Methods and Limitations of Security Policy Reconciliation

Patrick McDaniel
SIIS Laboratory, Pennsylvania State University
and
Atul Prakash
University of Michigan

Abstract

A security policy specifies session participant requirements. However, existing frameworks provide limited facilities for the automated reconciliation of participant policies. This paper considers the limits and methods of reconciliation in a general-purpose policy model. We identify an algorithm for efficient two-policy reconciliation, and show that, in the worst-case, reconciliation of three or more policies is intractable. Further, we suggest efficient heuristics for the detection and resolution of intractable reconciliation. Based upon the policy model, we describe the design and implementation of the Ismene policy language. The expressiveness of Ismene, and indirectly of our model, is demonstrated through the representation and exposition of policies supported by existing policy languages. We conclude with brief notes on the integration and enforcement of Ismene policy within the Antigone communication system.

1. INTRODUCTION

Policy is frequently the means by which the requirements of communication participants are identified and addressed. Session policies are stated by the different participants and organizations for the services supporting the communication. At present, facilities for the reconciliation of participant policies in existing policy

This work is supported in part 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. This work is also supported in part by the National Science Foundation under grant number 088285. This work was also supported in part by the National Science Foundation under grant CCR-0082851.

frameworks are limited in scope and semantics. Hence, policies must be reconciled manually, a frequently complex process. Where no provision for reconciliation is made, governing authorities must dictate policy. In that case, session participants accepting dictated policy have limited ability to affect how session security is defined.

Automated reconciliation is a means by which the possibly divergent requirements of session participants can be met. Participants specify their requirements through policy. These policies are reconciled at run-time, resulting in an unambiguous session-defining specification called a *policy instance*. In this case, session security is the result of all requirements, rather than dictated by a single authority.

A session security policy defines security-relevant properties, parameters, and facilities used to support a session. Thus, a session policy states how security directs behavior, the entities allowed to participate, and the mechanisms used to achieve security objectives. This broad definition extends much of existing policy; dependencies between authorization, data protection, key management, and other facets of a communication can be represented within a unifying policy. Moreover, requirements frequently differ from session to session, depending on the nature of the session and the environment in which it is conducted. Hence, the conditional requirements of all parties should be encompassed by policy.

This paper considers the definition, efficiency, and methodologies of security policy reconciliation within a general-purpose policy model. This model defines policy as the collection of interdependent statements of provisioning and authorization. Each statement identifies context-sensitive session requirements. A reconciliation algorithm attempts to identify a policy instance compliant with the stated requirements. We define and prove the correctness of an efficient two-policy reconciliation algorithm, and show by reduction that three or more policy reconciliation is intractable. We identify several heuristics for detecting and combating intractable provisioning policy reconciliation, and show that reconciliation of (many) reasonable authorization policies can be efficient.

We further consider the related problems of policy compliance and analysis. A compliance algorithm determines whether an instance is consistent with the requirements stated in a policy. The analysis algorithm determines whether the provisioning of a session adheres to a set of assertions that express correctness constraints on a policy instance. We identify efficient algorithms for compliance and analysis, and demonstrate that a more general form of analysis is intractable (coNP).

The Ismene policy language and supporting infrastructure is built upon the model and algorithms defined throughout. The expressiveness of Ismene, and indirectly the applicability of our policy model, is demonstrated through the representation and exposition of policies defined in several popular policy languages. We conclude by describing our experiences with the integration and enforcement of Ismene policy within the Antigone communication system.

Policy has been used in different contexts as a vehicle for representing authorization [Woo and Lam 1993; Blaze et al. 1996; Cholvy and Cuppens 1997; Woo and Lam 1998; Ryutov and Neuman 2000], peer session security [Zao et al. 2000], quality of service guarantees [Blight and Hamada 1999], and network configuration [Bellovin 1999; Bartal et al. 1999]. These approaches define a policy language

or schema appropriate for their target problem domain. This paper expands on this work by defining a general approach in which policy is used to both provision and to regulate access to communication services.

The problem of reconciling policies in an automated manner is only beginning to be addressed. In the two-party case, the emerging Security Policy System (SPS) [Zao et al. 2000] defines a framework for the specification and reconciliation of security policies for the IPsec protocol suite [Kent and Atkinson 1998]. Reconciliation is largely limited to intersection of specified data structures. In the multi-party case, the DCCM system [Dinsmore et al. 2000] provides a negotiation protocol for provisioning. DCCM defines the session policy from the intersection of policy proposals presented by each potential member. Each proposal defines a range of acceptable values along a multi-dimensional policy structure. Hence, reconciliation in these systems is largely based on the intersection of policy schemata. In contrast, this work attempts to define a general framework upon which more flexible expression-oriented policies are defined and reconciled.

Language-based approaches for specifying authorization and access control have long been studied [Woo and Lam 1993; Cholvy and Cuppens 1997; Woo and Lam 1998; Ryutov and Neuman 2000], but they generally lack support for reconciliation. These systems typically identify rigorous semantics for the evaluation of authorization statements. The PolicyMaker [Blaze et al. 1996] and KeyNote [Blaze et al. 1999] trust management systems provide a powerful framework for the evaluation of credentials. Trust management approaches focus on the establishment of chains of conditional delegation defined in authenticated policy assertions. Hence, policy is dictated by entities to which session authority is delegated, rather than through the reconciliation of participant requirements.

The following section considers the requirements of a general-purpose policy language. Section 3 considers the limits and methods of reconciliation in our general policy model. Section 4 presents the Ismene language. Section 5 explores algorithms for policy analysis. Section 6 illustrates the use of Ismene by representing policies supported by existing languages. Section 7 briefly discusses our experiences with the implementation and use of Ismene. Section 8 concludes.

2. REQUIREMENTS

To illustrate the policy reconciliation needs, we present very simplified security requirements for an example conferencing 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 $3DES$ or AES (provisioning requirement)
—sessions are restricted to *widget.com* employees (authorization requirement)

Now suppose *Alice* wishes to sponsor a session of `tc` under the following policy:

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

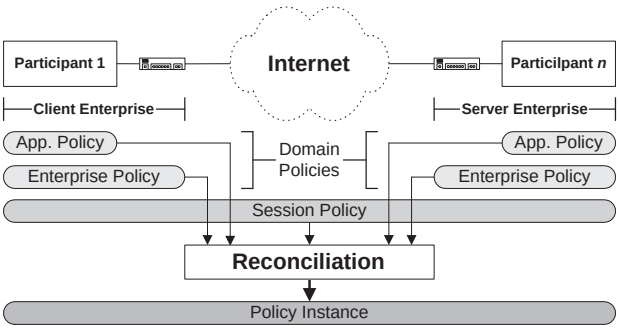

Fig. 1. Policy construction - A session-specific policy instance for two or more participants is created by an initiator. Each participant submits a set of domain policies identifying the requirements relevant to the session. The initiator constructs the policy instance compliant with each domain and the session policy through reconciliation.

A basic requirement on a policy approach for this scenario is that it must reconcile the provisioning and authorization requirements (policies) stated by any number of interested parties. It is through this process of reconciliation that a concrete, enforceable policy is developed. In the above example, Alice's and the widget.com policies are reconciled to arrive at a policy that restricts the participants to members of *widget.com*'s *BlueWidgets* team (authorization requirement), and `tc` 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 periodically, etc. (environment dependence). In some cases, the session must be able to make access control decisions based on the use and configuration of security mechanisms; for example, admit a member only if AES is being used for ensuring confidentiality. The policy model and associated language described in the following sections permits such dependencies between authorization and provisioning policy. This represents a divergence from many existing works that treat authorization and provisioning independently.

3. POLICY

This section presents the Ismene approach to policy management. Depicted in Figure 1, a session is established between two or more entities. Each participant in the session submits a set of relevant domain policies to the *initiator*. The initiator may be a participant or external entity (e.g., policy decision point [Durham et al. 2000]). Stated by a *policy issuer*, a session policy is a template describing a legal session provisioning and the set of rules used to govern access.

Domain policies state conditional requirements and restrictions placed on the session. In the scenario described in the previous section, Alice's domain policy states that AES must be used and the session be restricted to members of the BlueWidget team. The policies appropriate for a particular session are dependent on the environment in which it is to occur. The scenario described in Figure 1 depicts an environment in which the participants state policies for the supported application, as well as their local enterprise environments. The instance is the result

of the reconciliation of the session, application, and enterprise policies.

An initiator uses the reconciliation algorithm to create a *policy instance* compliant with the session and each domain policy. A policy is compliant if all stated requirements and restrictions are realized in the resulting instance. If an instance is found, it is used to govern the provisioning and authorization of the subsequent session. If an instance cannot be found, then the participants must revise the domain policies or abort the session. An instance concretely defines session provisioning and authorization. The initiator is trusted to reconcile the session and domain policies correctly.[1]

A session policy in Ismene is authoritative; the instance must be fully compliant with the session policy.[2] Domain policies are consulted only where flexibility is expressly granted by the issuer. Hence, the session policy acts as a template for operation, and domain policies are used to further refine the template toward a concrete instance. Conversely, domain policies represent the set of requirements that are deemed mandatory and relevant by the entities that issued them.

3.1 Policy Expressions

Session provisioning identifies the configuration of the security services used to implement the session. Ismene models provisioning as collections of security *mechanisms*. Associated with a mechanism is a set of *configuration* parameters used to direct its operation. Throughout, we use the term configuration to refer to an atom specifying a parameterized mechanism configuration. Each mechanism provides a distinct communication service that is configured to address session requirements. A (provisioning) *policy expression* explicitly states configuration through a set of mechanisms and parameters. To illustrate, consider the following (incomplete) Internet Key Exchange (IKE [Harkins and Carrel 1998]) session policy:

```
cipher: 3DES (and)
hash algorithm: MD5 (and)
exchange algorithm: MODP, (Group 1 (exclusive or) Group 2)
```

This policy states that there are three mechanisms used to implement IKE; a cipher algorithm, a hash algorithm, and a Diffie-Hellman exchange. Moreover, the exchange must use either group 1 or 2 MODP values, but not both or neither. The policy requirements can be expressed more precisely as:

$$Cipher(3DES) \wedge Hash(MD5) \wedge (Exchange(MODP, Group1) \oplus Exchange(MODP, Group2)$$

where each element of the expression specifies a mechanism (e.g., Cipher) and configuration (e.g., $3DES$). More specifically, the provisioning statement mandates, among others, that the runtime system use *3DES* (algorithm) to implement the *Cipher* (function).

