Antigone: Implementing Policy in Secure Group Communication*

Patrick McDaniel† and Atul Prakash
Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor
pdmcdanaprakash@eecs.umich.edu

Abstract

Significant strides have been made in achieving strong semantics and security guarantees within group communication and multicast systems. However, the scope of available security policies in these systems is often limited. In contrast, the applications that require the services provided by these systems can differ significantly in their security policy needs. Often application designers have to either make significant compromises in using a given group communication system or build their own customized solutions, an error-prone task. This paper presents Antigone, a framework that provides a suite of mechanisms from which flexible application security policies may be implemented. With Antigone, developers may choose a policy that best addresses their security and performance requirements of an application requiring group communication. We describe the Antigone's mechanisms, consisting of a set of micro-protocols, and show how different security policies can be implemented using those mechanisms. We also present a performance study illustrating the security/performance tradeoffs that can be made using Antigone. Through an example conferencing application, we demonstrate the use of the Antigone applications programming interface and consider the use of policy in several distinct session environments.

1 Introduction

Increases in the power of digital communication has lead to the emergence of useful business, commercial, and personal information management systems. However, the security models under which these applications and services operate are often limited. Many existing systems define a single threat model during development. Thus developers, rather than users and administrators, largely define the security services available to the participants and data. As a result, issues such as performance, availability, and trust are weighed during software design, and not within the context of a specific application session.

The lack of support for user-defined requirements in existing applications highlights the need for greater control over the types and use of software services. Increasingly, software *policies* are used to address this need. Derived from predefined rules and runtime context, a *policy* defines the relevant behaviors and

*This work is supported in part under the National Science Foundation Grant #ATM-9873025 and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

†Patrick McDaniel's research is supported in part by the NASA Graduate Student Researchers Program, Kennedy Space Center, Grant Number 10-52613.

infrastructure requirements of the participants and applications at the time at which software is used. Thus, by using policies, an application may address the changing requirements of each session independently.

In this paper we describe the Antigone system[1], a framework for the flexible definition and implementation of security policies in group communication systems. A central element of the Antigone architecture is a suite of *mechanisms* that provide the basic services needed for secure groups. Policies are implemented through the composition and configuration of these mechanisms. Mechanisms are composed in different ways to address new requirements and environmental constraints. Thus, Antigone does not dictate the available security policies, but provides high-level mechanisms for implementing them. Although the study of policy is applicable to a great many application domains, we focus centrally on issues of security policies in group communication. However, the design and analysis of Antigone may provide useful insights for the architects of policy frameworks in other domains.

Each Antigone mechanism provides some set of facilities needed for secure group communication through the implementation of one or more security *micro-protocols*. Antigone provides micro-protocols implementing a number of facilities useful in secure groups; member authentication, join, session key and membership view distribution, application messaging, process failure, and member leave.

Reliable detection of process failures is a central requirement of several important security policies. Antigone provides an efficient approach to detect process failures in group communication. This approach, called *secure heartbeats*, uses an approach similar to one-time passwords to amortize the cost of keep-alive generation over many messages. Unlike many other approaches to failure detection in groups, no assumptions about the availability of reliable communication or public key certificates are needed. Thus, using this approach, many of the costs typically associated with failure detection in groups may be eliminated.

To support the needs of users in group communication systems, we have developed a taxonomy of security policies. We identify dimensions along which a range of group communication policies can be configured, and describe the often subtle differences between policies found within these dimensions.

Addressing the security requirements of group communication is complicated not only by the lack of globally available security services, but also by the limited availability of efficient group communication services (e.g., multicast). Antigone provides a single abstraction for unreliable group communication that can be configured to use unicasts or multicasts. Using this abstraction, applications can avoid dependence on multicast, but where available, realize its performance advantages.

The majority of secure group communication systems present in the literature are designed to address the requirements of a limited range of users and environments. Conversely, Antigone seeks to address a wide range of environments within a single framework. We do not expressly attempt to discover new solutions for secure groups, but identify ways in which existing approaches may be efficiently integrated and controlled. We state the following as the primary goals of Antigone:

1. *Flexible Security Policy* - An application should be able to specify and implement a wide range of security policies, with appropriate performance tradeoffs.

2. *Flexible Threat Model* - The system should support threat models appropriate for a wide range of applications.

3. *Security Infrastructure Independence* - The framework should not be dependent on the availability of a specific security infrastructure or technology.

4. *Transport Layer Independence* - The solution should not depend on the availability of any single transport mechanism (such as IP Multicast [2]). On the other hand, our solution should be able to take advantage of available multicast services.

5. *Performance* - The performance overheads of implementing security policies should be kept low.

An early version of Antigone has been integrated into the `vic` [3] video-conferencing system. The result of that effort, called the Secure Distributed Virtual Conferencing (SDVC) application, was used to broadcast the September 1998 Internet 2 Member Meeting to several sites across the United States. Using high speed cryptographic algorithms we are able to attain television-like secure video frame rates (30 frames/second) over a LAN. Details of the implementation and our experiences deploying SDVC can be found in [4, 5].

In the next section, we describe related work in secure groups, policy management, and composable architectures and protocols. Section 3 considers a design space of secure group communication policies. Section 4 details the Antigone architecture, operation, and protocols. Section 5 presents results of a performance study investigating the costs associated with several important security policies. Section 6 presents an overview of the Antigone API through an example application. Section 7 considers the factors contributing to the selection of an application policy and illustrates their use within several session environments. Finally, Section 8 presents concluding remarks and directions for future work.

2 Related Work

Often cited as the genesis of current group communication technologies, the ISIS [6] and later HORUS [7] frameworks provide interfaces for the construction of group architectures. Using these frameworks, developers can experiment with a number of protocol features through the composition and configuration of protocol modules. One important feature introduced by the HORUS system was a comprehensive security architecture. A central contribution of this architecture was the identification of a highly fault-tolerant key distribution scheme. Process group semantics are used to facilitate secure communication. A single session key is used throughout each HORUS session. However, vulnerability of the group to attacks from past or future members is limited. Application messages have sender authenticity and may be confidential.

Virtual private networks provide an abstraction in which applications designed for (logically) local network traffic can be executed across physically larger networks. The Enclaves system [8] extends this model to secure group communication. Enclaves secures the group content from previous members by distributing a new group key after any member leaves the group. Also, it is implied that the group key should be changed periodically. Enclaves distributes group membership information but is not dependent on it. Messages have confidentiality and through point-to-point communication, sender authenticity.

The RAMPART system [9] provides secure group communication in the presence of actively malicious processes and Byzantine failures. Protocols in RAMPART rely heavily on distributed consensus algorithms to reach agreement on the course of group action. Secure channels between pairs of members are used to ensure message authenticity. Authenticity guarantees are used to ensure the accuracy of the group views[1] constructed through membership protocols. The security context is not changed through shared session keys, but through the secure distribution of group views. Application messages have sender authenticity and integrity.

An often cited limitation of existing group communication systems is their inherent lack of scalability. In [10], Mittra defines the 1 effects n failure, where a single membership change event can effect the entire group. Mittra's Iolus system addresses this limitation through locally maintained subgroups. Each subgroup maintains its own session key, which is modified after a member leaves. Therefore, the effect of a membership change is localized to the subgroup in which it occurs. Iolus provides confidentiality through encryption under session keys.

Key hierarchies [11, 12] provide an efficient alternative to subgrouping in achieving scalable, secure key distribution. A key hierarchy is singly rooted n-ary tree of cryptographic keys. The interior node keys are

[1] A group *view* is the set of identities associated with members of the group during a period where no changes in membership occur. When the membership changes (a member joins, leaves, fails, or is ejected), a new view is created. This is a similar concept to Birmans's group view [6].

assigned by a session leader, and each leaf node key is a secret key shared shared between the session leader and a single member. Once the group has been established, each member knows all the keys between their leaf node key and the root. As changes in membership occur, rekeying is performed by replacing only those keys known (required) by the leaving (joining) member. Rekeying without membership changes can be achieved by inexpensively replacing the root key. Thus, the total cost of rekeying in key hierarchies scales logarithmically with group size.

The Group Key Management Protocol (GKMP) [13] attempts to minimize the costs associated with session key distribution by loosening the requirement that the group content be protected from past and future members. After being accepted into the group, newly joined members receive a Key Encrypting Key (KEK) under which all future session keys are delivered. A limitation of this approach is that misbehaving members can only be ejected by the establishment of a new group. GKMP reduces the costs of authentication by introducing a peer-to-peer review process in which potential members are authenticated by active members of the group. The joining member's authenticity is asserted by existing members. GKMP provides periodic rekeying. Note that this is a key management protocol, and as such does not mandate how the session key is used.

The Group Secure Association Key Management Protocol (GSAKMP) [14] defines an architecture and protocol used to implement secure multicast groups. Policy is implemented in GSAKMP through the distribution of session specific *policy tokens* distributed to each group member. Similar in spirit to the security associations (SA) of IPSec [15] used for peer communication, the policy token defines the security requirements and access control for a lightweight multicast session. Much of the philosophy on which the policy layer of Antigone is based was developed from ideas present in the GSAKMP specification (see Section4.3).

The IPSec [15] standards enable secure peer communication through the introduction of network layer security mechanisms. The Scalable Multicast Key Distribution (SMKD)[16] standard extends this approach to the multicast environment. Developed for the Core Based multicast routing protocol [17], SMKD uses the router infrastructure to distribute session keys. As the multicast tree is constructed, leaf routers obtain the ability to authenticate and deliver session keys to joining members. Thus, the overheads associated with member authentication and key management can be distributed among the leaf routers. Similar to GKMP, SMKD provides periodic rekeying.

Introduced in [18] by Blaze et. al., trust management provides a unified approach for the specification and evaluation of access control. At the core of any trust management system is a domain independent language used to specify the capabilities, policies, and relationships of participant entities. Applications implementing trust management consult an evaluation algorithm (engine) for access control decisions at runtime. The engine evaluates the access control request using pre-generated specifications and environmental data. Therefore, applications need not evaluate access control decisions directly, but defer analysis to the trust management engine. Through rigorous analysis, the PolicyMaker [18] trust management engine has been proven to be correct. Thus, with respect to access control, any application using PolicyMaker is guaranteed to evaluate each decision correctly. However, enforcement is left to the application. Several other systems (e.g., KeyNote [19] and REFEREE [20]) have extended the trust management architecture to allow easier integration with user applications and a minimal set of enforcement facilities.

The Security Policy System (SPS) [21] is an architecture supporting the flexible definition and distribution of policies used to define IPSec (peer communication) security associations (SA)s. In SPS, policy databases warehouse and distribute specifications to policy clients and servers. Policy servers coordinate (with clients) the interpretation, negotiation, and enforcement of SA policies. The scope of policy in SPS is limited. To simplify, SPS policies state acceptable access control and identify the use of cryptographic message transforms (i.e. access control and message security).

The Secure Multicast Research Group (SMuG) is an IRTF sponsored research body chartered with the task of investigating technologies and frameworks for secure multicast. The SMuG efforts are directed at

three problem areas defining the central tasks required by secure multicast systems. These areas include multicast data security, key management, and policy management. A proposal architecture and detailed description of these efforts can be found in [22]. Findings and solutions of the SMuG research group are intended to fuel future standards.

The *micro-protocol* [23, 24] design methodology is useful when constructing systems with dynamic protocol stacks. Using micro-protocols, a system designer may decompose system facilities into their atomic components. *Composite protocols* are constructed from a collection of the smaller micro-protocols. Hence, differing facilities and guarantees may be provided through composition. In [24], the authors define a suite of micro-protocols used for maintaining group membership views in distributed systems.

