Security Policy Reconciliation in Distributed Computing Environments

Hao Wang, Somesh Jha, Miron Livny
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

{hbwang,jha,miron}@cs.wisc.edu

Patrick D. McDaniel
AT&T Labs-Research
Shannon Laboratory
180 Park Ave., Rm. A203
Florham Park, NJ 07932
pdmcdan@research.att.com

Abstract

A major hurdle in sharing resources between organizations is heterogeneity. Therefore, in order for two organizations to collaborate their policies have to be resolved. The process of resolving different policies is known as policy reconciliation, which in general is an intractable problem. This paper addresses policy reconciliation in the context of security. We present a formal framework and hierarchical representation for security policies. Our hierarchical representation exposes the structure of the policies and leads to an efficient reconciliation algorithm. We also demonstrate that agent preferences for security mechanisms can be readily incorporated into our framework. We have implemented our reconciliation algorithm in a library called the Policy Reconciliation Engine or PRE. In order to test the implementation and measure the overhead of our reconciliation algorithm, we have integrated PRE into a distributed high-throughput system called Condor.

Keywords: Security policy, reconciliation, and high-throughput distributed system.

1 Introduction

Security policy bridges the gap between static implementations and the broad and diverse security requirements of user communities. Security policy becomes more complicated in heterogeneous environments. When two or more entities share a security association, they must reach agreement on a governing policy (e.g., two end-points in an IPsec session). These entities express their requirements for the association through a security policy (called a domain policy). A *reconciliation algorithm* finds a policy that is consistent with all domain policies. Where a consistent policy can be found, the association is free to proceed. Where one cannot be found, the participants must alter their requirements or abstain from participating.

In the general case, policy reconciliation is intractable [15, 23]. As a result, past investigations have largely achieved tractability by limiting the policy representation or by using heuristic algorithms [11, 24, 26, 33]. Such approaches achieve the stated goals, but fail to efficiently capture dependencies between different aspects of a policy. Moreover, these systems do not consider *preferential policy*: it is advantageous (and often necessary) for policy not only to specify what is legal and illegal, but to state what is desirable.

This work addresses the limitations of past work by developing a policy framework based on graphical policy representations. We exploit the graph representation to efficiently encode the complex dependencies inherent to contemporary policy. We formally define the representation and specify an efficient preference and dependency-respecting reconciliation algorithm. Before introducing our formalism, we present an overview of security provisioning policy and the intuition behind our framework in the following section.

1.1 Security Policy

The term *security policy* has come to mean different things to different communities. For example, access control policy defines who has access to what and under what circumstances [4, 30, 31]. Other forms of security policy specify under what conditions credentials are accepted [6], or how a firewall is configured [3]. In its broadest definition, security policy is the specification of security relevant system behavior. This paper addresses session-specific configuration of security services. More commonly known as *security provisioning policy*, these configurations define the guarantees afforded the governed environment by explicitly identifying the algorithms, parameters, and protocols used to implement security.

To illustrate the importance and ubiquity of security provisioning policy, consider an email client (e.g., Netscape Communicator, MS Outlook). A user specifies a provisioning policy every time she adds an account. For example, the connection method (e.g., IMAP over SSL) dictates exactly the set of guarantees you will receive in obtaining and viewing your mail. Note that the decision to not use any security service is still a specification of policy. The policies defined for the applications and services used in an environment prescribe the security afforded its users.

In practice, provisioning policy is more complicated than our email example would suggest. It is often important that particular organization-wide goals are realized in the many policies implemented by the environment. Lower level policies must be constructed such that they are *compliant* with organizational goals [23]. Moreover, where an operation spans organizations, the policies of each organization must be *reconciled* to form a coherent and reasonable policy.

We now introduce our graphical provisioning policy representation. A graphical policy is a series of policy operations represented by cascading circular or square nodes in a singularly rooted directed acyclic graph (DAG) (formally this structure is an and-or graph). Policy is read from the root node. Each node may be a decision (circle) or a collection (square). A decision node requires that exactly one of the sub-graphs emanating from the node be resolved, and a collection node requires all sub-graphs be resolved. All leaf nodes are added to the policy. Any configuration derived from a policy respects these two simple rules.

