Ismene: Provisioning and Policy Reconciliation in Secure Group Communication *

Patrick McDaniel Atul Prakash
Electrical Engineering and Computer Science Department
University of Michigan, Ann Arbor
pdmcdan{aprakash}@eecs.umich.edu

Abstract

Group communication systems increasingly provide security services. However, in practice, the use of such systems is complicated by the divergent requirements and abilities of group members. In this paper, we define a policy language called Ismene that directs the *provisioning* of security-related resources at member sites. The communication service is defined through a reconciliation of a *group policy* and member's *local policies* into a security configuration. Group authorization and access control appropriate for the operating conditions and session configuration are also defined within the policy. The use of Ismene policies to define security is illustrated through an extended example of a group application built on the prototype Ismene framework.

1 Introduction

Recent advances have addressed many of the difficult problems in secure group communication [36, 29]. However, a critical, but as yet largely unaddressed, problem is the reconciliation of the differing security requirements of group members. Group members can belong to different organizations that desire to place unique restrictions on participation and represent different capabilities. Security requirements can also differ from session to session, depending on the nature of the session and the environment in which it is conducted. Thus, the conditional requirements of all parties must be considered in constructing a secure group. In this paper, we define a policy language, called Ismene, that permits the specification and reconciliation of group and member security requirements.

In Ismene, a security policy specifies two central aspects that are important in group communication: *provisioning* and *authorization and access control*. Both aspects are specified in group and local polices. We define a *group policy* as the statement of the entirety of security-relevant properties, parameters, and facilities used to support a group session. Thus, a group policy states how security directs behavior, the entities allowed to participate, and the mechanisms used to achieve security objectives. This use of policy affords a degree of coordination; dependencies between authorization, access control, data protection, key management, 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.

other facets of a communication can be represented and enforced within a unifying policy. A member states the set of local requirements on group sessions through a *local policy*. The group and member local policies are *reconciled* within an environment prior to the establishment of each session.

The provisioning aspect of a security policy identifies the basic security requirements and the mapping of those requirements into a configuration of security-related services or mechanisms at member sites. Ismene group and local policies are *reconciled* to arrive at a specific configuration for a session. Potential participants of a session verify that the session's configuration is *compliant* with their local policy before participating. A group policy is tested against a set of legal usage assertions through *analysis* to ensure that any configuration resulting from reconciliation will not introduce negative side effects.

The authorization and access control aspect of a security policy defines how sessions regulate action within the group. The authorization and access control implemented by a group is explicitly stated in its configuration. Ismene is limited to expressions of positive criteria under which access is allowed, but permits the integration of other authorization frameworks where more expressive power is required.

Ismene focuses on the configuration of secure groups from the group and local policies. We define a group policy as the statement of the entirety of security-relevant properties, parameters, and facilities used to support a group session. Thus, a group policy states how security directs behavior, the entities allowed to participate, and the mechanisms used to achieve security objectives. This use of policy affords a degree of coordination; dependencies between authorization, access control, data protection, key management, and other facets of a communication can be represented and enforced within a unifying policy.

Policy has been used in different contexts as a vehicle for representing authorization and access control [37, 4, 8, 38, 33], peer session security [39], quality of service guarantees [6], and network configuration [35]. These approaches define a policy language or schema appropriate for their target problem domain. Ismene expands on this work by defining an approach in which policy is used to provision and regulate the services supporting communication, and to check the compliance of the group definition with local requirements.

Mechanism composition has long been used as a building block for distributed systems [28, 30, 2, 3, 11, 27]. Composition-based frameworks specify the compile or run-time organization of sets of protocols and services used to implement a communication service. The resulting software addresses the requirements of each session. However, the definition and synchronization of specifications is largely relegated to system administrators and developers. Our approach seeks to extend compositional systems by defining a language in which security requirements are consistently mapped into a system configuration.

The remainder of this paper describes the design and use of the Ismene. We have constructed a number of tools for the creation and analysis of Ismene specifications. The format and semantic of Ismene is detailed. We define and briefly discuss approaches for policy reconciliation, compliance checking, and analysis. We have extended a secure group communication framework [1] to implement group security from Ismene policies. Several non-trivial group applications have been developed on the augmented framework. The use of Ismene is illustrated through an extended example of one such application in the latter sections of this paper. We discuss a number of works related to policy in secure group communication.

2 Language Goals and Requirements

To motivate the goals of Ismene, the following presents simplified security requirements for an example group teleconferencing application, `tc`. The `tc` application is to be deployed within a company $widget.com$. $widget.com$'s organizational policy for `tc` requires the following:

- the confidentiality of all session content must be protected by encryption using DES or AES (provisioning requirement)
- participation in a session is restricted to $widget.com$ employees (access control requirement)

Now suppose $Alice$ wishes to sponsor a session of application `tc` that meets her following local policy:

- Alice wishes to use only AES cryptographic algorithm only (provisioning requirement); and
- she wishes to restrict the session to the $BlueWidgets$ team (access control requirement)

A basic requirement on the policy language is that it must be able to specify provisioning and access control policies at the application level as well as for each member, and resolve them into a specific session policy instance. In the above example, the result of such resolution is that Alice's session is restricted to members who are in both $BlueWidgets$ and $widget.com$ (access control requirement), and the cryptographic modules must be configured so that all content is encrypted using AES (provisioning requirement).

In general, security requirements can be more complex. For example, Alice may wish to restrict access to certain hours of the day, require that the session be rekeyed when new members join or leave, etc. Furthermore, other members in the group may have their own local security policies. Before a new member participates, it must be able to check whether the session's policy satisfies its local policy. If it does not, it can choose to abstain from the group rather than compromise its security policy.

Policy must also be responsive; changes in membership or the execution of a security-relevant action can affect the session configuration. Conversely, the session must be able to make access control decisions based on the use and configuration of security mechanisms.

Ismene, thus, has the following primary goals:

- *flexible provisioning* - Ismene must allow the provisioning of groups based on an assessment of run-time conditions and the local policies of the members.
- *action-dependent provisioning* - Ismene must allow the modification of session configuration based on changes in membership or the execution of a security-relevant action.
- *authorization and access control* - The authorization and access control embodied by a session must be explicitly, but flexibly, stated in Ismene. Authorization and access control should not only be based on operating conditions, but also on the current session configuration.
- *policy compliance* - Any member participating in a group must be able to assess compliance of a configuration with its local security policy.
- *legal usage analysis* - Ismene must be able to determine whether a configuration represents the proper usage of the underlying security mechanisms.

3 Ismene Policy Description Language

This section describes the design and use of the Ismene policy language. We begin in the following subsection by detailing the construction and processing of group security policies.

3.1 Approach

Depicted in Figure 1, a group is modeled as the collection of *participants* collaborating towards a set of shared goals. We assume the existence of a *policy issuer* with the authority to state session requirements. The issuer states the conditional requirements of future sessions through the *group policy*. Each member states the set of local requirements on future session through a *local policy*. Each participant trusts the issuer to create a group policy consistent with session objectives. However, a participant can verify the compliance of a policy *instance* with their *local policy*.