3 A Taxonomy of Group Security Policies

Applications, depending on the perceived risks and performance requirements, require different levels of security. In Antigone, requirements for security are met through the specification, distribution, and enforcement of a session specific group policies. In this section, we outline several dimensions along which a group policy may be defined. However, as needed by new applications and environments, additional policy dimensions may be added to Antigone with minimal effect on its design.

We focus this discussion on those dimensions deemed essential for secure multi-party communication. The dimensions we have identified as essential for group applications and environments include: *session rekeying policy*, *data security policy*, *membership awareness policy*, *process failure policy*, and *access control policy*. A session rekeying policy defines a set of events after which the session security context is required to be changed (i.e. group rekey). A data security policy defines the security guarantees applied to application messages. A membership policy dictates the availability and accuracy guarantees of membership information. A process failure policy defines the type of failures detected by the system, and where available, the means of recovery. An access control policy states the rights (and potentially the responsibilities) of the group members. The remainder of this section describes the nature and implications of these policy dimensions.

3.1 Session Rekeying Policy

A popular approach used to implement secure groups among trusted members is the distribution and subsequent maintenance of shared symmetric session keys. An important issue in the use of these keys is determining when a session must be rekeyed, i.e., the old session key is discarded and a new session key is sent to all members. The session rekeying policy states the desired properties of the rekeying process. These properties indicate the lifetime and acceptable exposure of the session keys to past and future members of the group, and represent threats from which the group is required to be resistant. Four important rekeying properties are:

- *session key independence* - This property requires that possession of a session key does not give any meaningful information about past or future session keys.
- *membership forward secrecy* - This property states that a member leaving the group cannot obtain meaningful information about future group communication. This requires that possession of a session key does not give any meaningful information about future session keys (called *perfect forward secrecy*), and that session keys are replaced after each member leaves the group.
- *membership backward secrecy* - This property states that a member joining the group cannot obtain meaningful information about past group communication. This requires that possession of a session

key does not give any meaningful information about past session keys (called *perfect backward secrecy*), and that session keys are replaced after each member joins the group.

- *limited-lifetime* - This property states that a session key has a maximum lifetime (which may be measured in time, bytes transmitted, or some other globally measurable metric). Thus, a session key with a limited-lifetime is required to be discarded (and the session rekeyed) when its lifetime is reached.

Rekeying properties may be combined in different ways to precisely define the desired group security. For example, a group may wish to enforce a policy with both membership forward secrecy and limited-lifetime. Thus, the group would be protected from future members, and each session key would have some maximum lifetime. The combination of these two properties identifies a particular threat model for the group. Thus, as user environments evolve, their associated threat models may be addressed through combinations of rekeying properties.

Session key independence is a prerequisite for both membership forward and backward secrecy. If the process over which a key is derived is not independent, then a member may surreptitiously obtain past and future session keys. Throughout, unless otherwise specified, we will assume rekeying is session key independent.

A close relationship exists between session rekeying and group membership. Applications often need protection from members not in the current *view*. Therefore, as determined by the group threat model, changes in membership may require the session to be rekeyed. If rekeying is not performed after each change in membership, the view does reflect a secure group, but indicates only the set of members that are participating in the session. Past members may retain the session key and continue to receive content. Future members may record and later decode current and past content. In applications that need protection from past or future members, rekeying after membership events is necessary.

We say that a group security policy is *sensitive* to an event if the group changes the security context in response to the observation of the event. Typically, the security context is changed by distributing a new session key (rekeying). A group security policy is often sensitive to *group membership events*. Group membership events include; (1) `JOIN` event, which is triggered when a member is allowed to join the group; (2) `LEAVE` events, which is triggered when a member leaves the group; (3) `PROCESS FAILURE` events, when a member is assumed to have failed in some manner; and (4) `MEMBER EJECT` events, when a previously admitted member is purged from the group according to some group policy. A policy is called time-sensitive if it rekeys after a specified time interval has passed since the last session rekey.

The sensitivity of a policy to group membership events directly defines the group threat model. For example, consider a group model that is sensitive only to `MEMBER EJECT` events. Because the session is always rekeyed after an ejection, no ejected member can access current session keys. Thus, an application is assured that no ejected member will have access to the current session content. However, the session is not protected from members that have left voluntarily, are assumed to have failed, or join in the future. This policy defines the *ejection secrecy* rekeying property.

Sensitivity mechanisms can be used to build a large number of session rekeying policies. In the following text, we define and illustrate four general purpose policies that are representative of the kinds of secure groups found in existing systems.

3.1.1 Time-sensitive Rekeying Policy

Independent of membership events, groups implementing a time-sensitive policy periodically rekey based on a maximum session key lifetime. The group limits the exposure of the session to cryptanalysis by using the key only for a limited period. Groups implementing a time-sensitive rekeying policy measure key lifetimes

by wall-clock time. Other kinds of limited-lifetime rekeying operate essentially in the same way, save the means by which the lifetime is calculated. For example, a system supporting limited-lifetimes based on bytes transmitted would rekey when a threshold of data has been transmitted under a particular session key. The GKMP [13, 25] protocol implements a time-sensitive rekeying policy.

By periodically rekeying, the group may be protected from an adversary who wishes to block the delivery of new session keys. An adversary who blocks rekeying messages may intend for the group to continue to use an old session key. With a time-sensitive rekeying policy, if a new key is not successfully established after the current session key expires, group members can choose to no longer communicate rather than use a session key with an expired lifetime. Independent of other factors, almost all existing systems provide some form of limited-lifetime rekeying.

An example on-line subscription service illustrates the use of time-sensitive rekeying. In this service, paying members are periodically sent a session key that is valid until the next subscription interval. Because knowledge of a session key is predicated only on member subscriptions, there is no need to support sensitivity to membership events. Members may join or leave the group without loss of security; they have the right to all content for which they have paid.

Typically, systems implementing limited-lifetime rekeying (only) use *Key Encrypting Keys* (KEK) [13] to reduce the costs of rekeying. Rekeying via KEKs is not session key independent. Thus, because the KEK provides access to all session keys and content, the group is not protected from past or future members. Note that systems that use KEKs cannot forcibly eject members without additional infrastructure.

Another promising approach is to use KEKs only where no relevant membership events have occurred since the last rekey. In this way, a group is able to achieve the performance of KEKs when no loss of security results, and the strength afforded of other rekeying approaches elsewhere. Key hierarchies [11, 12] use a similar approach to reduce the costs associated with group rekeying.

3.1.2 Leave-Sensitive Rekeying Policy

Groups implementing a leave-sensitive policy rekey after `LEAVE`, `PROCESS FAILURE`, and `MEMBER EJECT` events. The threat model implied by leave-sensitive groups states that any member who has left the group will not have access to current or future content. For example, a business conferencing system that supports negotiations between a company's representatives and a supplier may benefit from leave-sensitive rekeying. Once the supplier leaves, a leave-sensitive rekey policy would prevent subsequent discussions from being available to the supplier, even if the supplier is able to intercept all the messages. Leave-sensitive policies achieve membership forward secrecy. The Iolus [10] implements a form of leave-sensitive rekeying.

A consideration of groups implementing leave-sensitive policies is the requirement for liveness. Unless each group member periodically asserts their presence in the group, process failures cannot be detected. In large or highly dynamic groups, the cost of these assertions may be prohibitive.

3.1.3 Join-sensitive Rekeying Policy

Groups implementing a join-sensitive policy rekey only after `JOIN` events. The threat model implied by join-sensitive groups states that any member joining the group should be unable to access past content. Join-sensitive policies achieve membership backward secrecy.

In large or highly dynamic groups, the cost of rekeying after each join can be prohibitive. For example, in network broadcasts the number of receivers is often large, and little control over member arrival and departure can be asserted. However, the threat models associated with join sensitivity are not commonly found in applications such as broadcasting. Several techniques may be used to mitigate the costs of implementing join sensitivity (e.g., batched joins, minimum session key lifetimes). In practice, a join-sensitive rekeying

policy is likely to be used in conjunction with a time-sensitive or leave-sensitive policy to limit the duration over which past members can access current session content.

3.1.4 Membership-sensitive Rekeying Policy

Groups implementing a membership-sensitive policy rekey after every membership event. The threat model implied by membership-sensitive groups states that joining members will not have access to past content, and that past members will not have access to current or future content. Thus, this policy is the combination of leave-sensitive and join-sensitive rekeying. Membership-sensitive policies achieve both membership backward and forward secrecy.

Because each membership event triggers rekeying, the group view defines exactly those members who have access to current content. Membership-sensitive policies are often among the most expensive to implement. Thus, these policies are typically avoided unless strictly needed by an application.

Applications with comprehensive security requirements often need membership sensitivity. For example, in reliable group communication systems, ensuring the security of message delivery (e.g., atomicity, reliability) requires tight control over the group. The RAMPART [9] system provides a type of membership-sensitive service.

3.1.5 Other Rekeying Policies

It often desirable for other (application level) events to influence rekeying. For example, in a business conferencing application, a policy may state that rekeying occur only when a member with the role `Supplier` leaves. In providing policies that integrate application semantic with rekeying, the group can achieve exactly the desired behavior at a minimal cost.

It may be important for the group to be more sensitive at certain times, but less at others. Similarly, groups may wish sensitivity to be a function of group size or resource availability. In this way, a group can adapt to the capabilities of the available infrastructure.

The number of factors that can contribute to rekeying policies is unbounded. As new application or environmental constraints emerge, new rekeying policies can be defined. Thus, as is available in Antigone, it is advantageous to allow developer defined events to affect when and how the group is rekeyed.

3.2 Data Security Policy

The canonical policy, a data security policy states the security guarantees applied to application messages. The most common types of data security are: *integrity*, *confidentiality*, *group authenticity*, and *sender authenticity*. These policies are essential for protecting the application content, and define in large part the quality of security afforded by the group.

Confidentiality guarantees that no member outside the group may gain access to session content. Although typically implemented through encryption under the session key, other techniques may be used to limit content exposure. For example, confidentiality may be achieved though the use steganography, or through encryption of only critical portions of messages.

Integrity guarantees that any modification of a message is detectable by receivers. As they are fundamentally insecure, one cannot trust underlying reliable communication (point to point TCP [26], reliable group communication [27]) to guarantee integrity. Sequence numbers, checksums, and other components of these protocols can be trivially altered by adversaries to manipulate message content. The use of session keyed message authentication codes (MAC) [28] is an inexpensive way to achieve message integrity.

Group authenticity guarantees that a received message was transmitted by some member of the group, and is typically a byproduct of other data security policies. In many cases, proof of the knowledge of

the session key (as achieved through most confidentiality and integrity guarantees) is sufficient to establish group authenticity.

Sender authenticity (also known as source authentication) guarantees the sender of a message can be uniquely identified. Achieving sender authenticity is expensive using known techniques (e.g., off-line signatures [29], stream signatures [30]). Thus, for high throughput groups, sender authenticity is often avoided.

One may consider a number of other useful data security policies. For example, some systems may require non-repudability. A non-repudability policy guarantees that the sender of a message cannot later deny transmission. Another potential policy is anonymity, in which the sender of a message specifically cannot be identified. There are several ongoing works investigating these and other policies.

Closely related to data security policies, a *cipher-suite* policy is one or more cryptographic algorithms used to enforce specified policies. As encryption algorithms have varying availability and characteristics, a cipher suite policy should support the specification of acceptable algorithms, parameters, and modes. A cipher suite policy is relevant not only to data security policies, but to any policy using cryptographic techniques. Thus, it is important to understand the strengths and weaknesses of each algorithm with respect to the policy for which it is used.