Figure 1 shows a graphical provisioning policy for key management used in an IPsec VPN. This policy

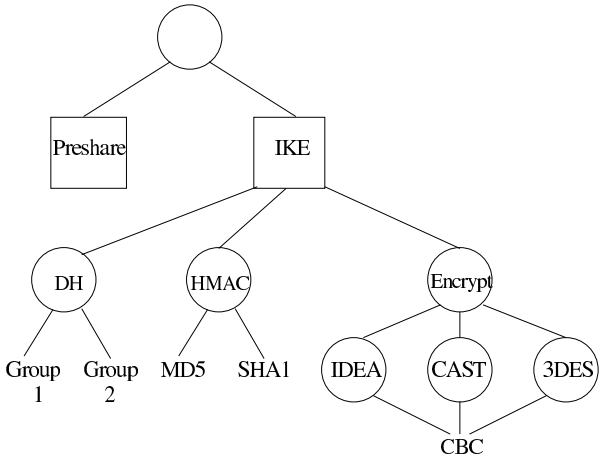


Figure 1: A graphical IPsec key management policy

would be specified by a user or network administrator as part of, for example, VPN setup. One reads the example policy's root (decision node) as:

(configure) *preshare*(ed keys) **or** *IKE*

The right hand side of the graph (IKE, from the root) depicts a complex series of configurations used to specify the behavior of the Internet Key Exchange (IKE) protocol [16]. The IKE sub-policy consists of three independent configurations. We read the top IKE (collection node) as:

(configure) *DH group* **and** *HMAC* **and** *Encrypt*ion

The remainder of the policy is read as a selection of a single DH group, a hashing algorithm, and an encryption algorithm. Independent of the encryption algorithm, a mode (e.g., CBC) must be selected. Moreover, this policy mandates the use of CBC mode.

The example policy is used at the point at which an end-point (host) is connected to the VPN. The policy is *evaluated* by identifying a subset of nodes and leaves in graph as defined by the structure of the collection and decision nodes. The IPsec implementation uses the resulting concrete specification, called an *evaluated policy* or *instance*, to implement the IPsec session. For example, one possible evaluated policy contains: *IKE*, *DH group*, *Group 2*, *HMAC*, *MD5*, *IDEA*, and *CBC*.

Two important factors are highlighted by this example. First, this is one of many possible policies for IPsec key management. Depending on the goals of the specified policy, the specifier may structure the policy in a number of different ways. For example, inasmuch as it is consistent with the IPsec implementation, the policy can allow other encryption modes (e.g., ECB) by adding an additional decision node.

The second factor of note is that unlike our email policy, this policy specifies a *range of behaviors*. That is, the policy states that there are a set of configurations that are equally acceptable. The structure of the graph directly mandates which sets of configurations should be considered acceptable. Having non-prescriptive policies allow the environment to make performance and security trade-offs at run time, and is essential to reconciling policies from different domains.

Now consider the case where there is not a single source of policy: for example, where the end-points of the VPN lie in different administrative domains. Each domain wishes to exert control over the session as specified through a *domain policy* (e.g., similar to Figure 1). Hence, the two parties must find an evaluated policy that is consistent with the domain policies supplied by both. This is performed by *reconciling* the domain policies. The session can continue only where a single governing policy can be found. If not, the domain policies are incompatible and the end-points must alter their policies or refrain from participating in

the session.

The study of provisioning policy is unlike other policy efforts in several ways. First and most obviously, provisioning policy is a planning process. Traditional authorization policy systems determine whether a particular access is legal with respect to some larger governing policy. Conversely, provisioning policy attempts to find some configuration that is consistent with a governing provisioning policy.

Provisioning policy also embodies complex dependencies. That is, decisions about particular aspects of the policy affect subsequent options. Figure 1 illustrates a very simple dependency: the decision to use IKE over pre-shared keys has enormous impact on the further development of policy. The selection of IKE leads to decisions concerning the kinds of Diffie-Hellman groups to use, what encryption algorithms are necessary, etc. However, if pre-shared keys were selected, other configuration values (e.g., Diffie-Hellman group) should and would not be considered.

Provisioning is also subject to preferential behavior. That is, there is a often a set of configurations that is most desired among several choices. Again consider Figure 1. According to the policy, either group 1 or group 2 is acceptable. In practice, we have found the vast majority of IPsec configurations use group 2. As such, we (rightly or wrongly) may decide that group 2 is best for our environment, and is thus preferred. However, for compatibility reasons, we do not wish to preclude the use of group 1. Note that preferential configurations are more than simple default values, but a partial ordering of the available options. The existence of preferences is largely ignored by previous work in this area.

