = k_i \otimes m_i$).

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 random 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 i-th site in the group encrypts a message to the group using the key stream generated from the i-th counter. Similarly, sites decrypt messages from the i-th site by exclusive-oring the next portion of the i-th 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. In addition, 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 cipher for each site synchronized at all sites in the group between view changes.

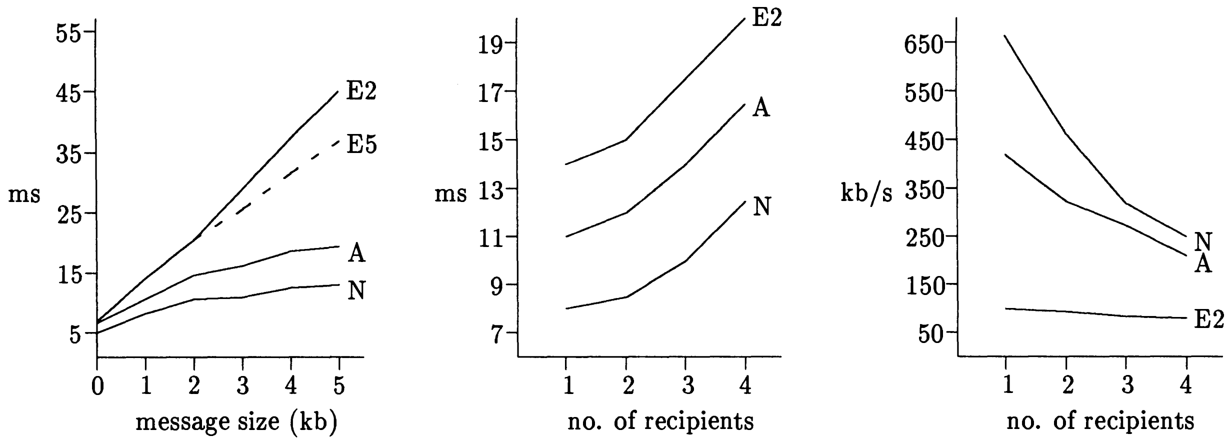
An advantage of synchronous stream ciphers over other encryption methods is that they allow much of the work for encryption to be done in the background. In our system, we precompute and cache portions of the key stream for each site in a group so that the key stream is immediately available for use. This reduces the encryption of a message smaller than the cache to simply an exclusive-or over the length of the message. The size of the cache for each site in a group is presently a compile-time constant, although in the future we intend to make this size dynamically adjustable.

Performance. Preliminary performance figures for group communication are pictured in figure 9. In each graph, the line labeled "N" indicates performance when no security mechanisms are employed. The line labeled "A" indicates performance when packets are authenticated by MUTS. The lines labeled "E2" and "E5" indicate performance when packets are authenticated by MUTS and the sender's message is encrypted by VSYNC. In the E2 tests, both the sender and the receivers maintained a 2 kilobyte (kb) cache of precomputed bits for encryption and decryption; in the E5 tests, these caches were of size 5kb. These tests used MD4 for f and were performed between user processes on moderately loaded 33MHz Sparc ELC workstations running SunOS 4.1.1.

In parts (a) and (b) of figure 9 are average latencies for member-to-group RPC interactions. In a member-to-group RPC, one sender multicasts a single message to the membership of the group, and all members acknowledge (with a null message). The latency is the measured time at the sender between sending the initial message and receiving all acknowledgements. Each data point in parts (a) and (b) is the mean latency of 100 consecutive member-to-group RPCs performed as quickly as possible. 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 1 megabyte (Mb) member-to-group RPC and dividing 1Mb by the time required for this RPC to complete.

Figure 9: Performance of group communication

N = no security; A = authenticated only;
E2 = authenticated & encrypted, 2kb cache;
E5 = authenticated & encrypted, 5kb cache



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

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

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

Two items are worth noting about the graphs in figure 9. First, in part (a) the rapid rise of E2 relative to E5 beginning after the 2kb message is due to the fact that in the E2 tests, only 2kb of the encryption key stream was precomputed prior to the message send. So, while the cache contents were sufficient to encrypt and decrypt the 2kb message at the sender and receiver, respectively, the 3kb message exhausted these caches and forced both to generate parts of the key stream before sending or delivering the message. Similarly, in part (c) the large impact on bandwidth due to encryption is partly due to the immediate exhaustion of the cached key stream when sending very large (in these tests, 1Mb) messages.