An *initiator* is an entity that generates an *instance* from group and local policies. The initiator may or may not be a participant of the group. An instance defines session provisioning and the rules used for authorization and access control. Provisioning is the result of the *reconciliation* of the group and local policies within the runtime environment. Through provisioning, an instance identifies relevant session requirements, and defines how requirements are mapped into a configuration. The initiator is trusted to evaluate the group and local policies correctly.

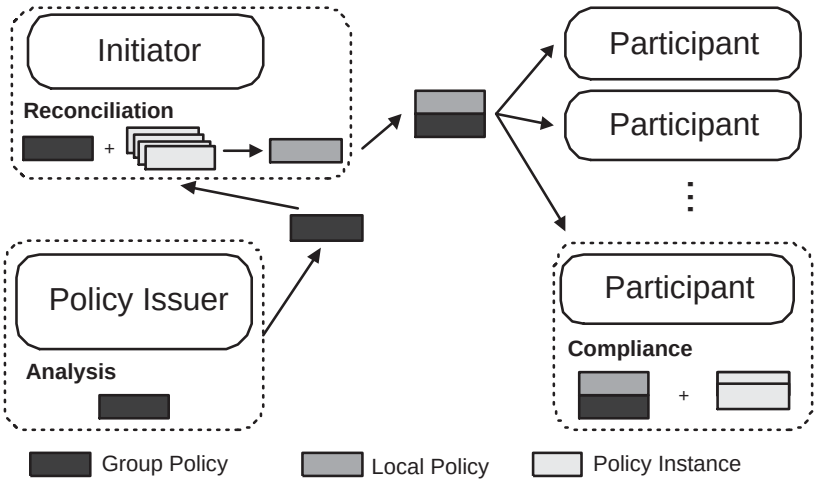

Figure 1: System Model - A session is a collection of *participants* collaborating towards some set of shared goals. A policy *issuer* states a group policy as a set of requirements appropriate for future sessions. The group and expected participant local policies are reconciled to arrive at a *policy instance* stating a concrete set of requirements and configurations. Prior to joining the group, each participant checks compliance of the instance with its *local policy*.

To simplify, Ismene policies are a collection of totally ordered provisioning and action clauses. *Provisioning clauses* identify configuration. Participant software is modeled as collections of security *mechanisms*. Associated with a mechanism is a set of *configuration* parameters used to direct its operation. Each mechanism provides a distinct communication service that is configured to address session requirements. A provisioning clause explicitly states configuration through a set of mechanisms and parameters.

Authorization and access control rules are defined in *action clauses*. An action clause defines the conditions under which a protected action should be allowed.

Each provisioning and action clause is defined as the tuple:

```
<tag> : <conditionals> :: <consequences>;
```

Tags are used to provide structure to the policy. Intuitively, tags represent session requirements or identify a protected action. The organization of tags dictates the relationships between clauses, and ultimately guides policy reconciliation. *Conditionals* contain zero or more predicates describing the conditions under which the consequences are to be enforced. Predicates are Boolean functions used to test the operating environment, session configuration, local or global state, or the presence of credentials. The result of the evaluation of a predicate is *true* where the conditional holds and *false* otherwise. *Consequences* identify provisioning and authorization. Each consequence states a session requirement, a configuration, or the acceptance of an access request. The Ismene grammar is described in Appendix A.

The semantic of a clause are as follows; if all the conditionals in the clause evaluate to **true**, then the consequences need to be enforced. For example, a clause containing conditionals $c_1 \ldots c_n$ and consequences $q_1 \ldots q_m$ represents the logical expression $(c_1 \wedge \ldots \wedge c_n) \Rightarrow (q_1 \wedge \ldots \wedge q_m)$. Thus, the result of an evaluation over a set of clauses is a conjunction of consequences. The collection of consequences defines precisely how the session's security parameters are allowed to be configured.

Clauses associated with a single tag are evaluated in the order in which they are defined in the policy. Evaluation of a tag stops when a clause evaluates to true (i.e., the conditionals hold). In this case, the consequences are enforced and all other clauses associated with the tag are ignored. If a clause evaluates to false then the next clause associated with the tag is evaluated.

Returning to the *widget.com* example described in the previous section, the network administrator responsible for setting application policy at *widget.com* acts as the policy issuer. The administrator generates the following group policy for the `tc` application:

```
provision : :: pick( config(idhdlr(conf=des)), config(idhdlr(conf=aes)) );
join : Credential(&cert,iss=$CA,subj.O=widget.com,subj.CN=$joiner) :: accept;
```

The `idhdlr` defined in the `provision` clause is a mechanism providing security guarantees over application content. The `conf=des` and `conf=aes` are configuration parameters applied to the `idhdlr` mechanism stating how confidentiality is to be provided. The `pick` configuration is used to state flexible policy; either DES [26] or AES [9] can be used to implement confidentiality, but not both or neither. The `join` action clause states that only entities supplying a certificate credential with subject organization of *widget.com* and issued by a (known and trusted) CA should be admitted to the group.

Local policies are used by each participant to describe local requirements on future sessions. Alice defines her local policy as follows:

```
provision : :: config(idhdlr(conf=aes));
join : Credential(&cert,iss=$CA,subj.O=BlueWidgets,subj.CN=$joiner) :: accept;
```

Through this local policy, Alice states that any session must implement a confidentiality policy using the `idhdlr` mechanism implementing AES. The `join` clause states the session must enforce a policy that requires participants supply a certificate issued by a trusted CA with a subject organization of *BlueWidgets*.

Alice acts as the initiator in sponsoring the `tc` session. She acquires the local policies from the expected session participants. For this example, only the group policy and Alice's local policy are used. The instance resulting from reconciliation is as follows:

```
provision : :: config(idhdlr(conf=aes));
join : Credential(&cert,iss=$CA,subj.O=widget.com),
       Credential(&cert,iss=$CA,subj.O=BlueWidgets,subj.CN=$joiner) :: accept;
```

The reconciliation of the `tc` group and Alice's local policies attempts to find a configuration that is consistent with both policies. In this case, the configuration `config(idhdlr(conf=aes))` is selected. The use of two credential conditionals in the `join` clause represents a conjunction; credentials fulfilling the criteria for each condition must be supplied to gain access to the group.

The $CA descriptor in the above examples identifies an *attribute* defining the public key of a known and trusted certificate authority. An *attribute* describes a single or list-valued invariant. For example, the following attributes define a single-valued version number and list-valued ACL:

```
version := < 1.0 >;
JoinACL := < {bob}, {john}, {george} >;
```

The occurrence of the symbol "$" in any clause signifies that the attribute should be replaced with its value. The *attribute set* is the set of all attributes. An application can add to the attribute set by passing name/value (list) pairs to the Ismene algorithms.