As we demonstrate in the following sections, reconciliation is made more complex by the introduction/appreciation of these deeper aspects of policy. While this work aspires to provide intuitive policy representations, it must do so within the constraints of these new complex semantics. Hence, our contribution lies not only in the representation or added semantics, but in the successful marriage of the two.

1.2 Contributions

This paper addresses the aforementioned deficiencies of existing systems by modeling dependencies and preferences in a graphical policy framework. The main contributions of this paper are:

•**Graph-based provisioning policy (exposes dependencies):** We present a model that represents policies as directed acyclic graphs (DAG). This model captures dependencies between policy components within a schema. Hence, because policies adhere to the schema, it is impossible to define a correctly formed policy that is not consistent with the dependencies.

•**Efficient reconciliation:** In general, policy reconciliation is NP-complete [23]. However, a graphical representation of policies expose their structure and present a basis for an efficient reconciliation algorithm. We provide an efficient reconciliation algorithm for our graphical model. Our reconciliation algorithm is linear in the total size of the policies.

•**Preferential policy:** Participant preferences, such as a server's preferences for authentication mechanisms, can be incorporated into our model. An important problem that arises in this context, is that of resolving multiple partial orders on the same set (intuitively, these partial orders represent preferences of different participants). We provide an efficient algorithm to resolve multiple partial orders and extend the reconciliation algorithm to handle preferences.

•**Implementation and deployment:** Based on our hierarchical framework, we have implemented a reconciliation module called the *Policy Reconciliation Engine* or *PRE*, which is available for download. We have integrated PRE with Condor [21], a high-throughput scheduling system used to manage resources in a complex distributed environment. We show experimentally that the cost of reconciliation is negligible.

2 Related Work

Other policy systems. Historically, policy systems have not addressed reconciliation. For example, trust management systems, such as KeyNote [5], SPKI/SDSI [12, 13], Binder [10], and SD3 [18] are concerned with compliance checking rather than reconciliation. In trust management systems, policies, called credentials, are simply cryptographic proofs that express authorization delegation. The compliance checker algorithm searches the available credentials for an accepting delegation chain that satisfies a specific request. Credentials can state a set of provisioning requirements. An action is only allowed where the provisioning of the environment matches the credential. Such approaches are useful for managing policy in a widely deployed or loosely organized environments [7]. However, because credentials mandate provisioning, there is no opportunity to perform reconciliation. Other systems simply assume a singular entity manually performs reconciliation when issuing policy for a domain [3].

Hardness of reconciliation. While reconciliation has only recently begun to be explored, the policy community has already developed a broad characterization of the problem. Gong and Qian discovered that reconciliation of authorization policy (in their work, called policy composition) is NP-complete [15]. Similarly, the authors of Ismene found that reconciliation of general purpose provisioning policy is also NP-complete [23]. Such results do not mean that progress cannot be made, but suggests a required shift in the goals of investigation. Much of the ongoing work in reconciliation has centered on techniques that alter the environment or restrict policy to obtain efficient reconciliation. However, our paper demonstrates that by using a representation that exposes structure of the policies, the reconciliation problem becomes tractable for a large class of policies.

Other reconciliation approaches. One way to address the inherent complexity of reconciliation is by essentially "flattening" the policy representation, i.e., explicitly enumerating the various choices. For example, the IPsec Security Policy System (SPS) [33] guarantees efficient two-party reconciliation by intersecting fixed and independent sets of policy values. The DCCM system extends this approach to the multi-party environments by providing a *Chinese menu* reconciliation algorithm [1, 2, 11]. Each participant chooses values from a fixed set of policy dimensions (e.g., one from column A, two from column, B, etc . . .). The policy is reconcilable where an intersection of proposals is found for each dimension. Conflicts (where no such intersection is found) are resolved by an unspecified algorithm.

A limitation of both SPS and DCCM is that they assume that there are no dependencies between policy values. For example, in an IPsec policy, an encryption algorithm is needed when the ESP transform is selected. Therefore, to ensure that the resulting policy is enforceable, one must disallow any policy that defines the ESP transform but no encryption algorithm. In practice, these systems define policy as an enumeration of legal policy combinations, such as ESP-3DES-HMAC-SHA. Since only legal enumerations are available, no dependency can be violated. However, the number of enumeration values grows exponentially in the size of the domains, and therefore the "enumeration approach" is inherently not scalable.