Note that this policy must be further refined for it to be enforced; the session participants (IKE initiator and responder) must agree upon an exchange group (group 1 or group 2). *Provisioning reconciliation* resolves these ambiguities by

[1]Where deemed necessary, participants can efficiently validate an instance against the relevant domain policies prior to acceptance of the instance (see Section 5.1).

[2]Where no such authoritative policy is available, a default session policy that places no constraints on session security is used. In that case, participant domain policies are reconciled to derive the instance, and the default (session) policy where domain policies provide no guidance.

attempting to find an *instance* that is consistent with each policy expression. Where multiple policies are considered, each must be satisfied.

In the remainder of this paper, policy statements identifying a range of mutually exclusive behaviors (identified by the XOR operator $\oplus$) are called **pick** statements. There exists a number of other desirable configuration semantics beyond exclusive selection. For example, a policy might require a threshold of configurations or represent "many out of many" selection. We chose this semantic because exclusivity is the semantic of extant policy systems. For example, an IKE encryption algorithm must exclusively select an algorithm to encrypt data.

Policy expressions give an alternative and more general way of viewing the reconciliation problem than that provided in current policy languages. For example, in IKE, a requester (acting as the entity providing a domain policy) must provide a proposal that precisely mirrors that of the responder (whose policy represents a session policy). IKE reconciliation trivially finds an intersection of the fields of the policy proposal. In contrast, reconciliation in Ismene is formulated as a satisfaction problem; the initiator seeks an instance that satisfies the set of expressions. Hence, the provisioning expression in domain policies need only specify those aspects of policy that the issuer wishes to influence.

Authorization policy maps identities or credentials onto a set of access rights [Woo and Lam 1993]. As in provisioning, authorization statements are modeled as logical expressions. Each authorization expression, called an *action clause*, is defined as a conjunction of positive conditions.[3] For example:

$$read : ACL(/etc/hosts, bob, read) \wedge ID(bob) \wedge FILE(/etc/hosts)$$

states that "read operation should succeed if the user is Bob, the file being accessed is /etc/hosts, and the ACL for the file allows read access to Bob". As in other systems such as KeyNote [Blaze et al. 1996], the interpretation of each condition is left to the environment; the establishment of the identity, file, and the evaluation of the file's ACL is outside the scope of the policy specification. Note that this clause represents a canonical access control policy; the subject (*bob*), object ($/etc/hosts$), and action (*read*) is mapped onto the access control matrix (ACL).

The Ismene language and our policy model came about from our detailed analysis of existing policy systems [McDaniel et al. 1999; McDaniel and Prakash 2005]. Centrally we found flexible systems were built around the largely independently configured software components [Hutchinson and Peterson 1994; Schmidt et al. 1993; Orman et al. 1994; Bhatti et al. 1998; Moriconi et al. 1997; Nikander and Karila 1998; Hiltunen et al. 2000]. The key insight of these works was that systems can be assembled on the fly as dictated by run-time requirements. However, the policy driving these component constructions was often static or ad-hoc; there was no theory or model that would help define how the component could be composed legally and securely. Recent systems, including our Antigone system have embraced this approach to building secure systems. However, this does not imply that component systems are the only domain in which the present work is relevant; any

[3]Because of the complexity imposed by the negative conditions, we only consider positive conditions in this paper. As many systems adopt this approach [Blaze et al. 1996], this does not significantly affect our ability to represent existing policies (see Section 6)

system which desires to reconcile disparate security requirements must face these issues.

A requirement arising from the previous work was the need for controlling the ways in which software components are defined. That is, some legal policies may be, for systemic or performance reasons, inoperable. Hence, we needed some way of guiding the evaluation of policies that were not only consistent with stakeholder desires, but those that would work well. We consider how to ensure this through the analysis algorithms in Section 5.2.

3.2 Policy Formalism

We introduce the following formalism to describe the semantics and reason about Ismene policy. All Ismene session and domain policies are defined under a *policy system* and contain a provisioning and authorization policy as described below. A policy system is defined as the tuple $\langle \Lambda, \Phi, \Gamma, \Sigma \rangle$. $\Lambda = \{m_1, \ldots, m_i\}$ is a set of security mechanisms used to implement the session. $\Phi = \{c_1, \ldots, c_j\}$ is the set of all possible configuration states of the mechanisms in Λ. The set $\Gamma = \{a_1, \ldots, a_k\}$ is the set of security relevant actions governed by the session, and $\Sigma = \{d_1, \ldots d_n\}$ is the set of security relevant systems states (i.e., conditions). To illustrate, the partial policy system supporting the examples described in the preceding section includes:

$\Lambda = \{Crypto, Hash, Exchange\}$
$\Phi = \{Crypto(3DES), Hash(MD5), Exchange(MODP, Group\,1), Exchange(MODP, Group\,2)\}$
$\Gamma = \{read\}$
$\Sigma = \{ACL(/etc/hosts, bob, read), ID(bob), FILE(/etc/hosts)\}$

A pick statement is an unordered set of mutually exclusive configuration states $c_1, \ldots, c_l \in \Phi$. Pick statements are denoted $pick(c_1, \ldots, c_l)$, or by their logically equivalent *policy expression* $(c_1 \oplus \ldots \oplus c_l)$. A configuration state *satisfies* ($\vdash$) a pick expression if it occurs in the expression, i.e., $c_i \vdash pick(c_1 \oplus \ldots \oplus c_l)$ if $c_i \in c_1, \ldots, c_l$. Note that we use $\oplus$ symbol throughout to denote an "exactly one" logical operation, e.g., the clause $(a \oplus b \oplus c)$ is satisfied where exactly one of a, b, or c is true (see below). This is explicitly not intended to denote the logical "exclusive or" operation, and greatly reduces the notational complexity of the following exposition.

A provisioning policy g is an unordered set of pick statements $P = \{p_1 \ldots p_q\}$ representing the configurations needed to implement the session. g represents a logical conjunction of pick statement expressions, and is denoted $pick(c_1, \ldots, c_l), \ldots, pick(c_{l+k}, \ldots, c_m)$, or by its equivalent policy expression, $(c_1 \oplus \ldots \oplus c_l) \wedge \ldots \wedge (c_{l+k} \oplus \ldots \oplus c_m)$.

A provisioning policy *instance* N is subset of elements of Φ. N is said to be *consistent* with a policy g (denoted $N \preceq g$) if every pick statement in g is satisfied by exactly one configuration in N. N *satisfies* a policy g (denoted $N \vdash g$) if it is consistent with g and every element of $n_i \in N$ is present in one or more pick statements in g. Note that the elements of Φ can be modeled as boolean variables. Assume a truth assignment where $\forall c_i \in \Phi$, c_i is true if $c_i \in N$, and false otherwise. $N \vdash g$ if the truth assignment given by N satisfies the policy expression for g. This highlights a key insight of this work; policy can be expressed in first order logic. We exploit this representation to inform the operations and limitations of policy under this model.

The conditions in Σ are atoms representing the evolving state of the environment. An action clause is a conjunction of conditions in Σ and a single action in Γ. A clause indicates logical implication, where a particular action is allowed where the conditions hold. Also known as access control rules, these clauses are denoted $a_k : d_0, \ldots, d_j :: accept$[4] or by their logical equivalent (called an action expression), $d_0 \wedge \ldots \wedge d_j \rightarrow a_k$. Continuing with our logical model, we note that instantaneous condition values represent a truth assignment for variables representing the elements of Σ. Satisfaction of an action expression with this truth assignment *implies* a truth assignment for the relevant element of Γ, i.e., true where $d_0, \ldots, d_j$ are satisfied and false otherwise. Hence, an action clause is satisfied when the truth assignment for Σ satisfies its conditions $d_0, \ldots, d_j$. Satisfaction only signifies that an action is *allowed*. The truth assignment for Σ defines the permissions of a session in the current environment. However, this indicates nothing about which permissions are exercised.

An authorization policy p is the set of action statements that govern the session. The semantics of an authorization policy states that any satisfying action clause is sufficient to permit access, i.e., as formulated with the expression $(d_0 \wedge \ldots \wedge d_j \rightarrow a_k) \wedge \ldots \wedge (d_0 \wedge \ldots \wedge d_n \rightarrow a_m)$. As before, the truth assignment of Σ, when evaluated over the authorization policy expression implies a truth assignment for the elements of Γ. Actions associated with clauses that evaluate to true under this truth assignment are allowed. This model represents a *closed world policy*, where an action is allowed only where it is explicitly granted by the successful evaluation of an action clause.

The remainder of this section considers the enforcement and use of policy in Ismene within this framework. We begin by considering the meaning and complexity of provisioning policy reconciliation.

3.3 Policy Semantics

We here sketch the semantics of our policy model. We begin by considering a provisioning policy expression. Every configuration $c_i \in \Phi$ is a term.[5] We now inductively define policy expressions and instances:

Definition 3.1. *Provisioning Policy Expression*

(i) if $\phi_1, \ldots, \phi_k$ are terms and $k > 0$, then $pick(\phi_1, \ldots, \phi_k)$ is an expression

(ii) if $\tau_1, \ldots, \tau_k$ are expressions and $k > 0$, then $\tau_1 \wedge \ldots \wedge \tau_k$ is an expression

Definition 3.2. *Policy Instance* (or just *instance* where the the context is clear)

(i) if ϕ_i is a term, then ϕ_i is an instance

(ii) if ϵ is an instance and ϕ_i is a term, then $\epsilon \wedge \phi_i$ is an instance

[4]The `accept` keyword closing each clause indicates that the action is allowed where the conditions are met. `accept` is intended as syntactic sugar, and is present in all action clauses. There is no `deny` in the model; authorization fails unless explicitly accepted.

[5]Note that the set of mechanisms Λ is specifically omitted from the formal definition of the policy model. We introduce them because they are useful in relating the higher level system software components to policy.

The semantics of the provisioning policy are largely defined by satisfaction of a policy by an instance. Such is the purpose of a policy, to find a particular system configuration that precisely meets the requirements of the policy.

Definition 3.3. *Valuation function* : we say $val(\epsilon, \phi)$ is the valuation of a term ϕ with respect to an instance ϵ. This function is inductively defined as follows.

(i) $val(\epsilon, \phi) = true$ if $\phi \in \epsilon$ and *false* otherwise

(ii) $val(\epsilon, pick(\phi_1 \dots \phi_k)) = true$ if $val(\epsilon, \phi_i) = true$ for exactly one $i \in (1, k)$. Implicitly $val(\epsilon, \phi_k) = false\ \forall j \in (1, k), j \neq i$

Definition 3.4. *Satisfaction relation* : we say $\mathcal{M}, \epsilon \models \tau$ to say that instance ϵ satisfies the policy expression τ. The relation $\models$ is defined as follows.