Prior to their use in any session, group policies are *analyzed* to determine if they represent legal configurations. Legal configurations are stated through *assertions*. Assertions represent invariant properties required by all instances. Logically, these statements identify illegal and mandatory configurations. For example, the assertion:

```
assert: config(keymgt(mem=leavesens)) :: config(membership(leave=explicit));
```

states that a key management mechanism configured with leave sensitivity [25] requires (is dependent on) the membership mechanism configured to provide explicit leaves. Thus, for this assertion to hold, any instance defining the leave sensitive key management must also define the membership mechanism with explicit leaves. Through *analysis*, Ismene guarantees that no instance resulting from the reconciliation of the group policy with any set of local policies will violate policy assertions.

Each potential participant acquires the policy instance prior to joining a group. The participant determines the consistency of the instance with its local policy through a *compliance* algorithm. A *late joiner* (i.e., a members whose local policy was not considered during the creation of the instance) is free to participate if the instance complies with their local policy. Participants in the `tc` session check the compliance of the instance received from Alice, and if successful, can participate in the session. Note that any participant whose local policy is used in the initial reconciliation phase is trivially compliant.

The following sections describe the format and use of the two types of clauses in Ismene; provisioning clauses and actions clauses.

3.2 Provisioning Clauses

Provisioning clauses are used to develop a policy instance from conditional statements. Each provisioning clause identifies zero or more *environmental* conditionals used to define when the consequences are applied to the instance. *Configuration*, *tag*, and *pick* consequences define how the instance is derived and defined.

Environment conditionals test the session environment. Each environment conditional is defined as a (possibly parameterized) predicate assessing a measurable aspect of the environment. However, the evaluation of environmental conditionals is outside the scope of the Ismene language. The environment in which Ismene is used is required to provide an interface for the evaluation of predicates. This approach separates the definition of relevant conditionals from the process of policy evaluation. Authorization and access control policy languages often defer condition evaluation (e.g., through upcalls in GAA API [33]).

Configuration consequences define how the requirements of a session are realized through configuration. Each such consequence identifies either a mechanism or mechanism configuration to be added to the policy instance. For example, consider the following clauses defining secrecy and integrity policies:

```
confidentiality : privateGroup() :: config(idhdlr(guar=conf));
integrity : RmteScope($mcaddr) :: config(idhdlr(intg=rfc2104,hash=md5));
```

The `confidentiality` clause states confidentiality should be provided when group is private (as indicated by the `privateGroup` predicate, but does not state how this is achieved (i.e., assumes the specification of a cryptographic algorithm occurs elsewhere). The second clause indicates that integrity is enforced through MD5-based HMACs [31, 23] if the session multicast address is remotely scoped (e.g., multicast traffic will extend beyond the local network). The application or infrastructure using Ismene is required to provide interfaces for evaluating the `privateGroup` and `RmteScope` conditionals.

Pick consequences afford the initiator flexibility in developing the session. Semantically, the pick statement indicates that exactly one configuration must be selected. Information in the local policies guide evaluation towards the most desirable configuration (see Section 3.4).

Tag consequences describe the organization of the group policy. The structure defined by the organization of tags defines the dependencies between sub-policies. Each tag consequence requires the evaluation of other clauses, which may lead to the introduction of further configurations and tags.

Consider the following policies appropriate for public and private sessions in a conferencing application:

```
provision : private($addr,$pt) :: config(idhdlr(guar=conf)),
                                  strong_key_mgmt, confidentiality;
provision :  :: config(idhdlr(guar=conf)), weak_key_mgmt, confidentiality;
strong_key_mgmt : :: config(lkh_rekeying()), secrecy;
secrecy : ManagerPresent($group) :: config(lkh_rekeying(sens=mem));
secrecy : :: config(lkh_rekeying(sens=leave));
weak_key_mgmt : Audio(), Video() :: config(kekkey(rekeyperiod=240));
weak_key_mgmt : Video() :: config(kekkey(rekeyperiod=120));
weak_key_mgmt : :: config(kekkey(rekeyperiod=60));
confidentiality : sensitive($subject) :: pick( config(idhdlr(encr=3des)),
                                               config(idhdlr(encr=desx)) );
confidentiality :  :: config(idhdlr(encr=des));
```

This policy states that private groups should be configured with strong key management and confidentiality, and non-private groups with weak key management and confidentiality. Initially, the conditionals associated with the first provision clause are evaluated. If the group is private (as determined by the address), then the `idhdlr` mechanism is configured to enforce confidentiality and the `strong_key_mgmt` and `confidentiality` tags are evaluated. If this fails, the evaluation falls to the next clause defining a policy for non-private groups by applying the `idhdlr` configuration and evaluating the `weak_key_mgmt` and `confidentiality` tags.

The evaluation of the strong key management requirement illustrates how a session provisioning can be responsive to membership. The (unconditional) strong key management clause states that a LKH mechanism should be used. However, in applying the *secrecy* tag consequence, the instance will arrive at a backward or membership rekeying policy, depending on a manager being expected to participate. Thus, groups with managers are afforded greater protection from non-members through a membership-sensitive policy that rekeys following any membership change.

The `weak_key_mgmt` clauses describe how the quality of service provided by key management can be determined by the types of data being transmitted (e.g, audio and video groups rekey least frequently, followed by video only, followed by other groups). Thus, through similar policies, configuration can be a reflection of the available resources or the demands made on surrounding infrastructure.

Pick consequences are useful in environments where the issuer wishes to set standards for operation, but does not wish to mandate an implementation. The first confidentiality clause states that, for groups with sensitive subjects, the `idhdlr` mechanism can (only) be configured to use either the 3DES [26] or DESX [22] algorithms to implement a strong confidentiality policy. If the subject is not sensitive, then the group will implement confidentiality by DES encryption.

3.3 Action Clauses

The action clauses defined in an instance identify the authorization and access control policy enforced by the group. Action clauses can contain configuration, credential, and environmental conditionals (i.e., tag consequences are not allowed), and are restricted to *accept* and *reconfig* consequences. An accept consequence indicates that an action should be allowed. The *reconfig* consequence represents the need for a reevaluation of the group and member provisioning policies.

Ismene represents a *closed world* in which denial is assumed. An action is allowed only if the evaluation of an associated action clause leads to an *accept* consequence. The tag of an action clause identifies the action to be considered (e.g., `join`, `send`). The set of protected actions are defined by the issuer, and assumed known *a priori* by the security mechanisms. Ismene is consulted for acceptance when any protected action is undertaken.

Configuration conditionals test the presence of configurations in an instance. A configuration conditional returns *TRUE* when the configuration is defined in the instance. The semantics of a pick conditional is the **or** of the configurations; the conditional returns *TRUE* if any one of the configurations described in the pick are contained in the instance.

Credential conditionals test the characteristics of authorization information associated with a protected action. A credential is modeled in Ismene as a set of attributes. For example, an X.509 certificate [18] can be modeled as attributes for subj.O (subject organization), issuer.CN (issuer canonical name), etc. To illustrate, consider the following action clause:

```
join : Credential(&cert,sgner=$ca,subj.CN=$joiner) : accept;
```

The first argument of a credential conditional (denoted with "&" symbol) represents binding. The credential test binds the matching credentials (see below) to the (&cert) attribute. This binding is scoped to the evaluation of a single clause. Conditionals are evaluated left to right.