Ismene policies are defined as expressions of provisioning variables [23]. The reconciliation algorithm tries to find a satisfying truth assignment for the universe of provisioning variables. Reconciliation is cast as an instance of satisfaction (over the conjunction of policy proposals). Efficiency is guaranteed by using a pair-wise satisfaction algorithm on restricted policy expressions. The iterative Ismene n-policy reconciliation algorithm is sound but not complete, i.e., some collections of reconcilable policies may be rejected. Furthermore, like SPS, the Ismene reconciliation algorithm does not consider dependencies. Dependencies are addressed in Ismene by evaluating the reconciliation result with respect to a set of "correctness rules" using an *analysis algorithm*. This approach is limited in that it occurs after the policy has been identified.[1]

[1]The authors further describe an offline analysis algorithm that determines if a policy could ever be reconciled and violate a correctness rule. Any algorithm of this sort is shown to be intractable.

Hence, reconciliation must be re-performed after each policy is rejected by analysis.

A central limitation of the approaches defined above is that they are not sensitive to the structure of policy. Dependencies between different aspects of policy are either inefficiently encoded or externally evaluated. This is a prime motivation of the current work. Dependencies are captured through the graphical structure of the policy schema, and hence any policy resulting from reconciliation is guaranteed to be consistent with these dependencies. Previous reconciliation algorithms also make no distinction between reconciliation results. Since no distinction is made, every possible result is equally desirable. However, environments often desire to specify default behavior and allow others where the defaults are inefficient or infeasible. This work allows such desires to be expressed through preferences.

Other work on representation and analysis of security policies. Cholvy and Cuppens consider the complexities of detecting and managing inconsistencies introduced by access control policy specifications [8]. Our approach differs not only in problem domain (i.e., provisioning), but in that we avoid consistency evaluation by encoding dependencies within the policy structure. Hence, collections of individual policies cannot be inconsistent. Cholvy and Cuppens further considered preference in the context of the ordered application of access control regulations, but focused on access control applications.

While it has not been explored for other forms of policy, graphical representations are well suited to access control policy [20, 25]. For example, the LaSCO language specifies access control policy using graphical idioms [17]. The developers of LaSCO assert that the representation allows not only specification a more intuitive operation, but permits the use of well known graph algorithms for subsequent enforcement. We embrace a similar approach by using structural representation to enforce dependencies.

3 A Formalization of Policy Reconciliation

In this section we provide a precise semantics of policy reconciliation where the policies are represented hierarchically. Moreover, we describe how preferences can be incorporated into our framework. Finally, we present an efficient reconciliation algorithm.

Definition 3.1 A *schemata* is a directed acyclic graph or DAG $S = (N, E, root)$, where N is a set of nodes, $E \subseteq N \times N$ is a set of edges, and $root \in N$ is a distinguished node. We assume that $root$ has no incoming edge. Each node n has the following attributes associated with it:

- Each node is a $\wedge$ or $\vee$ node.
- A tuple of variables (denoted by $Var_T(n)$) $\langle V_1 : \tau_1, \cdots, V_k : \tau_k \rangle$ (where τ_i is the type of variable V_i). Currently, we only allow types `string`, `int`, `real`, and `enum`. For an `enum` type τ_i we assume that a set of values is given, e.g., $\tau_i = \{\texttt{DES}, \texttt{3DES}, \texttt{AES}\}$.

The set of successors of a node n in a schemata S is denoted by $succ_S(n)$. However, when the schemata S is clear from the context we simply write $succ(n)$ instead of $succ_S(n)$.

A schemata is shown in Figure 2. The root node is a $\wedge$-node and represented as a square. The left and right child of the root are $\vee$-nodes and represent various authentication and encryption mechanisms respectively. The leaf nodes, such as the ones labeled with `none` and `3DES`, are $\vee$-nodes with no successors. The special keyword `none` signifies the fact that an authentication or encryption scheme is not required. Moreover, there are no variables associated with the $\vee$-nodes. However, if desired, associated attributes, such as key size for encryption schemes, can be associated with the $\vee$-nodes.