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 are also pursuing other optimizations to the performance of group communication. We are continuing to optimize the cryptographic mechanisms of our implementation. In addition, we intend to incorporate flow control mechanisms into MUTS to improve performance. At the present time, MUTS provides little flow control, and so in the tests of figure 9, we had to use small packet sizes and frequent acknowledgements to prevent packets from being dropped by the operating system. We anticipate that flow control mechanisms will improve the performance of MUTS transport substantially. Another important aspect of our effort to achieve better performance is integrating the system into microkernel-based operating systems.

4 Summary and discussion

In this paper we have presented a security architecture for fault-tolerant systems. An integral part of this system is an authentication and a time service that securely and fault-tolerantly support key distribution. In our system, we have chosen to replicate only the authentication service, because unlike with the time service, substantial benefits are obtained by making the authentication service highly available. To compensate for the potential security risks of this replication, the service is built to tolerate a minority of server corruptions and failures.

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.

Together these services are used to securely distribute group keys in support of a secure process group abstraction. This abstraction provides a means to replicate applications in a protected fashion. Authentication of group members and protection of group communication are achieved through the use of group keys. These are distributed to processes during the group join protocol, which employs the authentication and time services previously described. The mechanisms for ensuring the authenticity and secrecy of group communication have been decoupled to allow the use of faster algorithms for each. Preliminary performance results are encouraging.

By integrating our architecture into the Horus system, we have secured Horus' virtually synchronous process group abstraction. In addition, changes to the Horus process group interfaces due to the security mechanisms are minimal, consisting only of additional options to some routines to indicate when communication should be encrypted and routines to grant or deny join requests. As a result, applications and group programming toolkits designed for the Horus interfaces should be able to be ported 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, and so forth.

4.1 Status of the system

The Horus system, including the security architecture described in this paper, is scheduled for release to the public later this year. At the time of this writing, all of the mechanisms described in sections 2 and 3 have been fully implemented, with the exception of the name service discussed briefly in section 3.2.1. We presently have a preliminary name service running with limited functionality. A name service with the described functionality is intended for development in the near future. In addition to working on this and other components of the system not described here, we are working to improve the performance of the system as discussed in section 3.2.3.

4.2 Related work

To our knowledge, consideration of security issues unique to group-oriented systems first occurred in the design of the V kernel [CZ85]. 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, e.g., 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.

There has 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 [TY91]. Strongbox provides the mechanisms for mutually authenticating clients and servers and for encrypting communication between them.

Work related specifically to the topic of fault-tolerant key distribution was outlined in sections 2.1.2 and 2.2.2.

4.3 Future work

One important area for future work in the area of fault-tolerant key distribution is adapting the services described in section 2 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 [SNS88, LABW92]. An alternative deployment of the authentication service would be to place each domain in charge of a different server.

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; some of these may be pursued in the future. 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 installed on the site in non-volatile storage [LABW92].) 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 [LABW92]. We view these issues, and other intra-node issues such as protected address spaces and local inter-process 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 paper facilitate several solutions to this problem. For instance, the authentication service of section 2.2 and our boot service could easily be extended to provide certificates for users and users' private keys, respectively, similar to the Certificate Distribution Centers and Login Enrollment Agent Facility of SPX [TA91]. The secure channels provided by our architecture also facilitate simpler password-based user authentication mechanisms.

Acknowledgements

We are grateful to Brad Glade for commenting on an early version of this paper, and especially for suggesting the inclusion of the timestamp in the first message of the protocol of figure 2. We are also thankful to Fred Schneider and Tushar Chandra for providing comments on earlier versions of this paper.

References

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A commmunication sub-system for high availability. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, Digital Equipment Corporation Systems Research Center, February 1989.

[Bir93] K. P. Birman. The process group approach to reliable distributed computing. To appear in *Communications of the ACM*, 1993.

[BJ87a] K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[BJ87b] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BM90] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. *Computer Communication Review*, 20(5):119–132, October 1990.