The second and subsequent parameters of a credential conditional define a matching of credential attributes with attribute or constant values. The above example binds the credentials that were issued by a trusted CA (`sgner=$ca`) and have the subject name of the joining entity (`subj.CN=$joiner`) to the `&cert` attribute. The conditional returns true if a matching credential can be found. The assertion of valid and appropriate credentials is outside the scope of Ismene. Hence, it is up to the application is to validate and pass to Ismene the set of credentials associated with an action.

Consider the following set of action clauses:

```
join : config(idhdlr(encr=des)), In($JoinACL,$joiner),
       Credential(&cert,sgner=$ca,subj.CN=$joiner) : accept;
join : Credential(&cert,sgner=$ca,delegatejoin=true),
       Credential(&tocert,sgner=&cert.pk,subj.CN=$joiner) :: accept;
eject : sensitive($subject),
        Credential(&cert,sgner=$ca,role=X,subj.CN=$ejector) :: accept;
send : Credential(&key,key=$sessky), pick( config(idhdlr()),
                                           config(gdhdnlr()) ) :: accept;
```

The first `join` describes an ACL-based policy for admitting members to the group. The member is admitted to the group if she is identified in the `JoinACL` list attribute, she can provide an appropriate credential, and the session is encrypting traffic using DES.

The evaluation of action clauses is performed as with provisioning clauses. The second `join` is consulted only when the conditionals of first clause do not evaluate to *TRUE*. The second `join` clause describes a delegation policy. The first credential conditional binds `&cert` to the set of credentials delegating join acceptance (in this case, the set of certificates from the CA delegating join acceptance). The second conditional tests the presence of any credential signed with a delegated public key. Ismene is restricted to explicit delegation chains; each link in the chain must be explicitly stated as a credential conditional.

The `eject` clause describes basic role based authorization and access control. This clause states that the the `eject` action will be allowed only if a credential stating the requester's right to assume the role `X` can be found. If the credential is found, the requester, acting in role `X`, is allowed to eject another member. A similar clause can be defined for each action role `X` is authorized to perform.

Credentials can be used to test knowledge of session specific keys. For example, the `send` action clause describes the conditions under which an application message should be accepted. The clause states that the right to send a message in sessions configured with the `idhndlr` or `gdhdnlr` mechanism is predicated only on proof of the knowledge of current session key (matching `$sesskey`).

3.3.1 Reprovisioning the Group

The `reconfig` statement provides a means by which the group may advise the environment of a need to re-evaluate the session configuration. Consider the following action clauses that define a group policy requiring re-provisioning before members belonging to the `SpecialUsers` group are admitted:

```
prejoin : In($SpecialUsers, $joiner),
          Credential(&cert,sgner=$ca,subj.CN=$joiner) :: accept, reconfig;
join : In($SpecialUsers,$joiner), config(idhdlr(encr=des)),
       Credential(&cert,sgner=$ca,subj.CN=$joiner) :: accept;
provision : SpecialUsersPresent() :: config(idhdlr(encr=aes));
provision : :: config(idhdlr(encr=des));
```

To support re-provisioning, joining the group becomes a two phase process. Initially the member will perform a *prejoin*, after which the environment will be requested to re-provision the session. The `provision` tags specify that AES encryption should be configured if `SpecialUsers` are present and DES otherwise.

`reconfig` only causes notification; actual re-provisioning is outside the scope of Ismene. Re-provisioning may or may not be successful. The process of transitioning of a session to a new policy instance is a non-trivial problem that is largely unaddressed by current secure group communication systems. The prejoin step was introduced to handle the possibility that re-provisioning could fail or take substantial time. Only after the session is successfully re-provisioned to use AES, a member belonging to the `SpecialUsers` is admitted.

3.3.2 Integrating Ismene with External Authentication Frameworks

Ismene desires to take advantage of more expressive authentication frameworks (e.g., PolicyMaker [4], KeyNote [5], GAA API [33], Akenti [34]). In this way, Ismene need not replace existing approaches, but augment them. The following action clause describes how KeyNote can be used within an Ismene policy:

```
join: KeyNote($locCreds,$attrset) :: accept;
```

This clause states that a member should be admitted to the group only if KeyNote can generate a *proof of compliance* stating authorization to join the group. The conditional states that all credentials and the entire attribute set be passed to KeyNote (e.g., $locCreds, $workset). This is precisely the set of information used to evaluate a KeyNote policy.

3.4 Provisioning Policy Reconciliation

A group policy is reconciled with the local policies of the expected participants to arrive at a configuration. Thus, reconciliation determines which requirements are relevant to a session, and ultimately how the session is implemented. Described in Appendix B, policy reconciliation is the recursive assessment of clauses defined over the set of tags, conditionals, and consequences. The special *provision* tag is the start symbol for reconciliation of provisioning. Group and local policies are required to have at least one clause defined with the *provision* tag.

Reconciliation of provisioning policies takes the intersection of the group and local policies. However, in Ismene, the group policy is authoritative; all configurations and pick statements used to develop the instance must be explicitly stated in the group policy. Local policies are consulted only where flexibility is expressly granted by the issuer through pick statements.

Reconciliation begins by testing the conditionals associated with the first *provision* clause in the group policy. If all conditionals evaluate to true, the clause consequences are applied to the instance. If not, then the next clause associated with the tag is evaluated. If no clause evaluates to true, then the policy cannot be resolved and the session cannot be implemented.

Applied configuration consequences are added to the instance. Configurations are ordered by their introduction into the instance. Tag consequences indicate the need for further reconciliation. All applied tag consequences are added to the ordered set of tags that must be evaluated. Clauses associated with these tags are evaluated as described above. Reconciliation terminates when the set of tags to be evaluated is empty.

The local policy of each expected group member is evaluated prior to the reconciliation of the group policy. The local policies are evaluated starting with the *provision* tag as defined above. However, all pick statements are left unresolved. The result of each evaluation is an local policy instance defining the local requirements of the member.

The local policy of an expected participant guides the resolution of *pick* statements to the most desirable configuration. To simplify, if a configuration in the pick is in the local policy, it is selected. If the local policy provides no such guidance, the pick is left unresolved and the next local policy is consulted.

Conflicts may arise when consulting multiple local policies. For example, consider a group policy pick statement defining configurations for A and B. A conflict occurs when some local policies require A and others require B. The resolution of the pick statement will determine who can participate in the session.

One resolution algorithm uses a majority policy; whichever configuration gets the most votes (required by largest number of local policies) wins. However, this approach has the undesirable property that may leave a large number (or all) local policies with unaddressed requirements. Thus, potentially, all members may be prohibited from participation.

A second algorithm establishes an ordering between local policies. A more important member's local policy would be considered first, and others when they provide no guidance. Our current implementation uses

this solution. Note that there is a reduction from MIN COVER (NP-complete) [12] to the problem of finding an optimal solution (where the maximum number of local policies are satisfied).