Note that a single policy need not apply to every message. In many applications, individual messages have unique data security requirements, depending on the nature of the message and the assumed threat model. Thus, is is useful to provide facilities for the per-message specification of data security policies.

3.3 Membership Policy

Identification of the group membership is an important requirement for a large class of applications. For example, many reliable group communication systems need accurate membership information for correct operation. Conversely, as seen in typical multicast applications, members of other systems need not be aware of group membership at all. In this second environment, providing other services (such as reliability and fault-tolerance) is commonly left to the application. Because each relevant change in membership requires the distribution of new group views, guaranteeing the correctness and availability of membership views can be costly.

A membership policy states the availability and accuracy requirements of view distribution. Views need only be as accurate as required by an application. Thus, it is useful to provide a range of membership guarantees with associated costs. Several useful membership policies include:

- *best-effort membership* - In this policy, membership data will be delivered as available and convenient. No guarantees about the accuracy or timeliness of this information are provided. However, it is expected that due-diligence is expended in providing accurate views.

- *positive membership* - This policy guarantees that all members in the view are participating in the group. Thus, within some known time bounds, each member in the view is guaranteed to have not exited the group, failed, or been ejected. This policy is useful in applications that need to determine exactly who is participating in a session.

- *negative membership* - This policy guarantees that every member who has access to the session key is listed in the view. This policy is useful in applications that need to ensure that particular entities are not present.

- *perfect membership* - This policy guarantees that all members in the view are participating in the group, and that every member who has access to the session key is listed in the group view. That is, both positive and negative membership is provided.

Confidentiality of group membership is a requirement of some applications. However, concealing membership from members and non-members is difficult in current networks. This is primarily due to ability of adversaries to monitor messages on the network. These messages expose the source and destination of packets (in the case of unicasts) and at the multicast tree (in the case of IP multicasts). In mounting this *traffic analysis attack*, an adversary may deduce a close approximation of group membership.

3.4 Process Failure Policy

A process failure policy states the set of failures to be detected, the security required by the failure detection process, and the means and security of recovery. The defining characteristic of a failure detection mechanism is its fault model. The fault model defines the types of behavior exhibited by a faulty process that the mechanism will detect. Typical crash models include fail-stop, message omissions, or timing errors [31]. In the strongest (Byzantine failure) model, a faulty process may exhibit any behavior whatsoever.

Often, the failure detection process itself is required to be secure. In securing the failure detection, the group is protected from the masking of process failures by adversaries. However, protecting the group from an adversary who attempts to generate false failures may be more difficult. Failures may be forced by blocking all communication between the group participants. This *denial of service attack* is difficult to address solely in software.

3.5 Access Control Policy

An access control policy states which participants may perform actions or access information. These individual capabilities, called *rights*, provide a road-map for the group operation. The definition of participant identities, the enumeration of rights, and the mapping of identities to rights are the core components of an access control policy.

A key component of access control is the means by which group members are authenticated. Members are often authenticated at or before joining a group using public key certificates (e.g., PGP [32]), or through the use of centralized authentication servers (e.g., Kerberos [33]). In other applications, such as pay-per-view broadcasts, group members can establish rights through credentials obtained from application specific subscriptions [34]. In many cases, the true identity of the member need not be known (e.g., anonymous groups).

Access control models typically are gleaned from the organization of the participants in the target application domains or underlying communication infrastructure. In [14], rights are assigned to each member as needed to carry out the duties of a particular *role*. For example, a member who is assumes the `Member` role is assigned the right to access the session key.

4 Antigone Architecture

In this section, we present the design of the Antigone framework and they ways in it is used to implement secure groups. Described in Fig. 1, the Antigone architecture consists of three software layers; the broadcast transport layer, the mechanism layer, and the policy layer.

The broadcast transport layer defines a single abstraction for unreliable group communication. Due to a number of economic and technological issues, multicast is not yet globally available. Where needed, Antigone emulates multicast groups using the available network resources. The broadcast transport layer is described in detail in Section 4.4.

The mechanism layer provides a set of mechanisms used to implement security policies. Each mechanism represents a set of basic features required for secure groups. Policies are flexibly defined and imple-

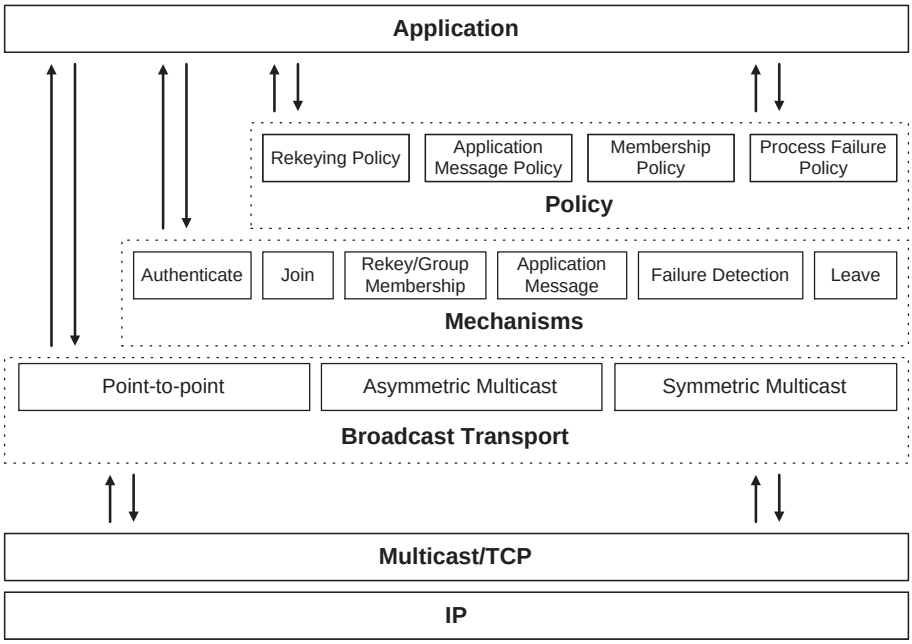

Figure 1: Antigone consists of three middleware layers; the broadcast transport layer, the mechanism layer, and the policy layer. The broadcast transport layer provides a single group communication abstraction supporting varying network environments. The mechanism layer provides a set of micro-protocols and software services used to implement secure groups. The policy layer implements a suite of general purpose policies.

mented through the composition and configuration of mechanisms. We describe the design and operation of the Antigone mechanisms and associated micro-protocols in Section 4.2.

The policy layer provides a suite of general purpose policies. These available policies represent those commonly needed in secure group communication systems. Clearly, there are some policies beyond what is available in this layer. Where required, an application may implement policies through direct integration with the broadcast transport and mechanisms layers. An overview of the policies supported by Antigone is presented in Section 4.1, and details of the operation of policy layer is described in Section 4.3.

4.1 Supported Policies

In this section we briefly describe the policies supported by the Antigone policy layer. We detail the way these policies are represented, distributed, and supported in section 4.3.

Antigone assumes that all group members are trusted, i.e., members do not attempt to circumvent the security of the system. We note that in some environments, this assumption may not be warranted. For example, the RAMPART [9] system assumes that a subset of the group membership may be malicious. There are known techniques addressing these scenarios (e.g., secure majority voting). As protection of the group from malicious members requires solutions with significant design complexity and performance costs, we chose not to address these threats in the initial version of Antigone. However, as described in Section 4, additional infrastructure addressing these threats may be easily integrated into Antigone. We assume that in mounting an attack, non-members (*adversaries*) may attempt to intercept messages, modify messages, or prevent messages from being delivered.

All rekeying in Antigone is session key independent. Thus, we provide a stronger guarantee than time-

sensitive groups that use KEKs; a member who has left the group may continue to access the group content only until the next rekey. A past member cannot access current or future group content without again joining as a member. However, to take advantage of its performance, we may choose to implement KEKs into future versions of Antigone. Antigone supports the rekeying policies defined by their sensitivity to membership events and key lifetimes described in Section 3.1.

The data security policies supported by Antigone include confidentiality, integrity, and sender authenticity. Specified in the group policy, one or more of these properties are guaranteed for application level messages.

We note that membership policies are strongly tied to the rekeying process. Based on membership and rekeying policies, the same events that trigger rekeying can also require distribution of a new group views. Antigone currently provides mechanisms to distribute keys with and without membership information. Thus, when membership information is distributed, the accuracy and timeliness of group views is strictly determined by the rekeying policy. For example, limited-lifetime rekeying provides best-effort membership, and membership-sensitive rekeying provides perfect membership.

Antigone supports detection of fail-stop faults of group members. To prevent problems due to timing errors, synchronized clocks and timestamps are not used in the Antigone protocols. However, some mechanisms for failure detection and time-sensitive rekeying rely on timeouts at individual processes. A process whose clock progresses at an incorrect rate may take longer to detect failures (if its clock progresses too slow) or may mistakenly assume that there is a failure (if its clock progresses too fast and thus times out).

Antigone implements a simple access control infrastructure. We define one access control right; the ability to gain access to the group. Group access is controlled through an Access Control List (ACL) stored at the group leader (see Section 4.2). Once a member has gained access to the session key, they may freely transmit and receive messages from the group.

4.2 Security Mechanisms

Policy in Antigone is implemented through the composition and configuration of software modules called *mechanisms*. Each Antigone mechanism consists of a set of behaviors and associated micro-protocols designed to perform some service required by secure groups. The mechanisms layer defines six mechanisms; authentication, member join, session key and group membership distribution, application messaging, failure detection, and member leave. The micro-protocols associated with these mechanisms are presented in Fig. 2.

All communication between policy-implementing software and mechanisms is through Antigone-specific events. Possibly in response to the observation of an external event (e.g., message arrival, member join), the mechanisms layer generates and delivers mechanism events to the policy layer. Where required, the policy layer directs mechanism operation through the generation and delivery of policy events.

Policy events are also used to direct mechanism composition. Composition and configuration events are generated by the policy layer in response to the arrival or creation of a group policy. Mechanisms receiving these events initialize the appropriate services and internal state variables. A map of subsequent mechanism behavior is derived from data included in the event. For example, the `ANTI_SET_POLICY`[2] event is triggered when the group policy has been established at a group member. The current failure detection mechanism creates a number of detection timers upon reception of this event. Data included with the event indicate the parameters with which the mechanism is to operate (e.g., timer values).

Before describing the operation of each mechanism, we define the Antigone group model, notation, and use of cryptography. Fig. 3 shows the principals in an Antigone logical group. A distinct member of the

[2]Named events containing the prefix `ANTI` indicate that they are specific to Antigone. The use of the prefix anti- in this context does not imply negation.