(i) $\mathcal{M}, \epsilon \models pick(\phi_1 \dots \phi_k)$ iff $val(\epsilon, pick(\phi_1 \dots \phi_k)) = true$ for exactly one $\phi_i \in \{\phi_1 \dots \phi_k\}$

(ii) $\mathcal{M}, \epsilon \models \tau_\infty \wedge \tau_\in$ iff $val(\epsilon, \tau_1) = true$ and $val(\epsilon, \tau_2) = true$

An authorization policy is defined similarly over Σ and Γ. We briefly define the semantics of the authorization policy model below.

Definition 3.5. *Authorization statement*

(i) if $\gamma_j \in \Gamma$ and $\sigma_1 \dots \sigma_k \in \Sigma$ and $k > 0$, then $\gamma_j : \sigma_1 \dots \sigma_k$ is an authorization statement.

An *environment* E is a valuation for all conditions conditions Σ, i.e., $\sigma_i = \{true, false\}\ \forall \sigma_i \in \Sigma$. An authorization statement a is satisfied by an environment E, denoted $E \models a$, iff $\sigma_i = true\ \forall \sigma_i \in (\sigma_1 \dots \sigma_k)$. An *authorization policy* is the set of authorization statements in a policy $A = a_1 \dots a_k$. An action $\gamma_j \in \Gamma$ is satisfied by a policy and environment E, denoted $(A, E) \models \gamma_j$, iff $\exists a_i \in A$ such that $E \models a_i$ and $\gamma_j \in a_i$.

3.4 Provisioning Reconciliation

In its simplest form, provisioning reconciliation searches for an instance N that satisfies a considered session policy (the more formal and complete definition of reconciliation is given below). The policy is said to be *reconcilable* if such N exists. The following shows that in its most general form, this simple form of reconciliation is intractable; any instance of positive, one-in-3 satisfiability [Schaefer 1978; Garey and Johnson 1979] (a known intractable problem) can be reduced to the problem of finding a solution that satisfies a policy expression with pick statements.

Definition 3.6. Unrestricted Policy Reconciliation (UPR) - **Given**: A session policy g within a policy system $\langle \Lambda, \Phi, \Gamma, \Sigma \rangle$. **Question**: Is there an instance N satisfying all configuration and pick statements in g, i.e., $N \vdash g$?

Definition 3.7. Positive, ONE-IN-THREE 3SAT (13SAT+) - **Given**: Set U variables, expression $e = C$ disjunctions over U such that each $c \in C$ has $|c| = 3$, no negated literals. **Question**: Is there a truth assignment for U such that each clause in C has exactly one *true* literal?

THEOREM 3.8. *Unrestricted Policy Reconciliation (UPR) is NP-complete.*

Proof: Note that a non-deterministic Turing machine can simply guess a satisfying instance N for g and validate it in polynomial time. Therefore, UPR is in NP. We show completeness via reduction of Positive, ONE-IN-THREE 3SAT to UPR in polynomial time:
Construction: Assume $U = \{x_1, x_2, \ldots, x_n\}$. For each $c_i \in C, c_i = (x_1 \vee x_2 \vee x_3)$, create the pick statement $pick(x1, x2, x3)$. For example, the expression $(a \vee b \vee c) \wedge (a \vee c \vee d) \wedge (b \vee d \vee f)$ would generate the following policy:

$$g = pick(a, b, c), pick(a, c, d), pick(b, d, f)$$

Any instance resulting from UPR must specify exactly one configuration from each pick statement. Trivially, such an instance of g represents satisfying truth assignment for the expression e. Hence, because 13SAT+ is NP-complete [Schaefer 1978], so is UPR. □

This result is in stark contrast to needs of policy management; the algorithms used to manage policy must be efficient. In response, we place the following restriction of the construction of policy:

> *Policy Restriction*: A mechanism configuration $c_i \in \Phi$ can only be stated in at most one pick statement p_i in a policy g, i.e., $\nexists c_i \quad | \quad p_j, p_k \in g, j \neq k, c_i \in p_j, c_i \in p_k$.

For example, if a, b, and c are mechanism configurations, the following policy expression is not allowed by the above restriction in a single policy because a occurs twice in the policy expression:

$$(a \oplus b) \wedge (a \oplus c)$$

On the other hand, the policy expression presented in Section 3.1 is legal because $Exchange(MODP, Group1)$ and $Exchange(MODP, Group2)$ are considered different mechanism configurations, though they refer to the same mechanism. Note that reconciliation of a single policy becomes trivially tractable under this reconciliation: a satisfying N is constructed by randomly selecting exactly one configuration for each pick statement in g. For the remainder of this article, all policies are assumed to be constructed under this restriction. Based largely on hierarchical policy templates, we have investigated other policy models that can be efficiently reconciled [Wang et al. 2004]. We have found that the restriction places requirements on the way in which mechanisms are defined, but does not unduly limit the expressivity of policy. We discuss this point within the context of the Ismene language in Section 4.3.

Before considering a tractable reconciliation algorithm we introduce the notion of *equivalent configurations*. An equivalent configuration is a set of two or more configurations contained within the same pick statement in two policies being reconciled. Consider the pick statements representing the logical expressions $(d \oplus e \oplus f)$ and $(d \oplus e \oplus b)$ in the example session and domain policies in Figure 3. d and e can be considered to be equivalent configurations; any instance including d can replace d with e and still satisfy both policies. Hence, the pair of configurations can, for the purposes of reconciliation, be treated as a single atom. This key insight leads to the following algorithm.

We now extend Ismene policy reconciliation to the multi-policy case. An Ismene reconciliation algorithm accepts a session policy g and a set of domain policies $l_i \in L$ and returns a policy instance N such that $N \vdash g, \forall l_i \in L | N \preceq l_i$, if

Two-Policy Reconciliation Algorithm (TPR)

(1) Any pick statement in the session policy that does not contain a configuration present in the domain policy is discarded, (the domain policy does not provide any guidance of that pick, and any configuration can be used).

(2) Any pick statement in the domain policy that contains no configurations present in the session policy can not be reconciled, and hence the reconciliation process fails. This is because the resulting instance could not simultaneously satisfy the session policy and be consistent with the domain policy.

(3) Collapse *equivalent configurations*, if necessary. Equivalent configurations must be replaced with a single place-holder configuration (representing the set of equivalent configurations) in step 2, and restored in the instance after reconciliation is completed. For two policies, equivalent configurations can be easily found in polynomial time by simply looking for overlap between pick statements of the two policies. With equivalent configurations, the output of 2-policy reconciliation can be a policy expression, rather than a concrete instance. Therefore, the resulting policy can be further refined by reconciliation with other policies.

(4) Reduce the session and domain policies. *Repeat steps 4a and 4b until no further reduction is possible*

(a) Remove each configuration in the session or domain policy that is not in the other policy. If any pick statement becomes empty, then the policy cannot be reconciled, and the algorithm halts and returns *irreconcilable*.

(b) Remove any pick statement containing a single configuration in one policy, and remove the corresponding pick statement containing that configuration in the other. Place the single configuration in the instance.

(5) Model the remaining policies as a bipartite graph. Add a node to the left side of the graph for every pick statement in the session policy. Repeat for the right side of the graph for every pick statement in the domain policy. Add an edge between two nodes if they share a configuration, and label it with that configuration. A reconciled policy is found by finding a perfect matching for the bipartite graph. Where a matching is found, the configurations associated with each edge used in the matching are added to the reconciled policy. Hence, the algorithm is completed by applying an efficient algorithm for perfect matching on the constructed bipartite graph [Karpinski and Wagner 1988]. If no matching is found, then the policies cannot be reconciled (see below), and the algorithm returns *irreconcilable*.

Fig. 2. The two policy reconciliation algorithm (TPR) - efficiently reconciles one session and one domain policy.

one exists. Before considering this problem in its full generality, we introduce an algorithm that efficiently reconciles a session and one domain policy in Figure 2.

Figure 3 illustrates the execution of TPR on an example session and domain policy (whose initial encoding is shown in part *a* of the figure). In this example, neither policy is reduced by step 1 (non-intersecting pick statements). Part *b* in the figure shows the result of the reduction of the policies in response to step 3 and 4a, where the place-holder configuration δ replaces d and e and configurations that are not present in the other policy (a, c, i, and l) are discarded. Part *c* shows the result of the application of algorithm step 4b, where pick statements with a single configuration are removed and the configuration added to the instance (f and b), and the intersecting picks in the other policy are discarded (δf) and (δb)). At this point, no further reduction by steps 4a or 4b is possible, and the graph described in

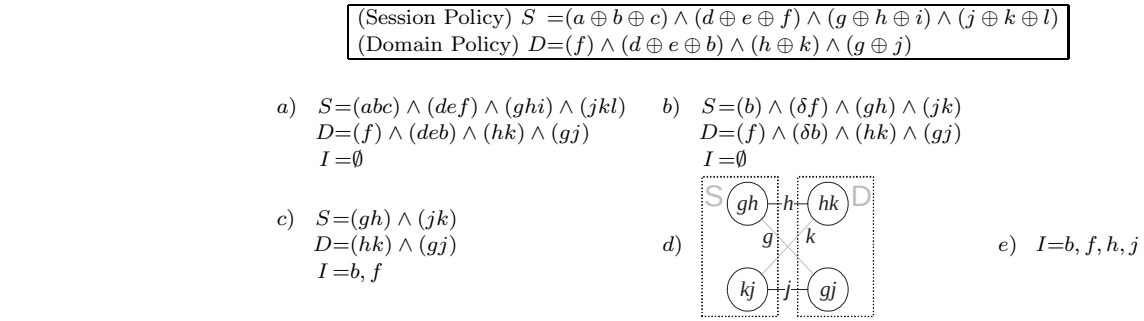

Fig. 3. Reconciliation - the Ismene reconciliation algorithm iteratively reduces the intersection of the session (S) and domain (D) policies. Any reconcilable policy will converge on configurations (denoted by single letter variables – e.g. a) existing exactly once in each policy. The remaining pick statements can be reconciled into a concrete instance (I) using an (efficient) bipartite matching algorithm.

part d is constructed as directed by step 5. A bipartite matching is found and the edge labels are added to the instance (h,j). The algorithm terminates by returning the policy instance (described in part e). We consider the correctness of this process in the following theorem.