Discussion: Our hierarchical model for expressing security policies opens up the possibility of using formalisms such as XML for representing policies. In the XML parlance, schemata and policies are like XML schemas, and instances are XML documents that confirm to these schemas. We wanted to present an abstract description of policies and not rely on specific formalisms for representing hierarchical structures, such as

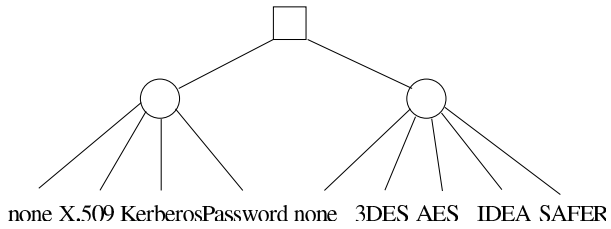


Figure 2: Schemata S

XML. However, in our implementation we use XML to represent policies. Another intriguing direction is to view schematas and policies as tree automaton, where trees can be viewed as instances [14, 32]. We also leave exploration of this analogy as a future direction.

Definition 3.2 An *instance* I of a schemata $S = (N, V, root)$ is a subgraph $(N', V', root)$, where $N' \subseteq N$ and $V' \subseteq V$. Additionally, following conditions need to be satisfied:

- For a $\wedge$-node $n \in N'$, $succ(n) \subseteq N'$. In other words, all successors of a $\wedge$-node are in the instance I.
- For a $\vee$-node $n \in N'$, if $succ(n)$ is non-empty, then $|succ(n) \cap N'| = 1$. In other words, for a $\vee$-node with a non-empty set of successors, exactly one successor is in an instance.
- Consider a node $n \in N'$ such that $Var_T(n) = \langle V_1 : \tau_1, \cdots, V_k : \tau_k \rangle$. In this case, I assigns values v_i of type τ_i to each variable V_i in $Var_T(n)$. The tuple of values assigned by I to the node n is denoted by $Val_I(n)$.

Definition 3.3 A *policy* P for a schemata $S = (N, V, root)$ is a 2-tuple (S, C), where $S : N \rightarrow 2^N$ and C maps nodes to a tuple of conditions. For each $\vee$-node $n \in N$, $S(n) \subseteq succ(n)$, and $C(n)$ is a k-tuple of conditions $\langle c_1, \cdots, c_k \rangle$ where $Var(n) = \langle V_1 : \tau_1, \cdots, V_k : \tau_k \rangle$. Moreover, we assume that the condition c_i applies to values of type τ_i. Given a value v_i of type τ_i, we use $v_i \models c_i$ to denote that v_i satisfies c_i.

Note: The syntax and semantics of the conditions depends on the types of the corresponding variables. For example, if a variable V_i has type `string`, the corresponding condition c_i can be a regular expression. For numeric types `int` and `real`, condition c_i might be range conditions $x \leq V_i \leq y$, where x and y are constants. If V_i is an enumerated type, then c_i might simply be a subset of the set of possible values for τ_i. Details of semantics and syntax of the conditions are not particularly important; we simply require that given a value v_i and condition c_i, we should be able to efficiently determine whether v_i satisfies condition c_i.

Two policies P_1 and P_2 are shown in figures 3 and 4 respectively. Consider the left child of the root. Policy P_1 specifies that only `X509`, `Kerberos`, and `Password` are allowed successors for the left node. Other edges and nodes can be interpreted in a similar manner.

Given an instance $I = (N', V', root)$ and a policy $P = (S, C)$, we say that I *satisfies* P (denoted by $I \models P$) iff the following two conditions are satisfied:

- For all $\vee$-nodes $n \in N'$, $(succ(n) \cap N') \subseteq S(n)$. In other words, instance I can only choose successors of a $\vee$-node from the subset $S(n)$ provided by the policy P.
- Let $Val_I(n) = \langle v_1, \cdots, v_k \rangle$ be the values assigned to the node n in I, and $C(n) = \langle c_1, \cdots, c_k \rangle$ be the conditions assigned to node n by the policy P. In this case, for $1 \leq i \leq k$, $v_i \models c_i$, or each value assigned in the instance I should satisfy the corresponding condition specified by the policy P.

Policy P for a schemata S is called *satisfiable* iff there exists I such that $I \models P$.

Next, we define conjunction of two policies. The conjunction of two policies $P_1 = (S_1, C_1)$ and $P_2 = (S_2, C_2)$ (denoted by $P_1 \wedge P_2$) is a policy (S', C'), where

- For each $\vee$-node $n \in N$, $S'(n) = S_1(n) \cap S_2(n)$ and $C'(n) = \langle c_1^1 \wedge c_1^2, \cdots, c_k^1 \wedge c_k^2 \rangle$, where $C_1(n) = \langle c_1^1, \cdots, c_k^1 \rangle$ and $C_2(n) = \langle c_1^2, \cdots, c_k^2 \rangle$.