Authenticate

1. $A \rightarrow SL : A, G, I_0$ (*authentication request*)
2. $SL \rightarrow TTP : SL, A, I_1$ (*pair key request*)
3. $TTP \rightarrow SL : \{[\pi_{SL,A} = \{A\}_{K_{SL}} \oplus \{SL\}_{K_A}], I_1\}_{K_{SL}}$ (*pair key response*)

4a. $SL \rightarrow A : SL, A, \{g, A, I_0, I_2, [policy\ block], Pu_G\}_{\sigma_{SL,A}}$ (*authentication response*)

4b. $SL \rightarrow A : SL, A, G, I_0, H(SL, A, G, I_0)\}_{\sigma_{SL,A}}$ (*authentication reject*)

Join

5. $A \rightarrow SL : A, \{A, I_2\}_{\sigma_{SL,A}}$ (*join request*)

Rekey/Group Membership

6a. $SL \rightarrow A : g, S_{SL}, (A, \{g, SK_g\}_{\sigma_{SL,A}}), \{H(g, S_{SL}, (A, \{g, SK_g\}_{\sigma_{SL,A}}))\}_{SK_g}$ *(key distribution)*

6b. $SL \rightarrow A : g, S_{SL}, (A, \{g, SK_g\}_{\sigma_{SL,A}}), B, C, D..., \{H(g, S_{SL}, (A, \{g, SK_g\}_{\sigma_{SL,A}}), B, C, D, ...)\}_{SK_g}$ (*key/group membership distribution*)

6c. $SL \rightarrow group : g+1, S_{SL}, (A, \{g+1, SK_{g+1}\}_{\sigma_{SL,A}}), (B, \{g+1, SK_{g+1}\}_{\sigma_{SL,B}}), \ldots, \{H(g, S_{SL}, (A, \{g+1, SK_{g+1}\}_{\sigma_{SL,A}}), \ldots)\}_{SK_{g+1}}$ *(session rekey)*

Application Messaging

7a. $A \rightarrow group : g, A, [msg], \{H(g, A, msg)\}_{SK_g}$ (*with integrity*)

7b. $A \rightarrow group : g, \{A, [msg]\}_{SK_g}$ (*with confidentiality*)

7c. $A \rightarrow group : g, \{A, [msg]\}_{SK_g}, \{H(g, \{A, [msg]\}_{SK_g})\}_{SK_g}$ (*with integrity and confidentiality*)

7d. $A \rightarrow group : g, A, [msg], \{H(g, A, msg)\}_{C_A}$ (*with sender authenticity*)

Failure Detection

8. $A \rightarrow SL : S_A^i, \delta^{k-i}, g, A, S_A^0, \delta^k, \{H(g, A, S_A^0, \delta^k)\}_{\sigma_{SL,A}}$ (*member heartbeat*)
9. $SL \rightarrow group : S_{SL}^i, \delta^{k-i}, g, SL, S_{SL}^0, \delta^k, \{H(g, SL, S_{SL}^0, \delta^k)\}_{Pr_G}$ (*session leader heartbeat*)
10. $A \rightarrow SL : g, A$ (*key retransmit request*)

Leave

11. $A \rightarrow SL : A, \{g, A, S_A, \{g, B\}_{SK_g}\}_{\sigma_{SL,A}}$ (*leave request*)

Figure 2: Antigone Micro-Protocol Description - micro-protocols for the various operating modes. A *composite protocol* is constructed from the selection of a subset of these modes. In implementing some group policies, a subset of these micro-protocols are omitted entirely.

group, called the *session leader* (SL), is the arbiter of group operations such as group joins, leaves, etc. We chose an arbitrated group because of its low cost and its appropriateness for existing multicast applications. For example, in a secure pay-per-view video application, a broadcaster would provide a session leader that enforces the desired access control and key distribution policies. As needed, additional mechanisms implementing peer groups[3] can be introduced.

External to the group, the *trusted third party* (TTP) is a service used by the session leader to authenticate potential group members. Each potential member *A* of a group (including the session leader) has a shared

[3]A *peer group* is a group in which there is no unique session leader. In these groups, a subset of the membership *contribute* to the process in which consensus of shared group state (e.g., session keys, membership) is achieved. Typically, the costs associated with algorithms achieving consensus in peer groups are significantly higher than those in arbitrated groups. The RAMPART system [9] implements a highly secure peer group.

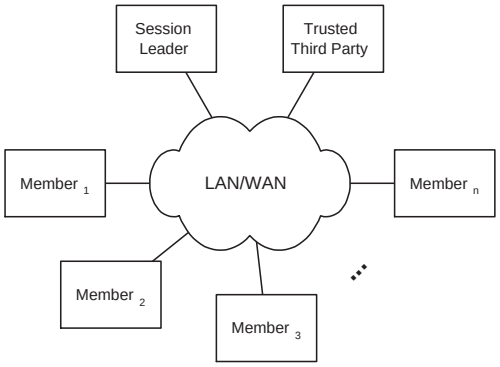

Figure 3: An Antigone group consists of an arbiter called the *session leader* and a set of group members. External to the group, the *trusted third party* is used to authenticate potential group members. No assumptions are made about the network topology or connectivity.

secret K_A registered with the TTP. This secret key is generated and registered with the *TTP* before the party attempts to join any session. We assume an out of band method for registering these keys.

It is assumed that all potential group members know the identity of the session leader and TTP prior to joining a session. The process whereby the existence, identities, and initial parameters of each session are discovered is outside the scope of Antigone. However, there are a number or known approaches and software for the session advertisement (e.g., SDR [35, 36]).

In our protocol descriptions, we use the term *SL* to refer to the identity of the session leader, A to refer to a current or potential member of the session, and *TTP* to refer to the trusted third party. $\{X\}_k$ denotes a message *X* encrypted under the key *k*. The view identifier, *g*, is used to uniquely tag the changing views of group membership. The term SK_g refers to a session key in view g, and SK_{g+1} for the (next) view $g+1$. The term *I*, possibly with a subscript, denotes a nonce value. Key distribution protocols based on Leighton-Micali key distribution [37] define a term $\pi_{A,B}$, called a *pair key*, used to support secure communication between collaborating members *A* and *B*. Derived from the pair key, the session leader and a potential member *A* maintain a shared secret key $\sigma_{SL,A}$. A cryptographic hash for the text x is denoted $H(x)$. The MD5 hash algorithm [38] is used in the current implementation. However, MD5 can easily be replaced with any suitably strong hash function.

The format of the identity, nonce, key, and view identifier values used in our current protocol implementation are as follows. Each identity is a unique 16 byte null terminated ASCII string of alphanumeric characters. A potential member is assigned this value when registering a long term key with the *TTP*. Nonce values are unique 64 bit values. To ensure that nonces are not reused, some source of monotonic values, such as the system clock, may be used. Key format is algorithm dependent. The DES standard used for symmetric encryption throughout is defined for an eight byte key (including eight parity bits). A view identifier g is the concatenation of a group identifier and nonce value. The group identifier G is an eight byte, null terminated name string that uniquely identifies the session. The nonce is an eight byte nonce value. A new view identifier ($g+1$) is created through the concatenation of the group identifier with a freshly generated nonce.

A $policy\ block$ is distributed by the session leader to each group member during the authentication process (message $4a$ in Fig. 2). Defined by policy-implementing software, the policy block is an arbitrary byte string stating the group policy. We describe the definition and use of this data by the policy layer in Section 4.3.

Associated with each member A is a sequence number, S_A (S_{SL} for the session leader). Sequence numbers are initially set to 0, and reset following each session rekey. Heartbeat messages are used by Antigone

to detect failed links and processes. The heartbeat messages contain the member's sequence number and a one-time password (δ). The sequence number is incremented by 1 for each subsequent heartbeat message. We describe the use of heartbeats by the failure detection mechanism in Section 4.2.5.

A session leader creates an *asymmetric key pair* (Pu_G, Pr_G) for each session during initialization. The public key exponent (Pu_G) is given to potential members during the authentication process, and is later used to verify the authenticity of session leader heartbeats. Where sender authenticity is required, we assume the existence of an authenticated certificate distribution service [32, 39] that provides access to the public key certificates of each group member (C_A). Note that certificate distribution services are not required for the generation or distribution of the session asymmetric key pair.

We use DES [40] for all encryption in the current implementation. Its inherent strength is evident from its 20-year history, yet its 56-bit key length has long been the subject of debate. Related algorithms such as triple-DES [41] or DESX [42] offer the strength of DES with considerably longer keys. Our protocols are not tied to any specific property of DES, and may be replaced with other cryptographic algorithms as necessary.

We assume that all processes that have achieved membership, and thus have been authenticated, adhere to the system specification. We assume that no member willingfully discloses its long term or session keys. All members trust the *TTP* not to disclose their long term key, and to generate pair keys according to the specification. The following text describes each of the Antigone mechanisms and their associated micro-protocols. All cited message numbers refer to Fig. 2.

4.2.1 Authentication Mechanism

The authentication mechanism provides facilities for a potential group member to initiate communication with the session leader. Using this mechanism, the session leader authenticates the potential group member and negotiates a *shared secret key*. The shared secret key is later used to implement a secure channel between the two parties.

We use the provably secure Leighton-Micali key distribution algorithm [37] to authenticate the joining process and negotiate the shared secret. The central advantage of the Leighton-Micali algorithm is its low cost; it uses symmetric key encryption throughout, with none of the modular exponentiation operations associated with public key cryptosystems. Traditional public key cryptography requires significantly more computation than symmetric algorithms. The de-facto standard for public-key cryptography, RSA [43], can be up to 100 times slower in software and 1000 times slower in hardware than DES, the predominant symmetric algorithm [28]. Additionally, the lack of universally available certificate services reinforced our decision to use symmetric cryptography.

A prospective member initiates the authentication process by sending message 1 to the session leader containing her identity (A), the session identifier (G), and a nonce value (I_0). Upon reception of this message, the session leader asks the policy-implementing software (policy layer) if A should be allowed access to the session G (see Section 4.3). If access is denied, the requested is rejected via message 4b. A notifies the application of the rejection upon reception of message 4b.

If A is to be allowed access to G, the session leader obtains a member specific pair key $\pi_{SL,A}$ from the *TTP* (messages 2 and 3). Derived from two identities and their associated long term keys, the pair key is used to establish an ephemeral secure channel between the processes. To prevent replay attacks, the session leader verifies the encrypted nonce value I_1 included in the *TTP*'s response. The session leader computes the shared secret key as follows. The session leader generates the value $\{A\}_{K_{SL}}$. This value is XOR-ed with the pair key $\pi_{SL,A}$ received from the *TTP*. The resulting value is the shared secret key ($\{SL\}_{K_A} = \sigma_{SL,A}$) is known only by session leader and the prospective member *A*. *A* need not communicate with the *TTP* to obtain the shared secret key $\{SL\}_{K_A} = \sigma_{SL,A}$; she can compute it directly.

After obtaining the shared secret key, the session leader responds with an authentication response message (4a). The response contains the identities of the session leader and the potential group member, and a block encrypted with the shared secret key $\sigma_{SL,A}$. The encrypted block contains the view (g) and group member (A) identifiers, the group member nonce (I_0), a session leader nonce (I_2), the policy block, and the group public key (Pu_G). Upon receiving this message, A decrypts the contents using $\sigma_{SL,A}$ and verifies the nonce I_0. If the nonce is correct, she knows the response is both fresh (through validation of the nonce) and authentic (from the use of shared secret key).

4.2.2 Join Mechanism

The join mechanism provides facilities for a previously authenticated process to gain access to the group. A potential member initiates the join process by transmitting message 5 containing her identity (A) and the session leader nonce (I_2) encrypted under the secret key shared by SL and A. A uses knowledge of I_2 to prove the completion of the authentication process.

Upon reception of message 5, the session leader validates the nonce value (I_2). If the nonce is not valid, the join request is ignored and the group continues. If the nonce is valid, the new member is accepted into the group. The means by which the group member is accepted into the group (i.e. the session key and group view are updated and potentially distributed) is determined by the configuration of the Rekey/Group Membership mechanism (*see below*).

Mutual authentication is achieved through the verification of the secrets *A* and *SL* share with the *TTP*. The potential member must be in possession of the secret shared with the *TTP* to obtain the session leader nonce I_2. *A* is convinced that message 4a is fresh and authentic by validating the encrypted nonce value I_0 sent in the original authentication request (message 1).