THEOREM 3.9. *The TPR algorithm returns an instance if and only if the policies are reconcilable.*

Proof sketch:[6] The following explores each implication of this theorem independently. This proof assumes that every pick statement in the session policy contains at least one configuration present in the domain policy. Such non-intersecting statements would be discarded in step 1, and have no bearing on the correctness of the algorithm (are not considered by reconciliation). For the same reasons, we do not consider equivalent configurations, which are collapsed prior to reduction and matching (steps 3-4). The session policy and domain policy being reconciled are denoted S and D, respectively.

(1) Implication 1 (**if**) - *TPR algorithm returns an instance where the policies are reconcilable.* Assume the TPR returns *irreconcilable* where the policies are reconcilable. There must have been some pick statement p_i that became empty in step 4a, or no matching was identified in step 5.
   (a) Assume p_i became empty in step 4a. For this to occur, every configuration $c_j \in p_i$ must have been removed in step 4a. If a c_j did not occur in the other policy at any point during reconciliation, then it could not be part of any instance, and was correctly discarded. Every remaining configuration $c_j \in p_i$ must have been removed as the result of some mandatory configuration (i.e., contained in the other policy by a pick statement removed in step

[6]The formal proof is significantly more complex than the sketch presented here, and for brevity is omitted. We do not comment further on the formal proof other than to state that it inducts on the number of elements in pick statements.

4b). The application of step 4b represents a mandatory configuration; *any* instance must contain the singular configuration because all other configurations in that pick cannot be selected (either because they do not intersect with the other policy or are ruled out by other mandatory configurations). Because every $c_j \in p_i$ was correctly discarded in step 4a, no $c_j \in p_i$ could be included in a valid instance, and p_i cannot be reconciled. Thus, p_i could not have become empty in step 4a.

(b) Assume that no matching for reconcilable policies was found in step 5. Because no pick statement will become empty if the policies are reconcilable (from above), every pick statement in the original policies must be satisfied by a mandatory configuration or considered by the matching algorithm in step 5. Because every configuration in any valid instance must intersect with exactly one pick statement in each policy, any two reconcilable policies will contain the same number of pick statements. The discovery of each mandatory configuration removes from consideration (satisfies) one pick statement from each policy. Hence, the number of pick statements in each policy used to construct the graph in step 5 will be equal (denoted n). If the matching algorithm fails, then there is no collection of n edges that cover all nodes. Because each configuration occurs only once in each policy, this signals that there is no set of n configurations (edges) that satisfy all pick statements (nodes). Hence, if no matching is found, the policies are not reconcilable, a contradiction.

Note that we also must prove that any instance returned by the algorithm satisfies all the policies, i.e., is valid. Consider an instance I. For all pick statements removed in step 1, then one $c_j \in I$ was added to I. Because they do not intersect with the domain policy and by the policy restriction, they satisfy exactly one pick statement in the session policy. Because equivalent configurations can be treated as a single configuration, we can ignore them for the purposes of validity.

Clearly, no configuration is added to I that does not satisfy any pick statement. It is sufficient to show that all remaining pick statements from the session and domain policies are satisfied by exactly one $c_j \in I$. Intersecting pick statements are removed in step 4b or 5. If removed in step 4b, no other configuration listed in the pick statement could be added to I prior to the adding of the satisfying configuration (that would have satisfied the clause) or subsequently (because that would require that the other configuration occur in the policy in more than one pick statement). Hence, the removed pick statements will be satisfied by exactly one configuration. Now consider step 5. By construction, the algorithm will ensure that the bipartite matching introduces configurations that satisfy exactly one clause in each policy, and that all pick statements are satisfied. Moreover, because it models the "remaining" pick statements, the matching must satisfy all the previously unsatisfied pick statements.

Thus, where the policies are reconcilable, neither failure case can occur, and TPR always returns a valid instance.

(2) Implication 2 (**only if**) - *TPR does not return an instance where policies are irreconcilable.* Assume an instance I is returned where policies are irreconcilable.

Because I is not a compliant instance, there must be some pick statement p_i in S or D which has zero or two or more configuration variables in common with I (i.e., $\exists\, p_i \in S$ or $p_i \in D$ such that $|p_i \cap I| \neq 1$). By definition, p_i must have either been satisfied in step 4b or incident to some edge the bipartite matching found in step 5. Assume p_i was satisfied in step 4b. Because p_i was reduced to a single element or contained the configuration of a pick statement reduced in the other policy, $|p_i \cap I| \geq 1$. Because all picks are removed that contain the configuration added to an instance in step 4b and each configuration occurs only once per policy, $|p_i \cap I| < 2$. Hence, $|p_i \cap I| = 1$, and p_i must have been matched in step 5. Because matching requires that exactly one edge is incident to each node $|p_i \cap I| = 1$, a contradiction. Hence, TPR does not return an instance where the policies are irreconcilable. □

n-Policy Reconciliation Algorithm (NPR)

In the case where more than one domain policy needs to be reconciled with a session policy, a simple algorithm would be to reconcile the session policy with one domain policy at a time. The policy expression resulting from each 2-party reconciliation is used as the session policy for reconciliation with the next domain policy. As a final step, a specific configuration is chosen from pick statements remaining after the final reconciliation (due to equivalent configurations). A reasonable strategy chooses the first configuration in each remaining pick statement from the session policy, assuming that the session policy lists configurations in decreasing order of preference.

The ordering of the reconciliation of each policy within the algorithm may affect the reconciliation results; some orderings of domain policies will not be reconcilable, while others will. For example, consider the following session and domain policies:

$$\begin{array}{cc} \textit{Session Policy} & (a \oplus b) \wedge (c \oplus d) \\ \textit{Domain Policy } 1 & (b \oplus e) \wedge (c \oplus d) \\ \textit{Domain Policy } 2 & (b) \wedge (d) \end{array}$$

If domain policy 1 is considered first, the policies *may* reconcile to $(b \wedge c)$, and domain policy 2 could not be reconciled. If domain policy 2 is considered first, reconciliation arrives at $(b \wedge d)$, and thus be reconcilable with domain policy 1. Hence, because the algorithm commits to some configurations, the reconciliation of individual policies is neither associative nor commutative. Note that this is specific to this algorithm; there may exist algorithms that reconcile policies without this limitation.

If one were to attempt to reconcile these policies at the same time, the introduction of a third policy would violate the property that a configuration occurs in at most two pick statements in the reconciliation expressions. We further show that the general n-policy reconciliation case is intractable by a reduction from the one-in-three satisfiability in the following theorem.

Definition 3.10. *3-Policy Reconciliation (3PR)* - Given: A session policy g and two domain policies $L = \{l_1, l_2\}$. Question: Can g, l_1, and l_2 be successfully reconciled?

Definition 3.11. *3-Dimensional Mapping (3DM)* - Given: set $M \subseteq W \times X \times Y$, where $|W| = |X| = |Y| = q$, and $W \cap X = X \cap Y = Y \cap W = \emptyset$. Question: Is there

an $M' \subseteq M, |M'| = q$, such that no two elements of M' agree in any coordinate (every $w_i \in W, x_j \in X, y_k \in Y$ occurs in M' exactly once) [Garey and Johnson 1979].

THEOREM 3.12. *n-Policy Reconciliation (NPR) is NP-complete.*

Proof: By reduction from 3DM. Perform the following polynomial time construction to create the policies g, l_1, and l_2 from an instance of 3DM.

(1) create a configuration m_i for each $M_i \in M$.
(2) create a clause c_i in a policy g for each $w_i \in W$
    (a) add configuration m_i to clause c_i if $w_i \in M_i$.
(3) create policy l_1 by repeating steps 2 and 2a from set X, and l_2 from set Y.

Note that this construction preserves the restriction placed on reconciliation. By definition, any configuration m_i occurs at most once in the g because the associated M_i contains exactly one element of W (and l_1 and l_2 contain exactly one element of X and Y, respectively). For the purposes of the following discussion, we denote an arbitrary solution for 3PR as I, and the set of $M_i \in M$ mapped to the $m_i \in I$ in step 1 of the construction as M''.

It is sufficient to show that any solution to 3PR is a solution for 3DM and vice versa. We begin by observing that $|W| = |X| = |Y| = q$ implies each policy contains at most q clauses. Moreover, if a solution for 3DM exists, each policy contains exactly q clauses. If not, then there exists some variable in W, X, or Y that does not occur in any $M_i \in M$. Such an occurrence trivially indicates an instance of 3DM that has no solution. Because each configuration $m_i \in I$ contains exactly one value from W, X, and Y, $|I| = q$ (if $|I| \neq q$ some clause would have more or less than 1 satisfying configuration in I). By construction $|I| = q$ implies $|M''| = q$. Each clause is satisfied by exactly one element in I because $|I| = q$ and there are q clauses. A given $w_i \in W$ occurs in only a single clause in g, so no w_i appears more than once in I. For the same reason, no $x_i \in X$ or $y_i \in Y$ appears more than once in I. Hence, I represents an instance for which the corresponding M'' has each element in X, Y, or Z appearing exactly once. Thus, any solution I for 3PR is a solution for 3DM (M''). Similarly, any solution for 3DM is a solution for 3PR.

NPR is in NP because a non-deterministic Turing machine can simply guess a satisfying reconciliation for g, l_1, and l_2 and validate it in polynomial time. Because 3DM is NP-complete, so is 3PR. To complete the proof we observe that 3PR is a special case of NPR. Because NPR is more general and 3PR is NP-complete, it follows that NPR is NP-complete. Note that that the problem remains NP-complete even when policies obey the policy restriction. □

A key question is whether intractability really matters. If the numbers of policies and pick statements are few, then why not use brute force satisfiability algorithms? Without question, many environments will not present complex enough policies such that computation becomes a problem. However, our experience in the Antigone system shows that as the number of policies grows reconciliation is likely to become a problem. We have observed this particularly in large multi-party environments and ubiquitous systems, and is likely to be similarly troublesome in

any large, heterogeneous environment. Hence, the complexity of reconciliation is likely to impact systems' ability to quickly define secure software configurations.

Where reconciliation is not possible, it may be desirable to find a subset of policies that *can* be reconciled.[7] One potential reconciliation algorithm, Largest Subset Reconciliation (LSR), would attempt to find an instance reconcilable with the largest number of domain policies. LSR has the undesirable property that it may fail to allow the participation of required members. Moreover, as demonstrated below, LSR is intractable.

Definition 3.13. Largest Subset Reconciliation (LSR) - **Given**: A session policy g and a set of domain policies L to be considered by reconciliation, and a number k. **Question**: Does a $\hat{L} \subseteq L$ exist such that g and all policies $l_i \in \hat{L}$ are successfully reconciled and $|\hat{L}| \geq k$?

THEOREM 3.14. *Largest Subset Reconciliation (LSR) is NP-complete.*

Proof: 3PR is a special case of LSR, where $K = |L| = 2$. Hence, because 3PR is NP-complete, so must LSR.