The following group and local policies illustrate reconciliation (local policy l_1 is ordered before l_2):

group policy	`provision: :: config(A), pick( config(B), config(C) ), X;` `X: :: pick( config(D), config(E) );`
local policy l_1	`provision: :: config(A), config(B);`
local policy l_2	`provision: :: config(B), config(D);`

Initially, the (unconditional) group policy would reconcile to the configuration A and picks for (B or C) and (D or E). l_1 would be consulted first. $config(B)$ would be selected from the first pick statement because l_1 requires it. l_1 provides no guidance for the second pick statement. In this case, l_2 would be consulted and D selected. Thus, the resulting instance would contain A, B, and D. Note that the introduction of other local policies or requirements may lead to an unreconciliable local policy. For example, if l_1 also required another configuration `config(E)`, the algorithm would arrive at the instance A, B,and E. In this case, the requirement for D in l_2 cannot be satisfied, and the member associated l_2 should choose not to participate.

Each policy represents a graph whose nodes are clauses and edges are tag consequences. We restrict the organization of tags by requiring that this graph be acyclic. Logically, this prevents recursively defined requirements. This ensures that no reconciliation of policy can lead to the introduction of a previously evaluated clause. Because no clause is visited more than once and assuming conditionals are evaluated in $O(1)$, reconciliation is P-time computable in the number of clauses.

3.5 Authorization Policy Reconciliation

Authorization and access control policies are enforced through the reconciliation of the action clauses defined by an instance. An instance defines the action clauses to be enforced by the group through *authorization Policy Reconciliation*.

The set of clauses determining authorization and access control in an instance is defined by the intersection of the group and local policies. For example, consider group policy that defines the action clauses ($t_i : c_1 :: accept$;) and (t_i: c_2 :: accept;). Further, a local policy defines an action clause as (t_i: c_3 :: accept;), and another local policy defines the action clause (t_i: c_4 :: accept;). Authorization reconciliation algorithm takes the logical **and** of these policies; the action clause is for $t1$ is defined as

$$t1 : ((c_1 \vee c_2) \wedge c_3 \wedge c_4) :: accept$$

Thus, the action clauses defined by an instance are the conjunction (one for each group and local policy) of disjunctions (one for each action clause defined for t_i in a policy). Authorization reconciliation constructs an action clause for each action $t_i \in T$ from the action clauses in the group and local policies. `reconfig` consequences listed in any of the considered policies are added to the action clause in the instance.

3.6 Compliance

Not all participants local policies are required to be consulted during reconciliation. There, a participant must be able to check the compliance of an instance with its local policy prior to participating in a session. Compliance is successful if all requirements stated in the local policy are satisfied by the instance. There are two phases of compliance; provisioning and authorization.

Provisioning compliance uses the instance resulting from the evaluation of the local policy starting from the *provision* tag. Each configuration and pick statement must be satisfied by the instance. A configuration is satisfied if it is explicitly stated in the instance. A pick statement is satisfied if at least one configuration from the list is contained in the instance. Thus, the execution time of the provisioning compliance grows linearly with the number of consequences.

Participants may wish to place requirements on authorization and access control. An instance should never be more permissive than the local policy. In general, checking compliance of authorization and access control policies is difficult. Gong and Qian found that the closely related problem of determining interoperability between authorization policies is NP complete [14].

Logically, Ismene authorization compliance assesses whether the instance implies the local policy. Given an expression e_1 describing authorizations in an instance, and a similar expression describing a local policy e_2, it is easy to check compliance between the policies by testing whether the expression $e_1 \Rightarrow e_2$ is a tautology. To illustrate, consider the action clauses defined in the following instance and local policies:

policy instance	$X : (c_1 \wedge c_2) \vee c_3 :: accept;$
local policy A	$X : c_1 :: accept;$ $X : c_3 :: accept;$
local policy B	$X : c_1, c_3 :: accept;$

The group policy will be compliant with the local policy A because the policy is less permissive (e.g., $(c_1 \wedge c_2) \vee c_3 \Rightarrow c_1 \vee c_3$). The group policy is not compliant with local policy B because the group policy is more permissive (e.g., $(c_1 \wedge c_2) \vee c_3 \not\Rightarrow c_1 \wedge c_3$). Tautology testing such expressions is NP-complete [12]. However, the local policies we have encountered are generally restricted to a few clauses with a small number of conditionals. Thus, compliance can be can be executed using reasonable resources.

Authorization compliance is the process of determining, for each action $t_i \in T$, that the conditionals of the instance action clause for t_i are sufficient to satisfy at least one action clause in the local policy. Failure to satisfy an action clause represents a more permissive policy; an action that would be allowed by the instance would be denied by the local policy.

3.7 Analysis

It is important to restrict instances to legal configurations. Thus, Ismene must be able to describe the acceptable usage and configuration of the security mechanisms. To this end, we introduce *assertions* into the policy language. Semantically, an assertion indicates that a configuration must or must never be true in any instance. Assertions are independent clauses (i.e, the clause does not contain tag consequences). Positive (negative) assertions must (not) be satisfied by any policy instance.

A number of systems have investigated techniques guaranteeing correct and efficient construction of software from components [17, 24]. These approaches typically describe relations defining compatibility and dependence between components. A configuration is deemed correct if it does not violate these relations. For example, Hiltunen [17] defines the conflict, dependency, containment, and independence relations. The following describes assertions representing these relations (independence is assumed):

conflict (A *is incompatible with* B)	`assert : :: ! config(A()),config(B());`
dependency (A *depends on* B)	`assert : config(A()) :: config(B());`
inclusion (A *provides* B)	`assert : config(A()) :: ! config(B());`

In our current implementation, the *analysis* algorithm attempts to determine if **any** instance resulting from a group policy can violate a set of assertions. In the worst case, this requires the generation of all possible instances (there may an exponential number of them). We found that most reasonable configurations exhibit a degree of *independence*; the introduction of a configuration is largely the result of the reconciliation of a few clauses. Hence, the evaluation of an assertion can be reduced to the analysis of only those clauses upon which the configurations stated in the assertions are dependent.

Assertions can be used in the group policy to state issuer requirements. For example, the issuer may wish to assert a completeness property [20, 7] that any instance resulting from reconciliation enforces confidentiality over the application data. Thus, knowing in advance that the *Ismene*, *generic*, and *xor* data handler mechanisms configured with confidentiality are the only available means by which this property can be provided, the issuer states the following completeness assertion:

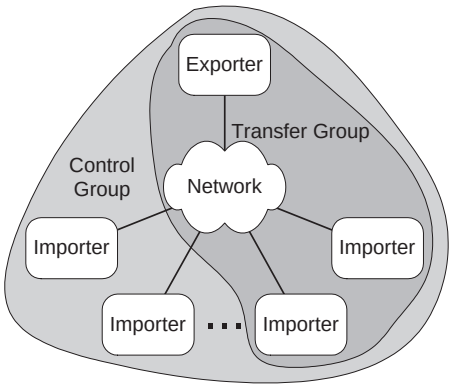

Figure 2: The `imird` filesystem mirroring application - Update announcements reporting changes to the mirrored filesystem are distributed by an exporter authority over the long-lived control group. File updates are transferred to interested importers over short-lived transfer groups.

```
assert :  :: pick( config(idhdlr(guar=conf)), config(gendhlr(guar=conf)),
                   config(xordhlr(guar=conf)) );
```

Analysis rejects any policy failing to preserve this requirement.