4.2.3 Rekey/Group Membership Mechanism

The Rekey/Group Membership mechanism provides facilities for the distribution of group membership *views* and session keys. We note the distinction between *session rekeying* and *session key distribution*. In session rekeying, a new session key is created and distributed to all group members. In session key distribution, the current session key is distributed to newly joined or current group members. The decision to rekey or distribute an existing session key is dependent on the group rekeying policy. As such, the policy layer directs all keying and group membership operations (see Section 4.3).

Each key distribution message (6a, 6b, and 6c) contains a group identifier (g), the latest session leader sequence number (S_{SL}), and a *Message Authentication Code* (MAC) calculated over the entire message ($H(\ldots)_{SK_g}$ or $H(\ldots)_{SK_{g+1}}$). The group identifier and sequence number identify the current group context. The MAC ensures integrity of the message.

Session keys are distributed via *session key blocks* ($A, \{g, SK_g\}_{\sigma_{SL,A}}$). The intended member of each block is identified by the group member identifier (A), and the remainder of the block is encrypted using the secret key shared with A, $\sigma_{SL,A}$. If, after decryption, the group identifier matches the identifier in the message header, the member is assured that the block was created by the session leader. If the MAC is valid, the member is assured that the message was not modified in transit. Thus, if both are valid, the member can accept the new session key, group identifier, and view as correct and authentic.

Messages 6a and 6b are used for the distribution of current session keys. Message 6a contains a session key block for one member. Message 6b contains a session key block for one member and enumerates the current group view ($B, C, D, \ldots$). Message 6c is used to rekey the session and contains a session key block for each member. The group view is extracted from 6c from the unencrypted portions of the individual session key blocks.

Rekeying in Antigone is similar to key distribution after a LEAVE operation in the Iolus system [10]. The session leader caches the shared secret keys, so creating this message is fast: encryption of 24 bytes (8 bytes of session key plus 16 bytes of group identifier) per member. Members receiving a 6a, 6b, or 6c extract the session key using the shared secret key and begin using it immediately. Note that the size of this message grows linearly with group size ($O(n)$), where n is the number of members in the group).

We note that key hierarchies (see Section 2) can significantly reduce rekeying costs ($O(\log(n))$). However, protocols for these algorithms are considerably more complex than that defined for the current Antigone rekeying mechanism. In the interest of reducing initial development effort, we deferred the investigation of key hierarchies to future work. We are in the initial stages of implementing a key hierarchy based rekeying mechanism, and plan to introduce it in the next version of Antigone.

4.2.4 Application Messaging Mechanism

The application messaging mechanism provides facilities for the secure transmission of application level messages. The available security guarantees include; *confidentiality*, *integrity*, *group authenticity*, and *sender authenticity*.

Message integrity is achieved through *Message Authentication Codes* (MAC). A MAC is generated by encrypting a hash of the message under the session key. A receiver determines the validity of a MAC by decrypting and verifying the hash value. If the hash is correct, the receiver is assured that message has not been modified in transit. Confidentiality is achieved through the encryption of the message under the session key. Group authenticity is a byproduct of either confidentiality or integrity.

Sender authenticity is achieved by digital signature [44]. The signature is generated using the private key exponent associated with the sender's certificate. Receivers obtain the sender's certificate and verify the signature using the associated public key. Note that a byproduct of the use of digital signatures is message integrity.

Due to the computational costs of public key cryptography, the use of per-message digital signatures to achieve sender authenticity is infeasible in high throughput groups. Several efforts have identified ways in which these costs may be mitigated [29, 30, 45]. While the speed of these algorithms is often superior to strictly on-line signature solutions, their bandwidth costs make them infeasible in high-throughput groups. Achieving efficient sender authenticity in high throughput unreliable communication is an outstanding research issue.

Message 7a shows the format of a message with integrity only, message 7b shows confidentiality only, and 7c shows a message with both integrity and confidentiality. Group authenticity can be achieved through messages 7a, 7b, or 7c. Message 7d shows the format of a message with sender authenticity.

4.2.5 Failure Detection Mechanism

The failure detection mechanism provides facilities for the detection and recovery of failed processes. An application's threat model may require the system to tolerate attacks in which an adversary prevents delivery of rekeying material. Thus, without proper failure detection, members who do not receive the most recent session information will continue to transmit under a defunct session key. Additionally, the accuracy of membership information is in part determined by the ability of the session leader to detect failed processes. Thus, the goal of the failure detection mechanism is to determine a) which members are operating, and b) that each process has the most recent group state (session keys and group view).

Defined in messages 8 and 9, Antigone uses *secure heartbeats* to detect failed or disconnected processes. The presence of the sequence number in the heartbeat ensures that it is fresh. The heartbeat group identifier is used to verify that the sending process has the most recent group state.

As directed by the policy layer, each process (including the session leader) periodically transmits heartbeat messages. The session leader detects incorrect or failed processes through the absence of correct member heartbeats (message 8). If a threshold of contiguous member heartbeats is not received, the member is assumed failed and expelled from the group. Similarly, group members confirm the session leader state and connectivity through session leader heartbeat messages (message 9). If a number of contiguous session leader heartbeat messages is not received, the member assumes that the session leader has failed and exits the group.

A member detecting a lost rekey, key distribution, or heartbeat message can initiate recovery by sending a key retransmit message (message 10). The key retransmit message indicates to the session leader that the member requires the most recent group state. The session leader sends a recent key/group membership distribution message (6a, 6b, or 6c) in response to the key retransmit message.

Antigone uses hash chains [46] to amortize the cost of heartbeat generation over many messages[4]. A hash chain is the sequence of values resulting from the repeated application of a secure hash function (f) on some initial value. For example, given an initial value x and chain of length $k+1$, the hash chain is: $\{f^0(x) = x, f^1(x), f^2(x), \ldots, f^k(x)\}$. Because, by definition, (even partial) inversion of f is not feasible, knowledge of $f^i(x)$ gives no meaningful information to derive $f^{i-1}(x)$, for some $i, 0 < i < k$. By revealing $f^k(x)$ securely, the remaining values can be used in reverse order as proof of the knowledge of x. This is useful as in authentication schemes (one-time passwords) because only a person who has knowledge of x can generate the intermediate values.

The secure heartbeats (messages 8 and 9) are generated as follows. Initially, the sending process A generates a random value x of length equal to the output of the hash function (e.g., MD5 has a 128 bit output). Then, A applies hash function a member-determined number of times (k) to generate the following hash chain:

$$\delta^0 = x, \delta^1 = f(x), \delta^2 = f^2(x), \ldots, \delta^k = f^k(x)$$

A generates a *heartbeat validation block* containing the group identifier g, her identity A, the first heartbeat sequence number for which this hash is to be used S_A^0, the last value in the hash chain $f^k(x) = \delta^k$, and a MAC covering these fields:

$$g, A, S_A^0, \delta^k, \{H(g, A, S_A^0, \delta^k)\}_{\sigma_{SL,A}}$$

A heartbeat message is generated by concatenating the current sequence number ($S_A^i = S_A^0 + i$) and the next value in the hash chain (in reverse order, δ^{k-i}) with the validation block. Because encryption is only required when creating the validation block and the hash chain itself is cached, heartbeat generation is fast. When the values of a chain are exhausted ($i > k$) or the session is rekeyed, the member generates a new hash chain and the associated validation block.

Messages 8 or 9 can be validated by checking the MAC and calculating:

$$f^{S_A^i - S_A^0}(\delta^{k-i}) = \delta^k.$$

If the relation holds, then the heartbeat is valid. The heartbeat is authentic because of the use of the shared secret key (or group private key) in the validation block. The heartbeat is fresh because of the presence of the next value in the hash chain. After receiving and validating the initial heartbeat for a hash chain, subsequent MAC validation can be achieved by byte comparison of a validation block of a previously validated heartbeat. Thus, heartbeat validation is fast.

[4]The use of hash chains in Antigone is similar to those found in one-time password authentication systems [47, 48].

Policy	JOIN	LEAVE	FAILURE	EJECT
Time-sensitive	N	N	N	N
Leave-sensitive	N	Y	Y	Y
Join-sensitive	Y	N	N	Y
Membership-sensitive	Y	Y	Y	Y

Table 1: Rekeying policies are defined by their sensitivity to membership events. Join-sensitive groups are MEMBER_EJECT sensitive to allow for member ejection.

It is worth noting that network congestion that causes message loss may be exacerbated by retransmit requests. This problem, known as *sender implosion*, is likely to limit the efficiency of Antigone in large groups or on lossy networks. A retransmit mechanism similar to SRM [27] addressing this limitation is planned for the next version of Antigone. This mechanism will distribute retransmission costs by allowing any member to respond to a request. Such requests are made a random time after loss detection (the delay is computed as a function of the measured distance from the session leader). Members observing a retransmission request suppress local requests, and wait until the desired message is received or a timeout occurs. Members receiving retransmission requests for data they have received delay the response randomly before retransmitting (the length of the delay is a function of the distance from the requester). If it is noted that some other member has performed the retransmission, the request is ignored. Note that this approach in no way affects the security of Antigone, but only serves to reduce the cost of retransmission request processing.

With respect to group members, the goal of the current failure protection mechanism is the reliable detection of a session leader's failure, not its recovery. As needed, additional mechanisms can be introduced in the future that implement recovery algorithms using primary backup, replication, or voting protocols to establish a new session leader.

4.2.6 Leave Mechanism

The leave mechanism provides facilities for members to gracefully exit the group. A member sends message 11 to indicate that it is exiting the group. Because knowledge of the shared secret key is required, message 11 is unforgeable. Because the sequence number and group identifier are present and encrypted, it cannot be replayed. Upon reception of message 11, the session leader will remove the member from the current view and rekey the session as directed by the policy layer.

A member may also use message 11 to request the ejection of another member from the group. The identity of the member to be ejected is placed in the $\{g, B\}_{SK_g}$ block (as B). The session leader receiving a message with this format will eject the member and rekey the session as directed by the policy layer.

4.3 Policy Specification and Enforcement

In this section, we show how policy can be implemented through the composition and configuration of Antigone mechanisms. The policy layer allows selection from a wide range of policies roughly corresponding to the taxonomy presented in Section 3. While this section illustrates the ways in which one can implement policy, Antigone's mechanisms and architecture are in no way restricted to those defined by the policy layer. Applications that require custom policies can, of course, implement their own through direct integration with the mechanisms layer.

In the policy layer, group policies are embodied by the *policy block* data structure. Stated by the session leader, a session specific policy block is distributed to each group member at the time at which they are authenticated. Subject to session leader authentication, this policy is accepted without negotiation. We note

several ongoing works investigating policy negotiation [49, 21], but defer their analysis to future work. The policy block contains the following six fields:

GG The GG field is used to specify the sensitivity of the rekeying process to membership events. The policies (`TIME_SENS`, `LEAVE_SENS`, `JOIN_SENS`, and `MEMBER_SENS`) correspond to the definitions presented in Section 3.1. Table 1 describes these policies in terms of their sensitivity to membership events.

RK The RK field is used to specify the lifetime of session keys. Current session key lifetimes are measured by wall-clock time.

SG The SG field is used to specify one or more security guarantees to be applied to application messages. The available guarantees are: *confidentiality*, *integrity*, and *sender authenticity*. A side effect of the selection of several of these guarantees is that application messages will have the *group authenticity*.

MM The MM field states the group membership policy (distribution of group views). If membership information is specified by MM, group views are distributed during every rekeying operation and session key retransmit response.

FP The FP states the group failure policy. The current version of Antigone supports the detection and recovery of fail-stop failures only. If the boolean valued FP policy states failures are to be detected, heartbeat messages are transmitted periodically.

HB The HB field is used to specify the periodicity of heartbeat messages, if any.