3.4.1 *Heuristic Reconciliation.* The intractability of multi-policy reconciliation led us to an investigation of heuristic algorithms. The simplest heuristic algorithm simply establishes an ordering of domain policies. Higher prioritized policies are reconciled against the session policy first and lower priority policies are reconciled only when higher priority policies provide no guidance, otherwise they are excluded. This polynomial time algorithm is used to derive the security policy in the Antigone communication system [McDaniel et al. 2001].

Our experience has shown that pick statements often intersect with at most one pick statement of all other policies. For example, all IKE policies define similar pick statements for *Crypto*, *Hash*, and *Exchange* mechanisms. In this case the problem of n-policy reconciliation is tractable, e.g., by efficiently identifying the common configuration values in each of the non-intersecting pick statements. Any violation of this property (over a set of session and domain policies) can be efficiently detected by a simple scan of the policies—in that case, the heuristic suggested above of prioritizing domain policies can be used.

Domain policies often address very specific security requirements. For example, organizational policies are likely to broadly restrict the ways in which communication occurs, and personal policies often specify fine-grained service configuration. Because these policies address different aspects of the session, they infrequently intersect. One algorithm exploits this fact by parallelizing reconciliation. L is initially divided into k sets of *independent* policies. Two policies are independent if a) they share no common configurations, b) they share no configurations in the same pick statement in g and, c) are independent of policies in the transitive closure of parts a and b. Each set of policies is reconciled with g independently to arrive at an instance $n_i, \forall i = 1 \ldots k$. N is the union of the configurations of all n_i, and a random selection of single configurations from pick statements that do not intersect with

[7] We acknowledge that many approximation algorithms for these and related satisfiability problems exist. A study of these algorithms may lead to efficient approximation of reconciliation (i.e., reconcile a fixed fraction of policies), but defer their consideration to future work.

any domain policy. By construction, $N \vdash g, N \preceq L$ because every pick statement in g is satisfied by exactly one configuration in an instance n_i or by a single randomly selected configuration in N.

These algorithms demonstrate that policies can be heuristically, and in our experience, practically reconciled. Whether by restricting the policy structure, separating service policies, or by simple ordering, one can quickly converge on a session provisioning, if one exists. How one deals with irreconcilable policies is of considerable systemic consequence, but is beyond the scope of this section. We consider these and other operational issues in Section 7.

3.5 Authorization Reconciliation

The authorization policy defined in an instance is the result of the reconciliation of action clauses of all considered policies. However, the semantics of such an operation are unclear; one may view reconciliation of access control to be a logical OR of each policy, i.e., any action that would be allowed by at least one domain policy would be allowed, a logical AND, i.e., the session and all domain policies must be satisfied, or something else (the session AND at least one domain policy must be satisfied). The first approach (logical OR), however, has the unfortunate side-effect that a permissive domain policy can circumvent any controls stated in the session or domain policies.

As described above, our Authorization Reconciliation Algorithm takes the conservative approach of accepting the logical AND of all access control policies. This approach will not allow any controls to be circumvented; however, an overly restrictive domain policy can place significant barriers to session progress. We discuss our experience with this issue further in Section 7.

We now illustrate authorization reconciliation. Consider an example session policy that defines two action clauses ($t_1 : c_1 :: \mathit{accept};$) and ($t_1 : c_2 :: \mathit{accept};$) and two domain policies with action clauses (t_1: c_3 :: accept;) and (t_1: c_4 :: accept;), respectively (where t_1 is an action and each c_i a condition). The resulting policy from reconciliation is:

$$t_1 : ((c_1 \vee c_2) \wedge c_3 \wedge c_4) :: \mathit{accept}$$

Action clauses are defined in all policies and instances in disjunctive normal form (DNF) [Mendelson 1997]. This representation enables efficient compliance testing (see next Section). For example, the above policy would be stored in the instance as:

$$t_1 : c_1 \wedge c_3 \wedge c_4 :: \mathit{accept}$$
$$t_1 : c_2 \wedge c_3 \wedge c_4 :: \mathit{accept}$$

Note that due to the conversion to DNF, the number of action clauses for a given action a in the instance resulting from the reconciliation of two policies is equal to the product of the number of clauses for a in each policy. For example, if policy X defines 2 clauses for a and policy Y defines 3, the instance resulting from X and Y's reconciliation will contain $2 * 3 = 6$ clauses for a. The total number of clauses in the instance is the sum of the products for all actions.

In terms of the number of clauses, the worst case occurs when both policies define a number of clauses for a single action. Assuming both policies define k, reconciliation of these policies would result in $k^2 = k * k$ clauses for that action.

More generally, pair-wise reconciliation of n policies would result in k^n clauses. Each clause is generated in linear time (with respect to the size of the clause). Hence, like provisioning reconciliation, authorization reconciliation is tractable for 2 party reconciliation, and intractable for n party reconciliation.

Note that while the worst-case analysis indicates that n-policy authorization reconciliation is intractable, every real world policy we have encountered could be reconciled efficiently. Few policies exert control over any given action, and those that do infrequently define more than one clause for that action. Hence, for reasonable policies, the growth in the number of clauses is slow, and often sub-linear. Because of the lack of real-world intractability, we have not found a need for complexity mitigating heuristics.

4. ISMENE

This section presents a brief overview of the Ismene policy language. Ismene specifies conditional provisioning and authorization requirements through a general-purpose policy language. A thorough survey of the grammar and semantics of Ismene is presented in [McDaniel 2001]. Ismene policies are collections of totally ordered provisioning and action clauses.

4.1 Provisioning Clauses

Each provisioning clause is defined as the tuple:

```
<tag> : <conditions> :: <consequences>;
```

Tags are used to associate meaningful names with provisioning requirements. Conditions are atoms that identify the circumstances under which the consequences are applicable. These are largely define parameterized functions which call out to the environment and test current conditions. The parameters are specific to the algorithm at hand, and are opaque to the policy algorithms. Consequences state session provisioning requirements through *configurations* and *pick* statements, or identify relevant sub-policies through *tags*. The reserved `provision` tag is used to name the overall provisioning requirements. Consider the following simple example, where x, y, z, and w specify mechanism configurations:

```
provision: :: confidentiality, rekeying;
confidentiality: c1, c2 :: x, y;
confidentiality: :: pick(w, z);
rekeying: :: d
```

The first (provision) clause says that the policy must provision both confidentiality and key management services (tags). The second and third clauses state that if $c1 \wedge c2$ is true, x and y must be configured; otherwise either w or z (but not both or neither) must be configured. The final clause says that d must be configured under all circumstances. Therefore, the policy expression used as input to reconciliation is $x \wedge y \wedge d$ where $c1 \wedge c2$ is true at the time of reconciliation, and $(w \oplus z) \wedge d$ where $c1 \wedge c2$ is false. Note that the ordering of clauses with the same tag (e.g., confidentiality tag) dictates the order of evaluation. If the conditions of an earlier instance of the tag hold (e.g., $c1 \wedge c2$), those consequences (e.g., x and y) must be enforced, and the subsequent clauses for the same tag are ignored. Note that any number of clauses can be defined for the same tag. They will be evaluated in

```
% Ismene Provisioning Clauses
provision : PrivSession($inaddr,$ipt,$oaddr,$opt) :: strong_key_mgmt, confidentiality;
provision : :: weak_key_mgmt, confidentiality;
strong_key_mgmt: Manager($ent) :: config(dh_key(refresh,60));
strong_key_mgmt : :: config(dh_key(refresh,240));
weak_key_mgmt : :: config(lm_key(refresh,300));
confidentiality : :: pick( config(dhndlr(3des)), config(dhndlr(des)) );

% Ismene Action Clauses
join : config(dhndlr(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;
```

Fig. 4. Ismene Policy - The provisioning clauses in the session and domain policies are evaluated to arrive at the policy expressions used as input to reconciliation. Action clauses are evaluated over the lifetime of the session to enforce authorization policy.

order. The first clause whose conditions are satisfied by the environment will have its consequences added to the instance, and the others will be ignored.

Conditions in a clause often refer to *attributes*. An *attribute* describes a single or list-valued invariant (constant variable with constant value). For example, the following attributes define a single-valued version number and list-valued ACL:

```
version := < 1.0 >;
JoinACL := < {alice}, {bob}, {trent} >;
```

An occurrence of the symbol "$" signifies that the attribute should be replaced with its value. As in the KeyNote action environment [Blaze et al. 1999], the *attribute set* is the set of all attributes defined in the policy and provided by the environment. Enforcement infrastructures (e.g., applications) provide additional evaluation context by adding attributes to the attribute set. Condition evaluation is outside the scope of Ismene; the environment in which Ismene is used is required to provide a predicate interface for each condition. This is similar to GAA API condition upcalls [Ryutov and Neuman 2000]) or the Antigone Condition Framework [McDaniel 2003].

Consider the provisioning clauses in Figure 4 that define requirements for public and private sessions of `tc`, the example teleconferencing application introduced in Section 2. If the session is private (as classified by session address attributes), then the `strong_key_mgmt` clauses are evaluated; otherwise `weak_key_mgmt` is evaluated. The `confidentiality` clause is evaluated in either case. The strong key management clause states that a Diffie-Hellman [Diffie and Hellman 1976] keying mechanism must be used. The behavior of this mechanism is further refined to refresh the session key every 60 (240) seconds where a management is (is not) present. When the session is not deemed private, the `weak_key_mgmt` clause simply provisions the Leighton-Micali key management mechanism [Leighton and Micali 1994]. The confidentiality clause instructs the data handler mechanism to use either 3DES or DES, depending on the result of reconciliation.

Note that the mechanisms indicated in the policy specification (e.g., `dh_key` and `dhndlr`) must be provided by the enforcement infrastructure. These are not keywords in the language; mechanism names are mapped to the service implementations by the enforcement infrastructure [McDaniel and Prakash 2002].

4.2 Action Clauses

Each action clause has the following structure:

$$\text{actionName: } c_1, ..., c_n :: \text{accept}$$

The specified action (operation) is allowed if all the conditions hold when the action is attempted (i.e. at run-time). *accept* is the only allowed consequence. Hence, Ismene represents a *closed world* in which denial is assumed. The protected actions are defined by the enforcement infrastructure, and assumed known *a priori* by the policy issuer.