Conjunction of the two example policies P_1 and P_2 is depicted in Figure 5.

Definition 3.4 A set of n policies $P_1, \cdots, P_n$ is *reconcilable* iff there exists an instance I such that $I \models (\bigwedge_{i=1}^n P_i)$ or in other words $\bigwedge_{i=1}^n P_i$ is satisfiable.

Remark: We have described the semantics of reconcilable policies using the satisfaction relation $\models$. One can give an alternative definition in terms of languages. A schemata S defines a language of instances $L(S)$, i.e., $L(S)$ contains all instances I of the schemata I. A policy P for the schemata S also defines a language of instances $L(P) \subseteq L(S)$, i.e., $L(P)$ contains all instances I such that $I \models P$. In this context, policies $P_i, \cdots, P_n$ are reconcilable iff $\bigcap_{i=1}^n L(P_i)$ is non-empty.

3.1 Resolving multiple partial orders

Later in this section we discuss policy reconciliation in presence of preferences. In preparation for that, we need to develop some theory about resolving multiple partial orders. Assume that we are given a finite set S. Suppose n agents give their preferences on the set S, i.e., agent i specifies a partial order $\preceq_i$ on the set S. Intuitively, an agent i is an organization or process with a policy, and $\preceq_i$ specifies the preference of the organization or process. The question is how does one construct a *single partial order* on the set S (denoted by $\preceq_{1,\cdots,n}$) from the n partial orders $\preceq_1, \cdots, \preceq_n$?. Precise definition for combining partial orders is given in appendix A. We also provide a a linear time algorithm to compute the combined partial order.

Example 3.1 Consider two partial orders shown in Figure 6 on the set { `Kerberos, X509, Password` }. Assuming that the agent giving the partial order (a) has higher preference than the agent with the partial order (b), the combined partial order is (b). Assuming no order between the agents the combined partial order is (a).

3.2 Reconciliation with preferences

This section describes reconciliation when policies are allowed to specify preferences. First, we define the concept of policy with preferences.

Definition 3.5 A *policy* P for a schemata $S = (N, V, root)$ is now a 3-tuple $(S, C, pref)$, where $S : N \rightarrow 2^N$, C maps nodes to a tuple of conditions, and $pref$ provides preferences. For each $\vee$-node $n \in N$, $S(n) \subseteq succ(n)$, $pref(n)$ is a partial order on $S(n)$, and $C(n)$ is a k-tuple of conditions $\langle c_1, \cdots, c_k \rangle$ where $Var(n) = \langle V_1 : \tau_1, \cdots, V_k : \tau_k \rangle$. Moreover, we assume that the condition c_i applies to values of type τ_i. Given a value v_i of type τ_i, we assume that $v_i \models c_i$.

A policy P induces a partial order $\preceq_P$ on the instances satisfying P. Given an instance I, the DAG rooted at a node n of I is called a *sub-instance*, i.e., a sub-instance consists of the node n and all of its descendants. The depth of a sub-instance is the length of the longest path from the root to one of its leaves. The partial order $\preceq_P$ is defined on sub-instances. Given two sub-instances $SI_1 = (N_1, V_1, root_1)$ and $SI_2 = (N_2, V_2, root_2)$, we say that $SI_1 \preceq_P SI_2$ iff the following conditions are satisfied:

- The roots are the same, i.e., $root_1 = root_2$.
- *$root_1$ is a $\wedge$-node.*
  Let the set of successors of $root_1$ be $\{n_1, \cdots, n_k\}$. Let I_i^1 and I_i^2 (for $1 \leq i \leq k$) be the sub-instances in SI_1 and SI_2 that are rooted at n_i. In this case the condition is that for all $1 \leq i \leq k$, $I_i^1 \preceq_P I_i^2$.