4 Ismene Implementation and Example Usage

The Ismene specification language has been fully implemented in the *Ismene Applications Programming Interface* (API). The API defines interfaces for the creation, parsing, reconciliation, and analysis of Ismene policies. A number of tools used to support the development of Ismene policies have been constructed. The Ismene policy compiler, `ipdlcc`, validates the syntax of a group policy and performs analysis.

We have extended a secure group communication framework [1] to construct secure groups through Ismene policies. The implementation consists of approximately 30,000 lines of C++ source and has been used as the basis for several non-trivial group applications. All source code and documentation for the Ismene language, the secure group communication framework, and applications are freely available from `http://antigone.eecs.umich.edu/`.

We now illustrate an actual use of Ismene to specify policies for a secure filesystem mirroring application, `imird`. The `imird` application, based on specified policies, is currently used to distribute the Ismene and group communication framework source code from our main repository to the users' and developers' machines in our experimental lab.

An `imird` filesystem is a directory tree whose contents are distributed by an *exporter*. The exporter is a distinct member of the group who distributes copies of the files within the exported directory tree. The *importers* are group members who maintain a synchronized copy of the `imird` filesystem on their local disk. The exporter acts as an authority for the files and directories in the filesystem. The operation of `imird` is depicted in Figure 2 and summarized in the following:

1. The exporter periodically transmits *announcements* to a group, called the *control group*, consisting of all potential importers. The announcement does **not** contain the actual files. Instead, it contains the names, MD5 content hashes, modification dates, and sizes of files in the filesystem. Importers receiving this information compare these contents with their copy of the imported filesystem. Where differences require update (e.g., a local file is not consistent with the announcement), a download request is sent to the exporter.

```
 1 % File : (imird.ipdl) Group Policy for the imird Mirroring Application
 2 % Attributes Section
 3 group := < imird Policy >;
 4 issr:= < iQBVAw ... >;
 5
 6 % Provisioning Section
 7 provision: :: authentication, membership, grouptypepol;
 8 grouptypepol: isControlGroup() :: memkey, weakconf;
 9 grouptypepol: :: timekey, strongdat;
10 authentication: :: config(OpenSSL());
11 membership: :: config(IMember(retry=3,rexmit=5));
12 memkey: :: config(lkhkey(sens=memsens));
13 timekey: :: config(kekkey(rekeyperiod=300));
14
15 % Data Security Policies
16 weakconf: :: config(idhdlr(guar=conf)),
17              pick( config(idhdlr(conf=des-cbc)), config(idhdlr(conf=rc2)) );
18 strongdat: :: config(idhdlr()),confsauth;
19 confsauth: isSensitive($file) :: config(idhdlr(guar=conf,conf=3des)),
20            config(idhdlr(guar=intg,intg=md5)), config(idhdlr(guar=sauth,sauth=ssig));
21 confsauth: :: config(idhdlr(guar=conf,conf=desx)), config(idhdlr(guar=intg,intg=md5)),
22               config(idhdlr(guar=sauth,sauth=ssig));
23
24 % Authorization/Access Control Policies
25 init: isControlGroup(),Credential(&cert,iss=$issr,subj.CN=$joiner) :: accept;
26 init: Credential(&cert,iss=$issr,fs=$fsys,subj.CN=$joiner) :: accept;
27 join: isControlGroup(),Credential(&cert,iss=$issr,subj.CN=$joiner) :: accept;
28 join: Credential(&cert,iss=$issr,fs=$fsys,subj.CN=$joiner),groupSmaller(100) :: accept;
29 rekey: isControlGroup(),Credential(&key,key=$kekkey) :: accept;
30 rekey: Credential(&key,key=$lkhKey) :: accept;
31 send: Credential(&key,key=$sessKey) :: accept;
32 sendauth: Credential(&cert,iss=$issr,subj.CN=$sender) :: accept;
33 leave: :: accept;
34
35 % Policy Verification
36 signature := < sdD5aR ... >;
```

Figure 3: An `imird` Group Policy

2. The exporter collects the download requests and creates a new group, called a *transfer group*, for each file that is to be updated. An announcement for each transfer group is broadcast to the control group prior to the initialization of the transfer group. This announcement contains the transfer group address (multicast address and port) and identifies the file to be transferred.
3. The importers who require the specified file join the associated transfer group. After a certain interval, the exporter reliably and securely multicasts the updated file to the transfer group. The importers leave the group at the completion of the transfer, and the group is terminated.

Security policies for the above application, in general, will differ, depending on factors such as the sensitivity of files and announcements, importers, and available crypto mechanisms. In Figure 3, we list an example policy we have used with `imird`. For brevity, we omit the legal usage assertions associated with this policy. This example policy has the following general provisioning and access control requirements.

Provisioning requirements in the example policy

- Announcements, sent to the control group, are considered less sensitive than the files broadcast to the

```
 1 % Description : (exploc.ipdl) Exporter Local Policy
 2 issr:= < iQBVAw ... >;
 3
 4 % Requirements
 5 provision: :: authentication, data_security;
 6 authentication: :: config(OpenSSL());
 7 data_security: isControlGroup() :: config(idhdlr(guar=conf));
 8 data_security: :: config(idhdlr(guar=conf,guar=sauth));
 9
10 % Authorization and access Control
11 export: Credential(&cert,iss=$issr,fs=$fsys,ex=true,subj.CN=$exporter) :: accept;
12 policy: Credential(&cert,pk=$issr,iss=$issr) :: accept;
13 init: Credential(&cert,iss=$issr,subj.CN=$joiner) :: accept;
14 sendauth: Credential(&cert,iss=$issr,subj.CN=$sender) :: accept;
15 % No local policy reguarding the actions below
16 join: :: accept; rekey: :: accept; send: :: accept; leave: :: accept;
```

Figure 4: An `imird` Local Policy

transfer groups. Thus, the control group requires that application traffic only have weak confidentiality (line 16 in Figure 3). Group content should be protected from past and future members (line 12).

- The actual data files, sent only to the transfer group, are generally more sensitive than announcements. Thus, they require stronger confidentiality as well as data integrity (lines 18-22). To prevent compromised importers from asserting forged files, sender authenticity is required (lines 20 and 22). In addition, as stated on line 13, the `timekey` clause is used to specify that a Key-Encrypting-Key [16] key distribution mechanism rekey the session every 300 seconds.

The provisioning clauses in the group policy also specify further details about the above requirements. The `pick` statement on line 17, for example, states that the weak confidentiality requirement can be met either by using DES-CBC [26] or RC2 [32] as the encryption algorithm.

The `strongdat` clause (line 18) states the requirement for strong guarantees over the group content. The `isSensitive($file)` conditional on line 19 tests the sensitivity of the file to be transferred (where $file is an application supplied attribute). If the file is deemed sensitive, triple DES must be used (line 19). Otherwise, DESX must be used (line 21). To ensure integrity, MD5 [31] must be used, and to provide sender authenticity, stream signatures [13] must be used (lines 20 and 22).