Used exclusively in action clauses, the reserved `credential()` condition evaluates available credentials. All credentials are modeled by Ismene as a set of attributes. For example, an X.509 certificate [Housley et al. 1999] is modeled as attributes for `subj.O` (subject organization), `issuer.CN` (issuer canonical name), etc. To illustrate, consider the following action clause:

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

The first argument of a credential condition (denoted with "&" symbol) represents binding. The credential test binds the matching credentials to the (&cert) attribute. Binding is scoped to the evaluation of a single clause, and conditions are evaluated left to right. The second and subsequent parameters of a credential condition define a matching of credential attributes with attribute or constant values. The above example binds the credentials that were issued by a trusted CA (`sgnr=$ca`) and have the subject name of the participant (`subj.CN=$part`) to the `&cert` attribute. The condition returns true if a matching credential can be found. The enforcement architecture is required to identify the set of credentials associated with an action. Note that the binding symbol "&" symbol has no bearing on the logical interpretation of Ismene (it only effects condition evaluation), and hence has no affect on the semantics of Ismene or operation of the algorithms defined in the preceding sections.

Credential conditions are similar to trust management assertions [Blaze et al. 1996; Chu et al. 1998; Blaze et al. 1999]; evaluation determines whether the attributes of an assertion satisfy the relevant policy expression. Conditions in action clauses can also evaluate whether mechanisms are currently provisioned. Hence, authorization policy can be predicated on session provisioning, e.g., access granted only where particular security mechanisms are used.

Denoted by the `config(...)` statement, provisioning conditions test session configuration. The presence of any configured atom identified in the policy instance simply by placing in the conditions. The semantics of this statement are straightforward—the condition returns *true* if the listed configuration is in the instance and *false* if not. The use of the `config` statement is the means by which authorization policy can be made dependent on authorization policy by (at least partially) predicating access on session configuration.

Consider the action clauses in Figure 4. The first `join` action clause describes an ACL-based policy for admitting members to the session. The member is admitted if she is identified in the `JoinACL` attribute, she can provide an appropriate certificate credential, and the session is provisioned with the DES-enabled data handler mechanism (i.e., `config(dhdnlr(...))`). The second `join` is consulted only when the conditions of the first clause are not satisfied.

The second `join` clause describes a delegation policy. The first credential condition binds `&cert` to the set of credentials delegating join acceptance (the delegation certificates issued by the trusted CA), and second tests the presence of any credential signed by the delegated public key.

4.3 Policy Expressiveness

At first glance, it may appear that the restriction that a configuration appear only once in a policy would severely limit the expressiveness of the policy. One may wish to have a single configuration fulfill two different functions within the session, e.g., a key management protocol mechanism could fulfill both the participant authentication and key agreement functions. In fact, we found such requirements configurations occur frequently in practice in the Antigone system [McDaniel and Prakash 2005].

We work practically around this restriction by parameterizing the features of the protocol. For example, we often used a Kerberos [Neuman and Ts'o 1994] mechanism for user authentication in an Antigone group session, and used a logical key hierarchy [Wallner et al. 1999] for key agreement. We can construct the Kerberos mechanism such that different features could be used (or not used) simply by enabling or disabling a feature, e;g., $kerberosAuth()$, and $kerberosSessionKey()$. Each such mechanism is wholly independent and can be treated as such. Note that we can use the assertions defined below to enforce dependencies and incompatibilities between these modules.

5. POLICY CORRECTNESS

Largely motivated by the Ismene language, this section considers the complexity of algorithms for policy analysis [Li et al. 2005]. These algorithms assess the correctness of a policy within some context (compliance) or in isolation (analysis).

5.1 Compliance

Not all domain policies are required to (or often can) be consulted during reconciliation. Hence, before participating in a session, a participant must be able to *check the compliance of its domain policy with the instance* that is governing the active session. Compliance is successful if all requirements stated in the domain policy are satisfied by the instance. Note that compliance in this work serves a different purpose than the compliance algorithms in trust management [Blaze et al. 1996; Chu et al. 1998; Blaze et al. 1999]; our compliance algorithm determines whether an instance is consistent with a domain policy. In contrast, compliance in trust management systems determines whether a given resource request in the current environment complies with the available policy statements.

As with reconciliation, there are two phases of compliance; provisioning and authorization. The provisioning compliance algorithm compares domain policy with a received policy instance. 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 exactly one configuration is contained in the instance. Provisioning compliance simply tests satisfaction of the domain policy (the conjunction of pick statements in its definition).

Several researchers have examined the problem of compliance in an authorization policy. Gong and Qian's model of a policy composition (i.e., reconciled policies) define a two-principle compliance definition [Gong and Qian 1994]. The *principle of autonomy* requires that any action accepted by one policy must be accepted by the composition (reconciled instance is not less permissive). The second principle, *secure interoperability*, requires that the composition must be no more permissive than either policy. However, this two-fold definition of compliance is extremely restrictive; all policies must specify equivalent authorizations. Moreover, Gong and Qian showed that compliance detection in their model is intractable.

Ismene adopts Gong and Qian's secure interoperability as a definition of compliance, but not the principle of autonomy. More precisely, compliance determines if, for any action and set of conditions, an action accepted by the policy instance is accepted by the domain policy. This embodies a conservative approach to compliance, where any action that would be denied by the domain policy **must** be denied by the instance. Compliant instances always respect the limitations stated in the domain policy and thus the instance can never be less restrictive than the domain policy.

The authorization compliance algorithm assesses whether the instance logically implies the domain policy. Given an expression e_1 describing the conditions of action clauses in an instance and a similar expression describing a domain policy e_2, it is conceptually simple 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 domain policies:

Instance	$X : c_1 \wedge c_2 :: accept;$ $X : c_3 :: accept;$
Domain policy A	$X : c_1 :: accept;$ $X : c_3 :: accept;$
Domain policy B	$X : c_1 \wedge c_3 :: accept;$

The policy instance is compliant with the domain policy A because it is less permissive (i.e., $(c_1 \wedge c_2) \vee c_3 \Rightarrow c_1 \vee c_3$). The instance is not compliant with domain policy B because the session policy is more permissive (i.e., $(c_1 \wedge c_2) \vee c_3 \not\Rightarrow c_1 \wedge c_3$). General-purpose tautology testing is intractable [Cook 1971]. However, the lack of negative conditions and DNF representation leads to an efficient compliance testing algorithm.

We observe that the following condition is necessary and sufficient to prove that the authorization policy in an instance (i) is compliant with policy (p):

> for every action a, the conditions of every clause for a in i must be a superset of the conditions in some clause for a in p.

To see why this is necessary, consider the case where this property does not hold for an action a'. This implies that there exists some conjunction of accepting conditions[8] for a' in i, but not for p. Clearly this violates the $i \Rightarrow p$ implication, and represents a non-compliant policy. Now consider the case where the condition holds.

[8]We use the term *accepting conditions for x in y* to denote a set of conditions which, if true, are sufficient to satisfy the authorization policy y for action x.

Because the (DNF) clauses in p explicitly enumerate the accepting conditions, every accepting conjunction of conditions in i would be accepted by p. Hence, there is no set of accepting conditions of p that are not accepting of i, and the above condition is sufficient.

This observation suggests an efficient compliance algorithm that simply tests that the conditions of each clause in i are a superset of some clause in p (with the same action). The compliance algorithm would deem policy A compliant ($c_1 \wedge c_2 \subseteq c_1$, and $c_3 \subseteq c_3$), and policy B non-compliant ($c_1 \wedge c_2 \not\subseteq c_1 \wedge c_3$ and $c_1 \wedge c_2 \not\subseteq c_3$) in the previous example. Hence, in the worst-case, this compliance test algorithm is polynomial in the number of action clauses.

5.2 Analysis

While a reconciliation algorithm may be able to identify an instance satisfying the session and domain policies, our approach makes no guarantees that the instance is properly formed. A properly formed instance adheres to a set of principles defining the legal use of security mechanisms. An analysis algorithm determines whether a policy or instance is properly formed with respect to a set of rules, called *assertions* governing mechanism use. These restrictions are typically defined by mechanism implementors and used to prevent instances that represent unworkable system configurations.

Assertions are used to define the meaning of properly formed policy by declaring legal and required relations between configurations. Each assertion contains a tag (`assert`), a conjunction of conditions, and a conjunction of consequences. Conditions and consequences are restricted to pick and configuration statement, and may be negated. Semantically, assertions state that the consequences must hold where the conditions are true (i.e., condition conjunction c, consequence conjunction q, $c \Rightarrow q$). For example, an issuer may wish to assert a completeness requirement [Branstad and Balenson 2000] that confidentiality of application data always be provided. Thus, knowing that the *ssl*, *ipsec*, and *ssh* transforms are the only means by which confidentiality can be provided, the issuer states the following (condition-less) assertion expression:

$$(ssl \oplus ipsec \oplus ssh)$$

Analysis determines if an instance (or policy) satisfies the assertion: exactly one confidentiality mechanism must be configured. Other relations are equally important and must be enforced to be usable, e.g., a SSL/TLS protocol mechanism's depends on a certificate mechanisms for acquisition and validation of certificates.

Analysis techniques guaranteeing correct software construction have been studied extensively within component architectures [Hiltunen 1998; Liu et al. 1999]. 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 [Hiltunen 1998] defines the conflict, dependency, containment, and independence relations. The following describes assertion expressions representing these relations (where independence is assumed):

conflict (A *is incompatible with* B)	$!(A \wedge B)$
dependency (A *depends on* B)	$A \Rightarrow B$
containment (A *provides* B)	$A \Rightarrow (!B)$

IKE Session Policy (Responder)

```
provision : selector(12.14.0.0,*,17,23,*,$name)
 :: pick( config(ike(idea-cbc,md5,group1)),
          config(ike(blowfish,sha1,group2)),
          config(ike(cast-cbc,sha1,group2)) ),
    pick( config(preshare()),
          config(kerberos()) );

auth : config(ike(preshare)),
       Credential(&cert,modulus=$prekey.mod)
     :: accept;
auth : config(kerberos()),
       Credential(&tkt,issuer=$realmtgs)
     :: accept;
```

IKE Domain Policy (Requester)

```
provision : selector(*,12.14.9.1,17,23,*)
  :: pick( config(ike(cast-cbc,sha1,group2)),
           config(ike(cast-cbc,md5,group2)) ),
     config(preshare());

auth : config(ike(preshare)),
       Credential(&cert,modulus=$prekey.mod)
     :: accept;
```

Fig. 5. IKE Policy - session (responder) and domain (requester) policies are used to implement IKE phase one policy negotiation. The IKE SA policy (instance) is arrived at through the intersection of the responder (session) policy and requester (domain policy) proposals.