Note that, for proper operation, several policies are dependent on guarantees provided by other policies. For example, a failure-sensitive membership policy requires the presence of failure detection. In the absence of a failure detection policy, group views may violate the membership policy by incorrectly indicating the presence of failed or disconnected processes. Currently, there is no enforcement of policy dependencies in the policy layer. Users of this layer are advised to carefully consider dependencies during the creation of a group policy.

The following text describes in detail how these group policies are implemented through the underlying mechanisms.

4.3.1 Rekeying Policy

The rekeying policy states when the group security context is to be changed (i.e. a new session key is distributed). Based on the rekeying sensitivity and session key lifetime indicated by the stated policy, the policy layer initiates rekeying after the observation of each relevant event (e.g., a member join, expiration of a session key). Described in Table 1, the group rekey policy (GG) determines after which membership events the session leader rekeys the group. In time-sensitive groups, the session leader rekeys the group after the expiration of the timer associated with the RK field.

RK states the maximum lifetime (in seconds) of each session key. In groups that do not desire limited-lifetime rekeying, RK is set to 0. Upon reception of a session key with a limited-lifetime, each member resets a key lifetime timer to RK. If the timer expires before a new session key is received, the member considers the current session key expired. The member then requests that the session leader send a new session key or exits the group. In the normal case, the session leader will rekey a session prior to the expiration of this timer.

Through minor modification of the existing policy layer, developers can implement a number of other rekeying policies. For example, the session leader can rekey only when certain members join or leave the group, or implement limited-lifetime rekeying based on bytes transmitted.

A policy issue arises during the transition of session keys. During a rekey, application data such as continuous media may continue to be broadcast. Because of delays in the delivery of the session key, a member may receive a message encrypted with a session key that it does not yet or will never possess. We present several solutions below. Note that this shortcoming is not unique to Antigone; any system without distributed commitment is required to address key transition.

The session key transition problem is difficult to address with a single general solution. In applications with low throughput characteristics, it may be reasonable to buffer data until the new key arrives. High performance applications, such as video-conferencing, have much higher throughput requirements and thus can not buffer data for long. Buffering is a reasonable solution only if the delay is likely to be short. A second solution is to establish a waiting period during which all transmitted data is encrypted under the old session key. Here, a newly joined member is unable to receive session content until the waiting period is over. Similarly, during the wait period, exited, failed, and ejected members are able to continue receiving session content. A third solution is to transmit each message under both the old and new session keys for some (presumably short) period. In this approach, the sender encrypts a one-time per-message random key under both the new and old session keys. The message itself is encrypted under random key. The ciphertexts of the random key and message would then be broadcast to the group in each message. A receiver would then reverse the process with an available session key. As in the wait period, this solution does not protect the group from members who have exited, failed, or been ejected.

After weighing these solutions, we have chosen to drop any message for which the member does not currently possess the key. In the average case delays will be acceptable, yielding little data loss. Recovery from lost data is left to the application. However, after obtaining more practical experience with Antigone, we may choose to reassess this decision.

A related problem occurs when received messages are encrypted with previous session keys (i.e. senders have not received a new session key). The acceptance of such messages could compromise group security. However, because of message timing in existing group applications, such situations are likely to occur in practice. Thus, we must weigh the security of the group against packet loss. The current version of Antigone implements a conservative solution in which all messages received with previous session keys are dropped.

4.3.2 Membership Policy

The membership policy states requirements for the secure distribution of group views. If this policy states membership is required, views are presented during session key distribution and rekeying. Thus, the accuracy of the group view is guaranteed only at the time at which the group is rekeyed. Similarly, the form of group membership guarantee is strictly determined by the rekeying policy. For example, time-sensitive groups (with no membership sensitivity) provide a best-effort rekeying, and membership-sensitive groups provide perfect membership.

4.3.3 Process Failure Policy

Once the underlying failure detection mechanism has been configured (as dictated by the FP and HB policy values), the policy layer plays no further role in the processing of failures. All heartbeat messages and retransmit requests are handled transparently by the underlying mechanisms layer. However, in the event of an unrecoverable error (e.g., loss of group communication), the policy layer dictates the course of action. In the current implementation, unrecoverable errors are reported to the application for user notification and the member is disconnected from the group.

Failure protection is useful in conjunction with a range of rekeying policies. For large groups, rekeying can be more expensive than the generation and transmission of session leader heartbeats. In such cases, the session leader can set the heartbeat interval to be lower than the rekey interval. The lower heartbeat interval

ensures that members do not use an old key beyond that interval. In the absence of heartbeats, the rekey interval bounds the time over which an old key may be used.

4.3.4 Data Security Policy

The data security policy defines the set of guarantees to be applied to application level messages. It is up to the application to make judicious use of the available guarantees, depending on application requirements. The performance associated with the various data security policies are described in Section 5.

The current policy layer software applies the group data security policy (SG) to all application level messages. In many contexts, it may be desirable for applications to determine those guarantees to be applied to each message independently. As necessary, the policy layer may be modified to provide per-message data security policies.

4.3.5 Access Control Policies

Access control in the policy layer is performed at the time at which each member is authenticated. Based on a pre-configured access control list (ACL) at the session leader, the authentication request will be accepted or rejected. A member who receives an acceptance will continue by joining the group in the manner defined by the group policy. A member who is rejected is expected to refrain from further communication with the group.

4.4 Communication Services

Multicast services have yet to become globally available. As such, dependence on multicast would likely limit the usefulness of Antigone. Through the broadcast transport layer, Antigone implements a single group communication abstraction supporting environments with varying network resources. Applications identify at run time the level of multicast supported by the network infrastructure. This specification, called a *broadcast transport mode*, is subsequently used to direct the delivery of group messages. The broadcast transport layer implements three transport modes; *symmetric multicast*, *point-to-point*, and *asymmetric multicast*.

The symmetric multicast mode uses multicast to deliver all messages. Applications using this mode assume complete, bi-directional multicast connectivity between group members. In effect, there is no logical difference between this mode and direct multicast.

The point-to-point transport mode emulates a multicast group using point-to-point communication. All messages intended for the group are unicast to the session leader, and relayed to group members via UDP/IP [50]. As each message is transmitted by the session leader to members independently, bandwidth costs increase linearly with group size. In some applications, these costs may be prohibitive. For example, a group of even modest size would have difficultly in maintaining a video transmission with reasonable frame rates.

In [5], we describe our experiences with the deployment of the *Secure Distributed Virtual Conferencing* (SDVC) application. This video-conferencing application is based on an early version of Antigone. The deployed system was to securely transmit video and audio of the September 1998 Internet 2 Member Meeting using a symmetric multicast service. The receivers (group members) were distributed at several institutions across the United States. While some of the receivers were able to gain access to the video stream, others were not. It was determined that the network could deliver multicast packets towards the receivers (group members), but multicast traffic in the reverse direction was not consistently available (towards the session leader). The lack of bi-directional connectivity was attributed to limitations of the reverse routing of multicast packets. We present significant technical detail of this issue in [5].

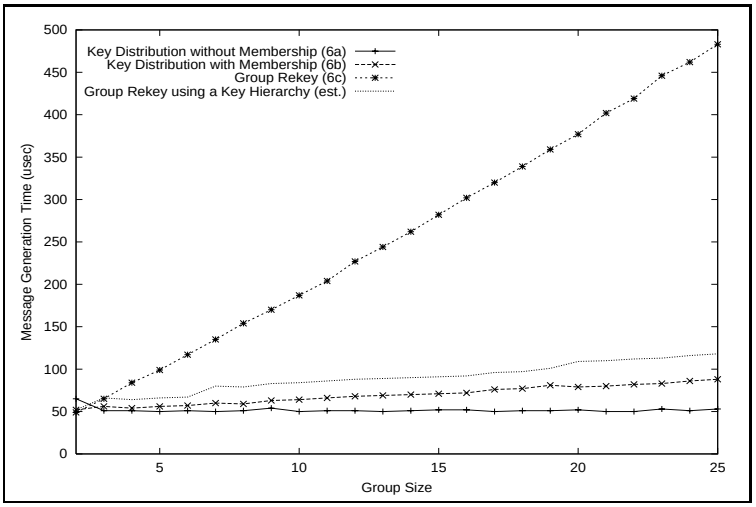

Figure 4: Group Membership/Session Key message generation costs - Measured and estimated costs associated with the generation of key distribution messages in groups of varying sizes and policies.

The limited availability of bi-directional multicast on the Internet coupled with the costs of point-to-point multicast emulation lead us to introduce *asymmetric multicast*. This mode allows for messages emanating from the session leader to be multicast, and all other message to be relayed through the session leader via unicast. Members unicast each group message directly to the session leader, and the session leader retransmits the message to the group via multicast. Thus, we reduce the costs associated with point-to-point group emulation to a unicast followed by a multicast.

5 Performance

This section presents results of a performance study investigating the costs associated with several important security policies. In Antigone groups, key management and application messaging are likely to consume the vast majority of network and computational resources. Thus, we study the costs associated with key distribution messages under several rekeying policies, and identify the latency and throughput characteristics of application messages under several data security policies. From this analysis, we are able to construct a rough profile of Antigone group performance.

The experiments described in this section were performed on Intel 200MHz Pentium Pro workstations running FreeBSD kernel version 3.0. All tests were executed on an unloaded 100 MBit Ethernet LAN. Application messaging experiments were conducted in an environment containing a single sender and nine receivers. Throughput is measured over 1 KB messages. Latency is measured over 10 KB messages. All tests use standard 56 bit DES keys and 512 bit RSA key pairs for symmetric and public key cryptography, respectively.

Fig. 4 shows the cost of group membership/session key message generation under varying policies and sizes. Because the contents the session key distribution message 6a is independent of group membership, the cost of generation is constant. Message 6a requires the generation of one session key distribution block and *Message Authentication Code* (MAC). The costs associated with the generation of key distribution message 6b increase slightly with group size. This trend can be attributed to the increasing amount of data to be hashed (group view). Thus, as the group membership grows, the costs associated with MAC generation increase. The costs associated with the generation of the session rekey message 6c increase linearly with

Policy	Throughput	Latency
Integrity	2577 KB/sec	5710 usec
Confidentiality	1697 KB/sec	8698 usec
Integrity and Confidentiality	1577 KB/sec	9037 usec
Sender Authenticity (est.)	71 KB/sec	14000 usec

Table 2: Application Messaging Performance - Measured throughput and latency of messaging policies.

group size. A session key block is generated for each member, requiring a distinct cryptographic operation. Similar to message 6b, the cost of MAC generation increases with group size. However, increases in the cost of message generation due to MAC construction is significantly less than increases due to session key block construction.

For comparison, we estimate the rekey message generation costs associated with a key hierarchy approach in Fig. 4. Rekeying in key hierarchies is performed by the distribution of a number of keys which is roughly proportional to the log of the group size. Therefore, we see little increase in message cost as the group becomes larger. This further indicates that key hierarchies can significantly improve the performance of rekeying in large groups. The cost of rekey message processing by receivers is slightly larger in key hierarchies ($O(\log n)$) than in the current Antigone mechanism ($O(1)$).

In all these rekeying messages, the size of each message mirrors its generation costs. For a group of size n, the size of message 6a is 80 bytes, message 6b is $80 + (n * 16)$ bytes, 6c is $40 + (40 * n)$ bytes, and the estimated size of a key hierarchy based key distribution message is $40 + (40 * log(n))$ bytes. Thus, the bandwidth costs of key distribution in the current Antigone protocols increase linearly with group size. However, it is expected that these costs will be small with respect to application traffic.