The compliance of the instance is assessed against the local policies. The exporter's local policy described in Figure 4 states minimal requirements for data security. In this example, these requirements are trivially satisfied by any instance resulting from the reconciliation of the group policy. The importers' local policies (not shown) are subject to the same compliance checks.

Access control requirements in the example policy

The group policy places authorization and access control restrictions on the control and transfer groups via action clauses (lines 25-33). Authorization is stated in `imird` through permission certificates. The exporter is issued a certificate identifying that it is the authority for a particular `imird` filesystem. Prior to joining any group, each participant is issued a certificate for each filesystem that they are allowed to import. The public key of the filesystem authority is stated explicitly through the `issr` attribute (line 4 in the group policy and line 2 in the local policy). The primary access control requirements stated in the group policy are as follows:

- *Initial authentication:* Authentication must occur before a potential member can acquire the policy instance (through the `init` action, lines 25 and 26). A potential participant is required to present a permission certificate to authenticate themselves to the control group. In addition, a certificate for the

parent filesystem of the file being transferred is also required in order to initiate participation in a transfer group.

- *Join actions:* The same criteria used for initialization is used for gaining access to the group, with the exception that a group member is not allowed to join transfer groups containing 100 or more participants. The `join` clause containing the `groupSmaller()` conditional (28) enforces a ceiling on the size of transfer groups.
- *Other actions:* Subsequent acceptance of action within the group is predicated on the presence of the appropriate keys. For example, the send clause (31) states that any group message must have, at a minimum, a session key credential. Implicitly, any malformed message is ignored (e.g., if a configured integrity property is not preserved).

The local policy of importers also places restrictions on the session. In particular, each importer must determine that the exporter is authentic and the group policy was issued by an authorized issuer. This is represented by the `export` (11) and `policy` (12) clauses in the local policy; the exporter must present a certificate stating that have authority over the filesystem and that the group policy was signed by an issuer identified in the local policy, respectively.

5 Related Work

Several recent group communication systems, including DCCM [10], GSAKMP [15], and Antigone [25], support the notion of a group security policy that defines the configuration of the security aspects of the system. In all these systems, generally, the group security policy is assumed to be *static*. In that sense, the policy instance generated from Ismene can be considered as the policy input to these group communication systems. Ismene extends these systems by stating the conditions under which certain policies should be enforced. In addition, Ismene expresses policies that involve aspects of both provisioning and access control (support for the latter is limited in the above systems).

The problem of reconciling multiple policies in an automated manner is only beginning to be addressed. In the two-party case, the emerging Security Policy System (SPS) [39] defines a framework for the specification and reconciliation of local security policies for the IPSec protocol suite [21]. To handle a similar situation in Ismene, two local policies for the two ends of the IPSEC connection can be specified. These policies will be resolved against a group policy that leaves the choice of mechanisms open via pick statements.

In the multi-party case, DCCM system [10] provides a negotiation protocol for provisioning. The first phase of the protocol involves the initiator sending a policy proposal to each potential member and receiving counter proposals. Subsequently, the initiator declares the final policy that potential members can accept or reject, but not modify. Policy proposals define an acceptable configuration (which, for particular aspects of a policy, can contain wildcard "don't care" configurations). An advantage of this protocol is that the local policy need not be revealed to the initiator. Ismene, if desired, can be easily adapted to use the DCCM's negotiation protocol. Ismene is more expressive because it can be used to state conditions under which various configurations can be used and when configurations need to be reconsidered in response to actions. The authorization and access control model is also more general in Ismene.

Language-based approaches for specifying authorization and access control have long been studied [37, 4, 8, 38, 5, 33], but they generally lack support for provisioning. Because of the vast earlier work in this area and to simplify the language design, Ismene does not attempt to be as expressive for stating complex access control rules. Instead, as identified in Section 3.3.2, Ismene is designed to leverage the expressive power of other access control systems via external authorization services.

The PolicyMaker [4] and KeyNote [5] systems provide a powerful and easy to use framework for the evaluation of credentials. Generally, support for provisioning and resolving multiple policies is not the focus of

these systems. When desired, these systems can be invoked in Ismene conditionals to leverage their expressive power and extend their use to group communication systems.

In [19], KeyNote has been used to define a distributed firewall application. The technique is to use conditional authorizations, where conditions involve checking port numbers, protocols, etc. However, it still remains problematic to construct a configuration, based on multiple local policies, or for determining the correctness of a configuration. The provisioning clauses and legal usage assertions of Ismene can help address these problems.

6 Conclusions

We have presented a language-based approach for the *provisioning* and *access control* of secure groups through the specification of security policies. Central to Ismene is the specification of conditional security requirements through *group* and *local policies*. The group and local policies are reconciled towards a *policy instance*. The definition of provisioning and access control for a session is precisely stated in the policy instance. Policies can be used to express the relationships and interactions between provisioning and access control.

We have constructed prototype support libraries and tools used for the creation and use of Ismene policies. Using the tools, group participants can determine the compliance of a policy instance with their local policies. A group policy can be analyzed for completeness and consistency against legal usage assertions.

Ismene has been successfully integrated with a secure group communication framework [1]. A number of applications have been built and policies tested. The use of Ismene to define a filesystem mirroring application was illustrated.

References

[1] Anonymous. *Masked for anonymity*.

[2] Philip A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, February 1996.

[3] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, 16(4):321–366, 1998.

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

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

[6] David C. Blight and Takeo Hamada. Policy-Based Networking Architecture for QoS Interworking in IP Management. In *Proceedings of Integrated network management VI, Distributed Management for the Networked Millennium*, pages 811–826. IEEE, 1999.

[7] D. Branstad and D. Balenson. Policy-Based Cryptographic Key Management: Experience with the KRP Project. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX '00)*, pages 103–114. DARPA, January 2000. Hilton Head, S.C.

[8] L. Cholvy and F. Cuppens. Analyzing Consistancy of Security Policies. In *1997 IEEE Symposium on Security and Privacy*, pages 103–112. IEEE, May 1997. Oakland, CA.

[9] Joaen Daemen and Vincent Rijmen. AES Proposal: Rijndael. AES submission, June 1998. `http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.prf`.

[10] 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 DCCM Project. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX '00)*, pages 64–73. DARPA, January 2000. Hilton Head, S.C.

[11] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 83–92. ACM, 1998.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractibility, A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, NY, first edition, 1979.

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

[14] L. Gong and X. Qian. The Complexity and Composability of Secure Interoperation. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 190–200, Oakland, California, May 1994. IEEE.

[15] H. Harney, A Colegrove, E. Harder, U. Meth, and R. Fleischer. Group Secure Association Key Management Protocol (*Draft*). *Internet Engineering Task Force*, June 2000. `draft-harney-sparta-gsakmp-sec-02.txt`.

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

[17] Matti Hiltunen. Configuration Management for Highly-Customizable Software. *IEE Proceedings: Software*, 145(5):180–188, 1998.