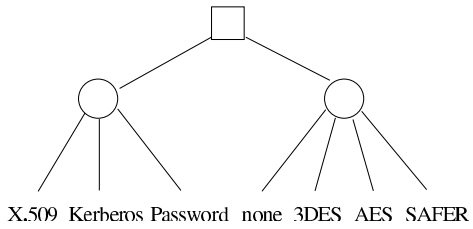


Figure 3: Example policy P_1

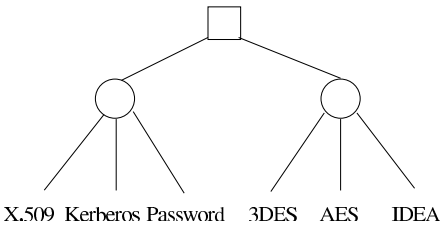


Figure 4: Example policy P_2

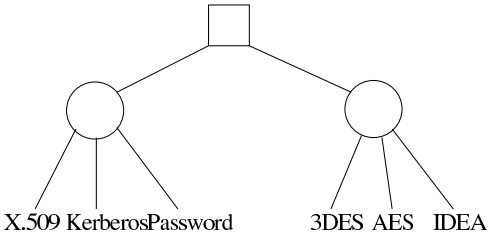


Figure 5: Combined policy $P_1 \wedge P_2$

- *$root_1$ is a $\vee$-node.*
  Let the successors of $root_1$ and $root_2$ in SI_1 and SI_2 be n_1 and n_2 respectively, and I_{n_1} and I_{n_2} be the sub-instances rooted at n_1 and n_2 respectively. In this case, the condition is the following:

  > If $n_1 = n_2$, then $I_{n_1} \preceq_P I_{n_2}$; otherwise, $n_1 \preceq n_2$ in the partial order $pref(root_1)$ given by the policy P.

Notice that $\preceq_P$ is inductively defined using the depth of the sub-instances. Intuitively, the partial order $\preceq_P$ extends the partial order $pref$ over nodes given by the policy P to sub-instances.

Next, we extend the definition of conjunction of two policies to incorporate preferences. The conjunction of two policies $P_1 = (S_1, C_1, pref_1)$ and $P_2 = (S_2, C_2, pref_2)$ (denoted by $P_1 \wedge P_2$) is a policy $(S', C', pref')$, where

> For each $\vee$-node $n \in N$, $S'(n) = S_1(n) \cap S_2(n)$, $pref'(n)$ is equal to $\preceq_{1,2}$,[2] and $C'(n) = \langle c_1^1 \wedge c_1^2, \cdots, c_k^1 \wedge c_k^2 \rangle$, where $C_1(n) = \langle c_1^1, \cdots, c_k^1 \rangle$ and $C_2(n) = \langle c_1^2, \cdots, c_k^2 \rangle$.

Given n reconcilable policies $P_1, \cdots, P_n$, an instance I is called a *most preferred instance* or *MPI* if $I \models (\bigwedge_{i=1}^n P_i)$ and I is a maximal element in the partial order induced by the combined policy $\bigwedge_{i=1}^n P_i$.

3.3 The Reconciliation Algorithm

Given n policies $P_1, P_2, \cdots, P_n$, the reconciliation algorithm proceeds as follows:

> First, we compute the combined policy $P = \bigwedge_{i=1}^n P_i$.
> Next, starting from the root the combined policy P is traversed recursively to find the most preferred instance according to partial order $\preceq_P$ induced by the combined policy.

The complexity of reconciliation algorithm is $O(n(|N| + |E|)$, where N and E are the nodes and edges in P. Details of the reconciliation algorithm can be found in appendix B.

[2]Note that before the resolving the partial orders $pref_1(n)$ and $pref_2(n)$ have to be restricted to the set $S_1(n) \cap S_2(n)$.

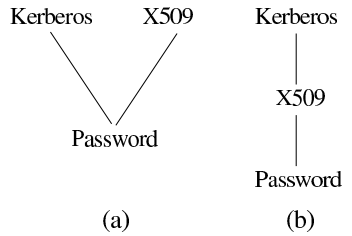


Figure 6: Two partial orders on authentication schemes.

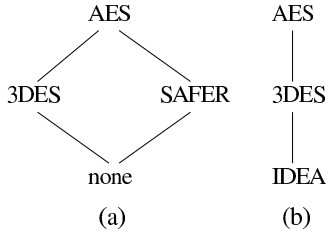


Figure 7: Two partial orders on encryption schemes.

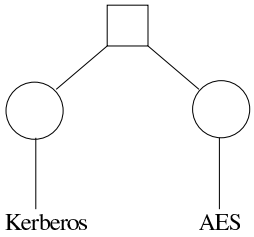


Figure 8: The most preferred instance.