Note that as the UDP protocol is used for delivery, message size will have an effect on the reliability of key distribution. Long messages will become fragmented, increasing the probability that some fragment is lost. In UDP, no attempt is made to retransmit lost datagrams. In these cases, Antigone is required to fall back on its own retransmit mechanism. Additionally, in exceptionally large groups, the size of a key distribution message may exceed the maximum datagram size. In the future, we will investigate alternative key distribution mechanisms that address these and other limitations.

Table 2 shows the throughput and latency characteristics of several application message policies. Although not surprising, our experiments show that each policy has a significant affect on message performance; stronger policies have less throughput and greater latency. These policies, in order of increasing cost, are *INTEGRITY* (message 7a), *CONFIDENTIALITY* (7b), *INTEGRITY* and *CONFIDENTIALITY* (7c), and *SENDER_AUTHENTICITY* (7d).

Currently, we have not implemented the sender authenticity application message policy, but estimate its performance. We estimate sender authenticity by the measured cost of MAC generation. In the current protocol, generating a MAC with sender authenticity requires a single 512 bit private-key encryption. As these operations will likely dominate the costs of message processing, they serve as a reasonable estimate of performance.

The performance of application policies costs mirrors the speed of the underlying cryptographic operations. We use DES [40] to achieve confidentiality, and MD5 [38] to achieve integrity. Our implementation uses the SSLeay v0.9.0b [51] `crypto` library. On the test machine, we found that DES encryption (≈ 3.7 Mbyte/second) is about $1/6$ the speed of MD5 hashing (≈ 21 Mbyte/second). The RSA algorithm could encrypt 71 512 bit blocks a second (≈ 35 Kbyte/second).

6 The Antigone API: A Brief Example

This section illustrates the API and use of policy in Antigone through an example audio-conferencing application. In this application, an audio conference is initiated by some user serving as the session leader. The existence and parameters of the conference is advertised by a service external to the application (e.g., SDP [36]). Conference participants apply for admittance to the group, and if accepted, transmit, receive, and playback audio data. The policy under which a session operates is strictly determined by the type of conference and its participants. In the following text, we give a brief overview of the the Antigone *Applications Programming Interface* (API) by describing its use in the audio conferencing application.

Written entirely in C++, the Antigone API consists of 40 classes and over 18,000 lines of source code. The API has been ported to Linux kernels version 1.2+, 2.0+, FreeBSD Kernel version 3.0, and Sun OS 5.6. Support for Microsoft Windows 98 and Windows NT is planned for future versions of Antigone. Recent versions of the source code and related documentation can be retrieved from `http://antigone.citi.umich.edu`.

Fig. 5 presents source code for a simplified `main()` function of the audio conferencing application described above. Illustrated in the source code, the use of Antigone within an application can be described in four phases; the definition of group policy, group creation and initialization, the execution of the application, and session shutdown. Note that the policy layer is used to implement this conferencing application. As needed, other policies may be implemented through direct integration with the Mechanisms layer.

Created by the session leader and subsequently delivered to group members during authentication, the AntiPolicy object embodies group policy. The attributes of this object define the policy parameters described in Section 4.3 and are used by the policy layer to direct group operation. In any flexible application, the construction of this object should be determined by application context. Thus, for flexibility, an interface for user specification of its parameters is desirable. The example application defines policy based on command line parameters. For brevity, we omit details of the policy object, relegating its construction to the application specific `ProcessCommandLineParameters()` function. Although not strictly part of the security policy, user specification of the broadcast transport mode (`btPol`), member identity (`memberName`), and session address information is also required.

Instantiated by each group member, the `PolicyLayer` object encapsulates an Antigone session. Group policies and session parameters are communicated to Antigone through the associated object constructor. All group relevant operations are performed through interfaces (object methods) defined on the session object. Where relevant, the construction of the session objects requires a long term secret key (as registered with the TTP) (`ltsKey`), some application defined error handling information (`errorCallBack, cbData`), the group broadcast transport mode (`btMode`), and session and TTP address information. In the case of the session leader, the group policy (`antiPolicy`) is also specified.

Antigone errors are reported to an application through callback functions. An application specific callback object and associated data are created and passed to the session object during construction. Subsequent errors are reported, with identifying information, through the callback function. Passed to the callback function during error processing, the callback data is used to provide additional context where the existence and identity of an error are insufficient.

The construction of a `PolicyLayer` object automatically initiates communication with the group. In the case of the session leader, the appropriate network interfaces are initialized, a session key is generated, and group information initialized. In the case of a group members, after initializing the appropriate network interfaces, the group member is authenticated by the session leader and joins the group. Errors are reported through the callback interface.

The `PolicyLayer` object provides interfaces similar to those available to BSD sockets [52]. Group messages can be sent or received through the (`readMessage()`) and (`sendMessage()`) methods, re-

```
// Functional Prototypes
void errorCallBack( void *pp, char *text );

// Main Function of Audio Conferencing Application
int main( int argc, char **argv )
{
    // Get name, policies, transport mode from command line arguments
    AntiPolicy antiPolicy;
    char *memberName, cbData = &memberName;
    unsigned long saddr, taddr, mcaddr;
    short btMode, sport, tport, mcport;
    ProcessCommandLineParameters( argc, argv, antiPolicy, &memberName, btMode,
                                  sport, saddr, tport, taddr, mcport, mcaddr );

    // Obtain the long term secrect registered with the TTP through
    // user password dialog
    SecKey ltsKey = GetKeyFromUser();

    // Create the Antigone object
    PolicyLayer *antiPol = NULL;
    if ( issrvr )
         // Create a session leader
        antiPol = new PolicyLayer( ltsKey, errorCallBack, cbData, btMode,
                                   mcaddr, mcport, sport, taddr, tport,
                                   antiPolicy );
    else
         // Create the group member
        antiPol = new PolicyLayer( ltsKey, errorCallBack, cbData, btMode,
                                   mcaddr, mcport, saddr, sport );

    // Loop until done with session
    AntiMessage *msg = NULL;
    while ( ! done )
    {
        // Read select for 1 second
        if ( readSelectOnAntigoneSockets() != 0 )
        {
            // Get audio data message from Antigone, play it
            antiPol->readMessage( &msg );
            playbackAudioSample( msg );
        }

        // If audio data to transmit, xmit it
        if ( msg = getNextAudioSample() != NULL )
            antiPol->sendMessage( msg );
    }

    // Leave the current group, cleanup keys and data
    antiPol->Quit();
    delete antiPol;

    // Return succesfully
    return( 0 );
}
```

Figure 5: Antigone API - The `main()` function for an example audio conferencing application.

spectively. Interfaces are provided to initialize the file descriptor sets used in `select()` calls. The message handling functions of Antigone process `AntiMessage` objects. These objects implement flexible message buffers by handling heap operations and bounds checking, and provide interfaces for the insertion and extraction of primitive data types. Developers can avoid many of the complexities associated with buffer management in application level protocols through the use of `AntiMessage` objects.

The conferencing application defines two functions used to process audio data, `playbackAudioSample()` and `getNextAudioSample()`. As can be expected, these functions read and write audio data to the audio hardware. During the session, group messages encapsulating audio data are sent and received as determined by session traffic, and processed through the audio processing functions.

While a member is actively participating in a session, the handling of events (e.g., recoverable failures, member joins) is transparently performed by the underlying Antigone software. However, a number of interfaces to group state are available. By accessing group membership, keying material, and policies, the application may react to changes in group state.

When a user wishes to leave the group, the *Quit()* method is called. Subsequent destruction of the session object zeros and deletes all security relevant keys and data. The application can then exit or create a new session object as desired.

7 Using Policy

The are a myriad of factors that can contribute to an application policy. One must weigh the (sometimes conflicting) user requirements and environmental constraints in assessing the needs of a given session. The tradeoffs made during this assessment define the group policy, and indirectly the quality of service provided to its participants. It may be discovered that a session's requirements cannot be met with the available infrastructure. Users making this discovery must either reassess their requirements, introduce additional infrastructure, or forgo the session. Below, we indicate several factors that can affect a security policy, and illustrate the use of policy in several session environments.

Performance is often a significant determinant of an application security policy. Each policy should be able to achieve the needed security without violating an application's minimal performance requirements. As noted in Section 5, some policies are inherently expensive to provide. For example, using current solutions, guaranteeing per-packet sender authenticity in high bandwidth applications (e.g., video broadcasts) is difficult without special purpose hardware. However, as network and computer technologies advance, performance will likely play a lesser role in determining security policy.

The semantics of an application's content also contributes greatly to the construction of session policies. Some security policies are clearly not appropriate for certain sessions. For example, a confidentiality policy does not make sense in a public chat group. However, an authenticity policy is relevant independent of the public nature of the session.

Limited availability of network and security infrastructures can restrict group policy. Some policies are simply not achievable or are prohibitively expensive to implement in the absence of required services. If a stated policy requires a service unavailable to some set of users, those users cannot securely participate in the session.

Any number of other factors may contribute to application policy. For example, certain cryptographic algorithms are prohibited from use or export by some countries. Thus, any policy applied to group (even partially) contained in such a country must avoid policies requiring the illegal cryptography. Social factors also may contribute to group policy. For example, an enterprise can state a minimum set of security to be applied to all inter-office conferencing.

There is no single set of factors that contribute to application policy. Each session must define its behaviors with an understanding of the needs and capabilities of the participants and content. Because

Policy	Session 1	Session 2	Session 3
Rekeying	membership	limited_lifetime	join
Membership	yes	no	yes
Failure	yes	yes	yes
Data Security	confidentiality, integrity	confidentiality	integrity
Transport	symmetric	point-to-point	asymmetric

Table 3: Session Policies - These policies define the operation of audio conferencing application for three distinct sessions.

policy is a direct derivative of this understanding, there is no "one-size fits all" policy for an application.

Below, we describe the use of policy in the example audio conferencing application defined in Section 6 in three distinct session environments. Note that each policy can be implemented by Antigone without modification of the application's executable. An overview of the policies for these sessions are presented in Table 3.

Session 1 Sales Conference - A sales conference gathers the sales force and customers of company X together to discuss some subject of mutual interest. As the conference is sponsored by X, its representatives will always be present. At any given point, there may be 0 or more customers present. Primarily, the conference participants wish to protect the session from competitors, and where customer private information is discussed, other customers.

In this conference, it is necessary to limit the exposure of the audio content to only those parties present. Thus, the group implements a membership-sensitive rekeying policy. For similar reasons, the audio should be protected from eavesdropping or modification by malicious parties (through confidentiality and integrity data security policies). It is also important that accurate membership views be distributed (membership policy), and that notification of failed or disconnected members be timely (failure protection policy).

Because X and its customers are likely to have significant IT resources, it is reasonable to assume that bi-directional multicast will be available (possibly through multicast tunneling [53]). Thus, the group can achieve near optimal throughput and low latencies through a symmetric multicast transport policy.

Session 2 Internet Telephony - Initiated by a private user on the Internet, this session allows a two or more parties to conduct personal business. Within the session, the parties are assumed to have had some previous contact and implicitly trust each other. The participants in these sessions typically want only to protect content from outside parties.

As the participants know and trust each other, there is no need to protect content from past or future members. Thus, membership-sensitive rekeying is not necessary. However, to prevent cryptanalysis of the session key, limited-lifetime rekeying is used. The threat model implied by this session states that the confidentiality of content is the only data security requirement. Because the parties are likely to know each others voice, and would be able to immediately detect bad content, integrity and authenticity of content is not required. Group membership information need not be distributed, but a failure protection policy may be useful for detecting participant crashes or network failures.

Typically, the Internet Telephony sessions initiated by private users occur over dialup links through Internet Service Providers (ISPs). As multicast is not widely deployed by ISPs, the group must use the emulated multicast services implemented by a point-to-point transport policy.