[18] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. *Internet Engineering Task Force*, January 1999. RFC 1949.

[19] Sotiris Ioannidis, Angelos D. Keromytis, Steve Bellovin, and Jonathan M. Smith. Implementing a Distributed Firewall. In *Proceedings of Computer and Communications Security (CCS) 2000*, pages 190–199, 2000. Athens, Greece.

[20] Sushil Jajodia, P. Samarati, and V. Subrahmanian. A Logical Language for Expressing Authorizations. In *In Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, March 1997. IEEE.

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

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

[23] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. *Internet Engineering Task Force*, April 1997. RFC 2104.

[24] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building Reliable High-Performance Communication Systems from Components. In *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33, pages 80–92. ACM, 1999.

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

[26] National Bureau of Standards. DES Modes of Operation, (FIPS PUB) 81. *Federal Information Processing Standards Publication*, December 1980.

[27] P. Nikander and Arto Karila. A Java Beans Component Architecture for Cryptographic Protocols. In *Proceedings of 7th USENIX UNIX Security Symposium*, pages 107–121. USENIX Association, January 1998. San Antonio, Texas.

[28] H. Orman, S. O'Malley, R. Schroeppel, and D. Schwartz. Paving the Road to Network Security or the Value of Small Cobblestones. In *Proceedings of the 1994 Internet Society Symposium on Network and Distributed System Security*, February 1994.

[29] Adrian Perrig, Dawn Song, Doug Tygar, and Ran Canetti. Efficient Authentication and Signature of Multicast Streams over Lossy Channels. In *2000 IEEE Symposium on Security and Privacy*, pages 56–70. IEEE, May 2000. Oakland, CA.

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

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

[32] R. Rivest. Description of the RC2(r) Encryption Algorithm. *Internet Engineering Task Force*, January 1998. RFC 2268.

[33] T. Ryutov and C. Neuman. Representation and Evaluation of Security Policies for Distributed System Services. In *Proceedings of DARPA Information Survuvability Conference and Exposition*, pages 172–183, Hilton Head, South Carolina, January 2000. DARPA.

[34] M Thompson, W. Johnson, S. Mudumbai, G. Hoo, K. Jackson, and A. Essiari. Certificate-based Access Control for Widely Distributed Resources. In *Proceedings of 8th USENIX UNIX Security Symposium*, pages 215–227. USENIX Association, August 1999. Washington D. C.

[35] A. Westerinen, J. Schnizlein, J. Strassner, Mark Scherling, Bob Quinn, Jay Perry, Shai Herzog, An-Ni Huynh, and Mark Carlson. Policy Terminology (*Draft*). *Internet Engineering Task Force*, July 2000. `draft-ietf-policy-terminology-00.txt`.

[36] 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.

[37] T. Woo and S. Lam. Authorization in Distributed Systems; A New Approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.

[38] T. Woo and S. Lam. Designing a Distributed Authorization Service. In *Proceedings INFOCOM '98*, San Francisco, March 1998. IEEE.

[39] J. Zao, L. Sanchez, M. Condell, C. Lynn, M. Fredette, P. Helinek, P. Krishnan, A. Jackson, D. Mankins, M. Shepard, and S. Kent. Domain Based Internet Security Policy Management. In *Proceedings of DARPA Information Survuvability Conference and Exposition*, pages 41–53, Hilton Head, South Carolina, January 2000. DARPA.

Appendix A - Ismene Grammar

The following describes the Ismene grammar. A $word$ represents a string of non-whitespace alphanumeric characters. A $string$ is a string of alphanumeric characters (i.e., may contain newline and whitespace characters).

```
<policy> := <statements>
<statements> := <statement> ";" ["," <statements>]
<statement> := <attribute> | <prov_clause> | <action_clause> | <assertion>

<attribute> := <identifier> ":=" "<" <value> ">"  |
               <identifier> ":=" "<" <value list> ">"
<value list> := "{" <value> "}" ["," <value list>]
<identifier> := word
<value> := string

<prov_clause> := <tag> ":" [<conditionals>] "::" <consequences>
<tag> := <identifier>
<conditionals> := <var> "=" <value> ["," <conditionals>] |
                  <predicate> "(" [<args>] ")" ["," <conditionals>]
<var> := "$" <identifier>
<predicate> := <identifier>
<args> := <var> | <identifier>  ["," <args>]
<consequences> := <pick> | <config> |  <tag> ["," <consequences>]
<pick> := "pick" "(" <config> "," <config> ["," <configs>] ")"
<configs> := <config> ["," <configs>]
<config> := "config" "(" [ <cfgstmts> ] ")"
<cfgstmts> := <mechanism> [ "(" <params> ")" ] [, <cfgstmts>]
<mechanism> := <identifier>
<params> := <identifier> "=" <value> ["," <params> ]

<action_clause> := <tag> ":" [<action_conditionals>]
                           "::" "accept" [, "reconfig"]
<action_conditionals> := <action_condition> ["," <action_conditionals>]
<action_condition> := <conditionals> | <credential>  | <config> | <pick>
<credential> :=  Credential "(" <bind var> "," <cred_args> ")"
<bind var> := "&" <identifier>
<cred_args> := <identifier> "=" <value> ["," <cred_args>] |
               <identifier> "=" <var> ["," <cred_args>] |
               <identifier> "=" <bind var> ["," <cred_args>]

<assertion> := "assert" ":" [<assert_conds>] "::" <configs>;
<assert_conds> := <condition> | <config>  ["," <assert_conds>]
```

Appendix B - The Ismene Reconciliation Algorithm

The following describes the reconciliation algorithm used to develop a group configuration (provisioning) from the group (α) and local policies (R). The $pick(d, R)$ function resolves pick consequence d towards desirable configurations using local policies $\in R$ (see Section 3.4 for details).

Function *reconcile*

inputs

s	start symbol (e.g, *provision*)
α	Group Policy
R	local policies

outputs

$\hat{\alpha}$	policy instance (ordered set of configurations)

variables

ω	FIFO queue of tags ($\omega.head$ returns first element)
c	clause ($c.tag$, $c.conds$, $c.consq$ denotes the tag, conditionals, and consequences of c, respectively)

$\textbf{function}\ reconcile(s, \alpha, R)$
{
  $\omega = s, \hat{\alpha} = \emptyset$
  $while\ (\omega! = \emptyset)$
  {
    $c = c_i \mid c_i.tag = \omega.head,\ c_i, c_j \in \alpha,\ (j \geq i) \Rightarrow (j = i)$
    if ($\forall d \in c.conds, d = TRUE$)
    {
      $\forall q \in c.consq$ **do**
      {
        if q **is** $configuration$ $\quad \hat{\alpha} = \hat{\alpha} + q$
        else if q **is** $pick$ $\quad \hat{\alpha} = \hat{\alpha} + pick(q, R)$
        else if q **is** tag $\quad \omega = \omega \cup q$
      }
    }
    $\alpha = \alpha - c$
  }
  $return\ \hat{\alpha}$
}