[BSS91] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[Cri89] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[CZ85] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[DES77] Data encryption standard. National Bureau of Standards, Federal Information Processing Standards Publication 46, Government Printing Office, Washington, D. C., 1977.

[DF92] Y. Desmedt and Y. Frankel. Shared generation of authenticators and signatures. In J. Feigenbaum, editor, *Advances in Cryptology—CRYPTO '91 Proceedings, Lecture Notes in Computer Science 576*, pages 457–469. Springer-Verlag, 1992.

[DH79] W. Diffie and M. E. Hellman. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*, 67(3):397–427, March 1979.

[DS81] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, August 1981.

[GMD91] J. M. Galvin, K. McCloghrie, and J. R. Davin. Secure management of SNMP networks. In I. Krishnan and W. Zimmer, editors, *Integrated Network Management, II*, pages 703–714. Elsevier Science Publishers B.V. (North-Holland), April 1991.

[Gon89] L. Gong. Using one-way functions for authentication. *Computer Communication Review*, 19(5):8–11, October 1989.

[Gon92] L. Gong. A security risk of depending on synchronized clocks. *ACM Operating Systems Review*, 26(1):49–53, January 1992.

[Gon93] L. Gong. Increasing availability and security of an authentication service. To appear in *IEEE Journal on Selected Areas in Communications*, 1993.

[GZ84] R. Gusella and S. Zatti. TEMPO—A network time controller for a distributed Berkeley UNIX system. In *Proceedings of the USENIX Summer Conference*, pages 78–85, June 1984.

[HT88] M. P. Herlihy and J. D. Tygar. How to make replicated data secure. In C. Pomerance, editor, *Advances in Cryptology—CRYPTO '87 Proceedings, Lecture Notes in Computer Science 293*, pages 379–391. Springer-Verlag, 1988.

[KT91] F. M. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, pages 222–230, May 1991.

[LABW92] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[Mar90] K. Marzullo. Tolerating failures of continuous-valued sensors. *ACM Transactions on Computer Systems*, 8(4):284–304, November 1990.

[Mar93] K. Marzullo. Personal communication, February 1993.

[Mil89] D. L. Mills. Network Time Protocol (version 2) specification and implementation. RFC 1119, Network Working Group, September 1989.

[NS78] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[OR87] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM Operating Systems Review*, 21(1):8–11, January 1987.

[RB92] M. K. Reiter and K. P. Birman. How to securely replicate services. Technical Report 92-1287, Department of Computer Science, Cornell University, June 1992.

[RBG92] M. K. Reiter, K. P. Birman, and L. Gong. Integrating security in a group oriented distributed system. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*, pages 18–32, May 1992.

[RD91] R. L. Rivest and S. Dusse. The MD5 message-digest algorithm, July 1991.

[Riv91] R. L. Rivest. The MD4 message digest algorithm. In A. J. Menezes and S. A. Vanstone, editors, *Advances in Cryptology—CRYPTO '90 Proceedings, Lecture Notes in Computer Science 537*, pages 303–311. Springer-Verlag, 1991.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SG92] S. G. Stubblebine and V. D. Gligor. On message integrity in cryptographic protocols. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*, pages 85–104, May 1992.

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

[SS75] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[TA91] J. J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 232–244, May 1991.

[Tsu92] G. Tsudik. Message authentication with one-way hash functions. In *Proceedings of IEEE INFOCOM '92*, pages 2055–2059, May 1992.

[Tuc79] W. Tuchman. Hellman presents no shortcut solutions to the DES. *IEEE Spectrum*, 16(7):40–41, July 1979.

[TY91] J. D. Tygar and B. S. Yee. Strongbox. In J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors, *Camelot and Avalon, A Distributed Transaction Facility*, chapter 24, pages 381–400. Morgan Kaufmann, San Mateo, California, 1991.

[VK83] V. L. Voydock and S. T. Kent. Security mechanisms in high-level network protocols. *ACM Computing Surveys*, 15(2):135–171, June 1983.

[vRBC+92] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable multicast between microkernels. In *Proceedings of the USENIX Microkernels and Other Kernel Architectures Workshop*, April 1992.