An analysis algorithm assesses whether a policy can or an instance does violate the assertions supplied by the policy issuer. An instance is simply a truth assignment for the universe of variables representing the possible configurations. The *online policy analysis algorithm* (ONPA) evaluates the assertion expressions over that truth assignment. An expression which evaluates to *false* is violated, and the instance is not properly formed. Obviously, by virtue of the tractability of expression evaluation, online analysis is efficient.

An *offline policy analysis algorithm* (OFPA) attempts to determine if **any** instance resulting from reconciliation can violate a set of assertions. However, as demonstrated in the following proof, offline analysis is intractable (coNP). Note that this algorithm need only be executed once (at issuance), and thus does not impact session setup. Moreover, in policies we have encountered, the inclusion or exclusion of related configurations is typically dependent on 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. We present an optimized algorithm for OFPA in [McDaniel 2001].

Definition 5.1. Offline Policy Analysis (OFPA) - **Given**: A session policy g and set of assertions S. **Question**: Can any instance of g violate an assertion in S.

THEOREM 5.2. *Offline Policy Analysis (OFPA) is NP-complete.*

Proof: By reduction of UPR (Theorem 3.8) to OFPA. Create a unsatisfiable assertion in S, e.g., $A \Rightarrow (!A)$. Trivially, a violating instance of g exists if and only if $N \vdash g$. Hence, because UPR is a subproblem of OFPA and UPR is NP-complete, so it OFPA. □

6. MODELING POLICY

This section demonstrates the use of Ismene policy by modeling the semantics of existing policy approaches. These policies serve to highlight the similarities and differences between Ismene and other policy languages and architectures.

DCCM Session Policy (CCNT)

```
provision: ::
  pick( config(conf(3DES)), config(conf(CAST)),
        config(conf(IDEA)), config(conf(RC4)) ),
  pick( config(kman(OFT)), config(kman(LKH)),
        config(kman(DH)), config(kman(pswd)) ),
  pick( config(trans(SSH)), config(trans(SSL)),
        config(trans(IPsec)) );
```

DCCM Domain Policy 1 (member)

```
provision: ::
  pick( config(conf(3DES)),
        config(conf(CAST)) ),
  pick( config(kman(OFT)),
        config(kman(LKH)) ),
  pick( config(trans(SSH)),
        config(trans(SSL)),
        config(trans(IPsec)) );
```

DCCM Domain Policy 2 (member)

```
provision:
  :: pick( config(conf(CAST)), config(conf(RC4)) ),
     pick( config(kman(OFT)) ),
     pick( config(trans(SSH)), config(trans(SSL)) );
```

Fig. 6. DCCM Policy - Designed for policy negotiation in multi-party communication, DCCM creates a session policy through intersection of (domain) policy proposals defined over a template structure (session policy). DCCM does not specify authorization policy.

6.1 Internet Key Exchange

The Internet Key Exchange (IKE) [Harkins and Carrel 1998] dynamically establishes security associations (SA) for the IPsec [Kent and Atkinson 1998] suite of protocols. The IKE phase one exchange negotiates an IKE SA for securing IPsec SA negotiation and key agreement. Policy is negotiated through a round of policy proposals defining the algorithms and means of authentication protecting the IKE SA.

Figure 5 depicts Ismene policies whose reconciliation models an IKE phase one policy negotiation. The session policy (IKE policy of the responder) and domain policy (IKE policy proposal) are reconciled to arrive at the SA policy. Similar to IPsec selectors, the `selector` condition in the example identifies where the identified policy is relevant. Hence, by creating similar policies with different selectors, it is possible to construct policies for all IPsec traffic supported by a particular host or network; a provision clause and associated selector is created for each class of traffic that requires IKE SA negotiation.

As in IKE negotiation, the reconciliation algorithm intersects the policy proposals resulting in the provisioning of `ike(cast-cbc,sha1,group2)` and `preshare` mechanisms. The reconciliation of the action clauses results in a single `auth` (peer authentication) clause. Note that the `config` condition in the Kerberos `auth` clause is statically evaluated; Kerberos is not configured in the instance, so the clause can never be satisfied. In this case, the clause is removed during reconciliation. The `preshare` action clause (which simply tests whether the peer has proved knowledge of the pre-shared key) is identical in both policies, and thus reconciles to a single condition clause.

6.2 Dynamic Cryptographic Context Management

Designed for policy negotiation in multi-party communication, the Dynamic Cryptographic Context Management (DCCM) [Dinsmore et al. 2000] system defines a protocol used to negotiate a group session policy. The abstract Cryptographic Context Negotiation Template (CCNT) defines a provisioning policy structure from which the session policy is negotiated [Balenson et al. 1999]. Each CCNT structure is defined as an n-dimensional space of independent services. To simplify, a ses-

GAA-API Printer Policy

Token	Authority	Value
USER	Kv5	joe@acme.edu
rights	manager	submit_job
time_window	PST	6am-8pm
printer_load	lpd	20%

Token	Authority	Value
GROUP	Kv5	operator@acme.edu
rights	manager	submit_job

Ismene Printer Policy

```
submit_job :
  Credential(&tkt,srvr=Kv5,id=joe@acme.edu),
  timeWindow(6am,8pm,pst),
  printerLoad($lp,lpd,20%) :: accept;
submit_job :
  Credential(&tkt,srvr=Kv5,id=operator@acme.edu)
  :: accept;
submit_job :
  Credential(&tkt1,srvr=Kv5,id=joe@acme.edu),
  Credential(&tkt2,srvr=Kv5,id=$id),
  Credential(&del,id=$tkt2.id,
             grantor=$tkt1.id,rghts=submit_job),
  timeWindow(6am,8pm,pst),
  printerLoad($lp,lpd,20%) :: accept;
```

Fig. 7. GAA-API Policy - GAA-API defines session-independent authorization policies through extended ACL tokens. The semantics of tokens are realized in Ismene through structured action clause conditions.

sion policy is constructed by intersecting the points on each dimension satisfying member policy proposals. DCCM does not specify authorization policy.

The creation of session policy DCCM is operationally similar to that of IKE; policy is calculated from the intersection of known policy structures. However, where no such intersection exists, an undefined algorithm is used to identify which proposals to reconcile. The extended (prioritized) reconciliation algorithm provides guidance; important member policies are considered first, and others afterward. However, defining a total ordering to the policies frequently requires human intervention.

Ismene session and domain policies modeling the semantics of DCCM policy creation within an example CCNT (from [Dinsmore et al. 2000]) is depicted in Figure 6. The session policy defines the template CCNT, and domain policies represent policy proposals submitted by expected group members (domain policies). Ismene reconciliation finds the intersection of policies associated with the three essential mechanisms securing the group; confidentiality (`conf`), key management (`kman`), and key management transport (`trans`).

6.3 GAA-API

The Generic Authorization and Access Control API (GAA-API) provides a general-purpose framework for describing authorization in distributed systems [Ryutov and Neuman 2000]. Hence, policy in GAA-API is not session oriented, but used to continuously govern access to resources. Ismene, however, can be used to define non-session policy. Reconciliation and compliance approaches enable administratively disconnected communities to share resources while maintaining the integrity of independent authorization policies.

GAA-API policies, called extended ACLs (EACL), consist of tokens describing the authorization, rights, and conditions of access. Tokens are associated with resources to precisely describe to whom and under what conditions access is granted. Access is allowed where conditions are satisfied and credentials matching the policy statements are found. For example, Figure 7 describes equivalent GAA-API and Ismene authorization policies associated with `acme.edu`'s printers. These policies

KeyNote Local Policy

```
Authorizer: POLICY
Licensees: ADMIN_KEY
Conditions: app_domain == 'IPsec policy'
  && ( esp_enc_alg = '3des' ||
       esp_enc_alg = 'aes' ||
       esp_enc_alg = 'cast' )
  && ( esp_auth_alg = 'hmac-sha' |
       esp_auth_alg = 'hmac-md5' )
```

Ismene Session Policy

```
ADMIN_KEY := < 0xba34... >;
provision : ::
  pick( config(esp_enc_alg(3des)),
        config(esp_enc_alg(aes)),
        config(esp_enc_alg(cast)) ),
  pick( config(esp_auth_alg(hmac-sha)),
        config(esp_auth_alg(hmac-md5)) );
accept_policy :
        Credential(&pol, pol.issuer=$ADMIN_KEY)
              :: accept;
```

KeyNote IPsec Credential

```
Authorizer: ADMIN_KEY
Licensees: Bob
Conditions: app_domain == 'IPsec policy'
  && ( esp_enc_alg = '3des' ||
       esp_enc_alg = 'cast' )
  &&  esp_auth_alg = 'hmac-sha' ;
```

Ismene Domain Policy

```
signer := < 0xba34... >;
signature := < 0x98cc... >;
id := < Bob >;
provision : ::  pick( config(esp_enc_alg(3des)),
                      config(esp_enc_alg(cast)) ),
                config(esp_auth_alg(hmac-sha));
```

Fig. 8. KeyNote Policy - KeyNote credentials are only consulted where they have been explicitly delegated authority by a local policy. Conversely, Ismene regulates the acceptance of policy through the proper assignment of `accept_policy` conditions.

state that the user `joe` (authenticated by the local Kerberos service) should be allowed to submit print jobs only between 6am and 8pm and when the printer is not loaded. Moreover, the policy states that an operator can always submit a print-job.

The example delegation policy in Figure 7 demonstrates a fundamental difference between GAA-API and Ismene. While GAA-API implicitly permits delegation, Ismene requires the issuer to state a policy allowing it. The Ismene policy states that `joe` is allowed to delegate (through a delegation credential to the identity *id* for which the requestor has a usable Kerberos ticket) the `submit_job` right to any entity authenticated by the same Kerberos service. Moreover, the clause states that conditions under which `joe` is allowed access are explicitly imposed on any such delegation. For brevity, we omit the operator's right to delegate job submission.

6.4 KeyNote

Central to KeyNote trust management system is the notion of credentials [Blaze et al. 1999; Blaze et al. 1999]. A credential is a structured policy describing conditional delegation; an authority (authorizer) states that a principal (licensee) has the right to perform some action under a set of conditions. An action is allowed if a delegation chain can be constructed from a credential matching the requested action to a trusted local policy. Users supply credentials as is needed to gain access. Hence, KeyNote significantly eases the burden of policy management by allowing policy to be distributed to users, rather than configured at all policy enforcement points. The KeyNote policy depicted in Figure 8 delegates decisions about IPsec policy to the `ADMIN_KEY`, and restricts the provisioning to a range of cryptographic algorithms. The `ADMIN_KEY` credential encapsulates a policy that the user Bob (who is identified by a key) should be allowed access if IPsec is configured with the 3-DES or CAST encryption algorithms and SHA-1 HMACs are used for message

authentication.