Session 3 Internet Broadcast - In this session, a broadcaster wishes to publicly transmit an audio stream to a large group of Internet users. The intended audience consists of a set of users who have established some relationship with the broadcaster. Receivers will not send audio data, only receiving broadcast content. We

state that, for this particular broadcast, content authenticity is the central security requirement.

The broadcaster may desire to know the identity of the listeners over the course of the broadcast, but may not need timely membership views. Thus, the group does not provide membership information. Because protection from past or future members is not required, the group implements a limited-lifetime rekeying policy. Similarly, the broadcaster may not care that the parties external to the group are able to eavesdrop content, and only wish to ensure that authentic content is delivered to all valid users. These (group) authenticity and integrity requirements are met through the integrity data security policy. Because of the cost of implementing failure detection in the potentially large group, it is not enabled.

As we found in our experiences deploying SDVC [5], establishing bi-directional multicast connectivity to a large number of sites on the Internet is difficult. However, because of the expected group size, the use of multicast can significantly improve session performance. Thus, the broadcaster may choose to use asymmetric multicast. As the session leader is the only party transmitting application content, the overhead associated with asymmetric multicasts will be negligible.

8 Conclusions

The Antigone framework presented in this paper provides flexible interfaces for the definition and implementation of a wide range of secure group policies. Antigone enabled applications implement policy through the composition and configuration of security implementing *mechanisms*. Thus, Antigone does not dictate the available policies, but provides facilities for building them.

As we have discovered, there are as many distinct policies as there are applications and environments. We have investigated and documented the critical policy requirements of existing group based applications and frameworks. It is from this analysis that the set of mechanisms currently defined by Antigone was synthesized.

The Antigone mechanisms represent the set of facilities required by secure groups. These mechanisms implement facilities for session key management, data security, membership management, failure detection and recovery, and access control. Applications use these mechanisms to construct a feature set specific to the session context and the assumed threat model. Each mechanism provides simple, but substantive, features needed by various security policies. Through composition, the mechanisms can implement highly complex group policies. However, these mechanisms are not restricted to any particular implementation. They may be modified as needed to implement new technologies and policies easily.

We have identified an efficient approach to detect process failures in group communication. Using our *secure heartbeats*, we can amortize the cost of keep-alive generation over many messages. Unlike many other approaches to failure detection in groups, no assumptions about the availability of reliable communication or public key certificates are needed.

In investigating the security requirements of Antigone groups, we have constructed a suite of general purpose security policies. These policies represent those that have been found useful or have been suggested as being useful by various group communication systems. Through the selection of primitive policies, over 96 distinct kinds of secure groups can be implemented. These policies can be further refined through selective parameters such as time-outs, acceptable participants, etc. Antigone groups can implement the security found in the overwhelming majority of existing secure group frameworks through the use of these policies.

As support for multicast services on today's networks is inconsistent, Antigone provides an abstract interface for unreliable group communication. Using this abstraction, applications can avoid dependence on multicast, but where available, realize its performance. In deploying multicast-based solutions, we have found that though multicast connectivity in one direction is often possible, achieving bi-directional multicast is more difficult. As a result, we introduce a transport mode called *asymmetric multicasting*. In asymmet-

ric multicasting, messages emanating from a single source use multicast, and all others use unicast. Thus, the cost of emulating bi-directional multicast is one additional unicast per message. Antigone's implementation also provides interfaces for symmetric multicast (bi-directional) and point-to-point (unicast) group communication.

Our initial performance study indicates that as security requirements increase, so do performance costs. This is not a surprising result, but indicates the need for infrastructures that support policies appropriate for sessions with varying performance requirements.

Any number of factors can contribute to the selection of a security policy. Issues such as performance, application semantics, service availability, and participant trust can affect the kinds of security appropriate for a given session. One must weigh these factors in constructing a coherent security policy. Ultimately, the tradeoffs between security policies cannot be made by developers alone. Often, only users of software are able to completely articulate security requirements.

9 Acknowledgements

We would like to thank Peter Honeyman, Hugh Harney, Sugih Jamin, Trent Jaeger, and Brian Noble for their many helpful comments. We would also like to thank William A. (Andy) Adamson, Charles J. Antonelli, and Kevin Coffman for their assistance in the development of the Antigone system, and Weston A. (Andy) Adamson for his help in constructing and maintaining the Antigone web-site.

References

[1] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A Flexible Framework for Secure Group Communication. In *Proceedings of 8th USENIX UNIX Security Symposium*, pages 99–114. USENIX Association, August 1999. Washington D. C.

[2] S. Deering. Host Extensions for IP Multicasting. *Internet Engineering Task Force*, August 1989. RFC 1112.

[3] Network Research Group, Lawrence Berkeley National Laboratory. `vic` - Video Conferencing Tool, July 1996. `http://www-nrg.ee.lbl.gov/vic/`.

[4] P. D. McDaniel, P. Honeyman, and A. Prakash. Lightweight Secure Group Communication. Technical Report 98-2, Center for Information Technology Integration, University of Michigan, April 1998.

[5] A. Adamson, C.J. Antonelli, K.W. Coffman, P. D. McDaniel, and J. Rees. Secure Distributed Virtual Conferencing. In *Proceedings of Communications and Multimedia Security (CMS '99)*, September 1999.

[6] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[7] R. Van Renesse, K. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4):76–83, April 1996.

[8] L. Gong. Enclaves: Enabling Secure Collaboration Over the Internet. In *Proceedings of the Sixth USENIX Security Symposium*, pages 149–159. USENIX Association, July 1996.

[9] M. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of 2nd ACM Conference on Computer and Communications Security*, pages 68–80. ACM, November 1994.

[10] S. Mittra. Iolus: A Framework for Scalable Secure Multicasting. In *Proceedings of ACM SIGCOMM '97*, pages 277–278. ACM, September 1997.

[11] Debby M. Wallner, Eric J. Harder, and Ryan C. Agee. Key Management for Multicast: Issues and Architectures (*Draft*). *Internet Engineering Task Force*, September 1998. `draft-wallner-key-arch-01.txt`.

[12] C. K. Wong, M. Gouda, and S. S. Lam. Secure Group Communication Using Key Graphs. In *Proceedings of ACM SIGCOMM '98*, pages 68–79. ACM, September 1998.

[13] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Specification. *Internet Engineering Task Force*, July 1997. RFC 2093.

[14] Hugh Harney and Eric Harder. Group Secure Association Key Management Protocol (*Draft*). *Internet Engineering Task Force*, April 1999. `draft-harney-sparta-gsakmp-sec-00.txt`.

[15] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. *Internet Engineering Task Force*, November 1998. RFC 2401.

[16] A. Ballardie. Scalable Multicast Key Distribution. *Internet Engineering Task Force*, May 1996. RFC 1949.

[17] T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees (CBT). In *Proceedings of ACM SIGCOMM '93*, pages 85–95. ACM, September 1993.

[18] M. Blaze, J. Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, 1996. Los Alamitos.

[19] M. Blaze, J. Feignbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System - Version 2. *Internet Engineering Task Force*, September 1999. RFC 2704.

[20] Y. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust Management for Web Applications. In *Proceedings of Financial Cryptography '98*, volume 1465, pages 254–274, Anguilla, British West Indies, February 1998. Springer-Verlag.

[21] L. Sanchez and M. Condell. Security Policy System (*Draft*). *Internet Engineering Task Force*, November 1998. `draft-ietf-ipsec-sps.txt`.

[22] T. Hardjono, R. Canetti, M. Baugher, and m P. Dinsmore. Secure Multicast: Problem Areas, Framework, and Building Blocks (*Draft*). *Internet Engineering Task Force*, October 1999. `draft-irtf-smug-framework-00.txt`.

[23] N.C. Hutchinson and L.L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1994.

[24] M. Hiltunen and R. Schlichting. A Configurable Membership Service. *IEEE Transactions on Computers*, 47(5):573–586, May 1998.

[25] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Architecture. *Internet Engineering Task Force*, July 1997. RFC 2094.

[26] Postel J. Transmission Control Protocol - DARPA Internet Protocol Program Specification. *Internet Engineering Task Force*, September 1981. RFC 793.

[27] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, pages 784–803, December 1997.

[28] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[29] S. Even, O. Goldreich, and S. Micali. On-Line/Off-Line Digital Signatures. *Journal of Cryptology*, 9(1):35–67, 1996.

[30] R. Gennaro and P. Rohatgi. How to Sign Digital Streams. In *Proceedings of CRYPTO 97*, pages 180–197, August 1997. Santa Barbara, CA.

[31] Sape Mullender. *Distributed Systems*. Addison-Wesley, First edition, 1993.

[32] P. Zimmermann. PGP User's Guide. Distributed by the Massachusetts Institute of Technology, May 1994.

[33] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, pages 33–38, September 1994.

[34] Bob Briscoe and Ian Fairman. Nark: Receiver-based Multicast Non-repudiation and Key Management. In *Proceedings of E-commerce '99*, Denver, Colorado, June 1999. ACM.

[35] M. Handley. The sdr Session Directory: An Mbone Conference Scheduling and Booking System. Department of Computer Science, University College London, April 1996.

[36] M. Handley and V. Jacobson. SDP: Session Description Protocol. *Internet Engineering Task Force*, April 1998. RFC 2327.

[37] T. Leighton and S. Micali. Secret-key Agreement without Public-Key Cryptography. In *Proceedings of Crypto 93*, pages 456–479, August 1994.

[38] R. Rivest. The MD5 Message Digest Algorithm. *Internet Engineering Task Force*, April 1992. RFC 1321.

[39] C. Adams and S. Farrell. RFC 2510, X.509 Internet Public Key Infrastructure Certificate Management Protocols. *Internet Engineering Task Force*, March 1999.

[40] National Bureau of Standards. Data Encryption Standard. *Federal Information Processing Standards Publication*, 1977.

[41] ANSI. American National Standard for Financial Institution Key Management. *American Bankers Association*, 1985. ANSI X.917.

[42] J. Kilian and P. Rogaway. How to Protect DES Against Exhaustive Key Search. In *Proceedings of Crypto '96*, pages 252–267, August 1996.

[43] R. 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.

[44] W. Diffie and M.E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[45] C. Wong and S. Lam. Digital Signatures for Flows and Multicasts. In *6th International Conference on Network Protocols*. IEEE, 1998. Austin TX.

[46] Leslie Lamport. Password Authentication with Insecure Communication. *Commuications of the ACM*, 24(11):770–772, 1981.

[47] N.M. Haller. The S/Key^{tm} One-Time Password System. In *Proceedings of 1994 Internet Society Symposium on Network and Distributed System Security*, pages 151–157, February 1994. San Diego, CA.

[48] Aviel D. Rubin. Independent One-Time Passwords. *USENIX Journal of Computer Systems*, 9(1):15–27, February 1996.

[49] P. Dinsmore, D. Balenson, M. Heyman, P. Kruus, C Scace, and A. Sherman. Policy-Based Security Management for Large Dynamic Groups: A Overview of the DDCM Project. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX '00*, pages 64–73. DARPA, January 2000. Hilton Head, S.C.

[50] J. Postel. User Datagram Protocol. *Internet Engineering Task Force*, August 1980. RFC 768.

[51] T. Hudson and E. Young. SSLeay and SSLapps FAQ, September 1998. `http://psych.psy.uq.oz.au/ftp/Crypto/`.

[52] W. R. Stevens. *Unix Network Programming*. Prentice Hall, ISBN 0 13 949876 1, Second edition, 1998.

[53] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. *Internet Engineering Task Force*, November 1988. RFC 1075.