The Ismene policies state a similar requirement, while also providing a reconciliation algorithm for generating an acceptable policy instance to provision the session. However, one facet of KeyNote not captured in Ismene is the explicit delegation of policy; KeyNote credentials are only consulted where they have been explicitly delegated authority by a local policy. In contrast, Ismene does not make any assumptions about the origin and authentication of policy but focuses on the construction of session policy. More specifically, Ismene does not provide an administrative model for issuance and revocation or directly authenticate the policy statements, e.g., via digitial signature. In the example, the KeyNote delegation approach is partially modeled in the Ismene policies. The session policy is consulted for the `accept_policy` action prior to the acceptance of any domain policy and accepted where signed by `ADMIN_KEY`. In this case, Ismene enforces policy through reconciliation; only instances consistent with the KeyNote conditions can result from reconciliation.

7. IMPLEMENTING ISMENE

The *Ismene Applications Programming Interface* (IAPI) defines interfaces for the creation, parsing, reconciliation, and analysis of Ismene policies.[9] The Ismene policy compiler, `ipcc`, validates the syntax of session and domain policies and implements the algorithms presented in Section 3. We have further integrated IAPI with the Antigone communication system [McDaniel et al. 1999], and used it as the basis for several non-trivial diverse group applications [McDaniel et al. 2001]. These include a group white-board, file-system mirror, and reliable group services. Our experience indicates Ismene is sufficiently powerful to capture a wide range of application-specific policies. The investigation suggests areas of further study:

Performance - The enforcement of fine-grained access control can negatively affect performance. For example, one file-system mirroring policy requires the evaluation of `send` action clauses prior to each packet transmission. Such evaluation slowed file transfers. We noted that because action clause evaluation was often invariant, results could be cached. We present the design of a policy evaluation cache and a comprehensive study of enforcement performance in [McDaniel 2001]. Caching significantly mitigated the cost of policy enforcement.

Authorization Reconciliation - As authorization policies defined by an instance are constructed from the conjunction of the session and domain policies, clauses can become restrictive. For example, consider the case where the session policy requires, for some action, the presentation of an X.509 certificate, and a domain policy require the presentation of a Kerberos ticket. In this case, the resulting instance requires that both a certificate and a ticket be presented. We are currently investigating ways in which overly-restrictive or unsatisfiable authorization policies can be detected at reconciliation time or at run-time.

Policy Dependencies - The effectiveness of analysis is predicated on the correct construction of policy assertions. In practice, mechanisms and configurations have

[9] All source code and documentation for the Ismene language, the augmented Antigone communication system, and applications are freely available from `http://antigone.eecs.umich.edu/`.

complex relationships. Assertion construction requires a comprehensive knowledge of the use of the cryptographic algorithms, protocols, and services. This knowledge must be reflected in the policy construction. This situation is not unique to Ismene; any policy infrastructure must ensure that unsafe policies are not allowed.

8. CONCLUSIONS

Networks are becoming more open and heterogeneous. This stands in stark contrast to the singular nature of contemporary security infrastructures; communication participants have limited ability to affect session policy. Hence, the participant security requirements are only addressed where they are anticipated by policy issuers. Ismene, and works similar to it, seek to expand the definition and usage of policy such that run-time policy is the result of the requirements evaluation, rather than dictated by the policy issuers.

In this paper, we have presented a model and language for the specification and reconciliation of security policies. Policy in our model defines interdependent statements of provisioning (session configuration) and authorization. We show that the general problem of provisioning policy reconciliation is intractable. By restricting the language, we show that reconciliation of two policies becomes tractable. However, reconciliation of three or more policies under this restriction remains intractable. We identify heuristics that detect intractability in n-party provisioning policy reconciliation. Such heuristics prioritize policies and perform pair-wise reconciliation to achieve efficiency.

We have demonstrated that like provisioning reconciliation, authorization policy reconciliation is tractable for two policies, but intractable for three or more. However, every reasonable authorization policy we have encountered could be efficiently, and often trivially, reconciled. Hence, we have not sought heuristic algorithms for authorization reconciliation.

A compliance algorithm determines whether a policy instance is consistent with a participant's domain policy. The analysis algorithm determines whether the provisioning of a session adheres to a set of assertions that express correctness constraints on a policy instance. We identify efficient algorithms for both compliance and analysis. We demonstrate that the more general problem of determining if any instance generated from a policy can violate a set of correctness assertions is intractable.

Based on the model, we presented an overview of the Ismene policy language and demonstrated its expressiveness and limitations through the representation of policies defined in several policy languages. The language has been implemented and is being used in several non-trivial applications. This and similar investigations of applications' use of policy will help illuminate how we can reconcile and meet the often divergent requirements of user communities.

9. ACKNOWLEDGMENTS

We would like to thank Peter Honeyman for his many contributions to this work. We would also like to thank Martin Strauss, Avi Rubin, Sugih Jamin, Trent Jaeger, Paul Resnick, Dave Johnson, Boniface Hicks, Luke St. Clair, Patrick Traynor and the anonymous reviewers for their many thoughtful and substantive comments, Jim Irrer for his help in maintaining the policy compiler, and to Megan McDaniel for

her many editorial comments.

REFERENCES

Balenson, D., Branstad, D., Dinsmore, P., Heyman, M., and Scace, C. 1999. Cryptographic Context Negotiation Template. Tech. Rep. TISR #07452-2, TIS Labs at Network Associates, Inc. February.

Bartal, Y., Mayer, A. J., Nissim, K., and Wool, A. 1999. Firmato: A Novel Firewall Management Toolkit. In *IEEE Symposium on Security and Privacy*. 17–31.

Bellovin, S. 1999. Distributed Firewalls. *;login:*, 39–47.

Bhatti, N. T., Hiltunen, M. A., Schlichting, R. D., and Chiu, W. 1998. Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems 16,* 4 (November), 321–366.

Blaze, M., Feigenbaum, J., Ioannidis, J., and Keromytis, A. 1999. The Role of Trust Management in Distributed Systems Security. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*. Vol. 1603. Springer-Verlag Lecture Notes in Computer Science State-of-the-Art series, 185–210. New York, NY.

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

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

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

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

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

Chu, Y., Feigenbaum, J., LaMacchia, B., Resnick, P., and Strauss, M. 1998. REFEREE: Trust Management for Web Applications. In *Proceedings of Financial Cryptography '98*. Vol. 1465. Anguilla, British West Indies, 254–274.

Cook, S. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of 3th Annual ACM Symposium on Theorey of Computing*. ACM, 151–158.

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

Dinsmore, P., Balenson, D., Heyman, M., Kruus, P., Scace, C., and Sherman, A. 2000. 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)*. DARPA, 64–73. Hilton Head, S.C.

Durham, D., Boyle, J., Cohen, R., Herzog, S., Rajan, R., and Sastry, A. 2000. RFC 2748, The COPS (Common Open Policy Service) Protocol. *Internet Engineering Task Force*.

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

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

Harkins, D. and Carrel, D. 1998. The Internet Key Exchange. *Internet Engineering Task Force*. RFC 2409.

Hiltunen, M. 1998. Configuration Management for Highly-Customizable Software. *IEE Proceedings: Software 145,* 5, 180–188.

Hiltunen, M., Jaiprakash, S., Schlichting, R., and Ugarte, C. 2000. Fine-Grain Configurability for Secure Communication. Tech. Rep. TR00-05, Department of Computer Science, University of Arizona. June.

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

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

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

Karpinski, M. and Wagner, K. W. 1988. The Computational Complexity of Graph Algorithms with Succinct Representations. *Zeitschrift für Operations Research 3,* 32, 201–211.

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

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

Li, N., Mitchell, J. C., , and Winsborough, W. H. 2005. Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM*. *To appear*.

Liu, X., Kreitz, C., van Renesse, R., Hickey, J., Hayden, M., Birman, K., and Constable, R. *1999. Building Reliable High-Performance Communication Systems from Components. In* Proceedings of 17th ACM Symposium on Operating Systems Principles (SOSP'99). *Vol. 33. ACM, 80–92.*

McDaniel, P. *2001. Policy Management in Secure Group Communication. Ph.D. thesis, University of Michigan, Ann Arbor, MI.*

McDaniel, P. *2003. On Context in Authorization Policy. Tech. Rep. TD-5JCJCK, AT&T Labs - Research, Florham Park, NJ. January.*

McDaniel, P. and Prakash, A. *2002. An Architecture for Security Policy Enforcement. Tech. Rep. TD-5C6JFV, AT&T Labs - Research, Florham Park, NJ. July.*

McDaniel, P. and Prakash, A. *2005. Security policy enforcement in the antigone system.* Journal of Computer Security. *Accepted for publication. Draft.*

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

McDaniel, P., Prakash, A., Irrer, J., Mittal, S., and Thuang, T.-C. *2001. Flexibly Constructing Secure Groups in Antigone 2.0. In* Proceedings of DARPA Information Survivability Conference and Exposition II. *IEEE Computer Society Press, 55–67. Los Angeles, CA.*

Mendelson, E. *1997.* Introduction to Mathematical Logic. *Chapman & Hall, London.*

Moriconi, M., Qian, X., Riemenschneider, R. A., and Gong, L. *1997. Secure Software Architectures. In* Proceedings of the 1997 IEEE Symposium on Security and Privacy. *84–93.*

Neuman, B. C. and Ts'o, T. *1994. Kerberos: An Authentication Service for Computer Networks.* IEEE Communications 32, *9 (Sept.), 33–38.*

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

Orman, H., O'Malley, S., Schroeppel, R., and Schwartz, D. *1994. 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.

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

Schaefer, T. J. *1978. The Complexity of Satisfiability Problems. In* Proceedings of 10th Annual ACM Symposium on Theorey of Computers. *ACM, 216–226. New York, New York.*

Schmidt, D., Fox, D., and Sudya, T. *1993. Adaptive: A Dynmaically Assembled Protocol Transformation, Integration, and eValuation Environment.* Journal of Concurrency: Practice and Experience 5, *4 (June), 269–286.*

Wallner, D. M., Harder, E. J., and Agee, R. C. *1999. Key Management for Multicast: Issues and Architectures.* Internet Engineering Task Force. *RFC 2627.*

Wang, H., Jha, S., McDaniel, P., and Livny, M. *2004. Security policy reconciliation in distributed computing environments. In* Proceedings of 5th International Workshop on Policies for Distributed Systems and Networks (Policy 2004). *IEEE Computer Society Press, 137–146. Yorktown Heights, NY.*

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

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

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

February 2003