Assume that we are given two policies P_1 and P_2 shown in Figures 3 and 4. The combined policy $P_1 \wedge P_2$ is shown in Figure 5. Suppose that the partial order on authentication mechanisms corresponding to policies P_1 and P_2 is as shown in Figure 6, and the partial order on the encryption schemes corresponding to the policies P_1 and P_2 is as shown in Figure 7. The partial orders are resolved so that policy P_1 has precedence over policy P_2. In this case, the partial orders on the authentication and encryption schemes in the combined policy $P_1 \wedge P_2$ is the one corresponding to policy P_2, i.e., the partial order labeled (b) in the two figures. The MPI computed by our algorithm is shown in Figure 8.

4 Applications of the policy reconciliation framework

This section illustrates the use of graphical policy in real application environments. To this end, we show how our policy reconciliation framework can augment IPsec's existing policy negotiation and support the Condor distributed computing system.

4.1 Graphical Policy in IPsec

The IPsec [19] suite of protocols provides *source authentication*, *data integrity* and *data confidentiality* at the IP layer. These services are implemented by the Authentication Header (AH) and Encapsulating Security Payload (ESP) transforms. Although not a security service, PCP implements data compression. Each IPsec node (host or security gateway) maintains a security and compression policy defined in terms of these transforms. Communicating peers establish one or more pairs of policy instances (an instance is represented as a *security association*, or SA) by reconciling configured local policies (called proposals). The Internet Key Exchange protocol (IKE) [16] is used to, among other things, negotiate this governing policy.

IKE policy can be modeled using our graphical approach. To illustrate, suppose that a host desires the following policy:

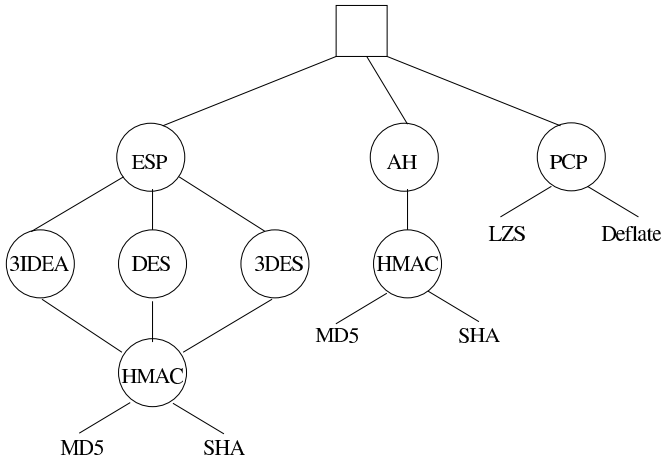
- All outgoing data must be protected by *ESP* and *AH* protocols, and must be compressed using the *PCP* protocol.
- *ESP* can use *3DES, 3IDEA* or *DES* encryption algorithms, and either *HMAC-MD5* or *HMAC-SHA* integrity/authentication algorithms.
- *AH* can use either *HMAC-MD5* or *HMAC-SHA*.
- *PCP* can use either *LZS* or *Deflate*.

One (of potentially many) schema for IPsec policy is shown in Appendix C. This schemata reflects a top-down structure, i.e., each specification is recursively fine tuned by identifying the transforms and then the algorithms.

An IPsec proposal and graphical representation (from the schemata in Appendix C) for the example policy is depicted in Figure 9. The hierarchical DAG structure is clearly more expressive and efficient, i.e.,

```
Proposal 1: AH
   Transform 1: HMAC-MD5
   Transform 2: HMAC-SHA
Proposal 1: ESP
   Transform 1: 3DES with HMAC-SHA
   Transform 2: 3DES with HMAC-MD5
   Transform 3: 3IDEA with HMAC-MD5
   Transform 4: 3IDEA with HMAC-SHA
   Transform 5: DES with HMAC-MD5
   Transform 6: DES with HMAC-SHA
Proposal 1: PCP
   Transform 1: LZS
   Transform 2: Deflate
```

a.) Original IPsec Policy Proposal



b.) IPsec Policy Schemata in DAG Format

Figure 9: IPsec Policy Example

one only needs to understand the difference between $\wedge$ (square) and $\vee$ (circle) nodes to interpret policy. Conversely, one needs a great deal of domain knowledge to interpret the proposal/transform structure of IPsec. Such intuitive representation simplifies specification, and ultimately reduces policy errors.

Consider an extension to the above policy that states that the use of 3IDEA must use either 128-bit or 256-bit keys. In IPsec, attributes such as key length can be specified only once with each transform. Hence, a separate transform is required for each key length. More generally, the number of transforms grows exponentially in the number of independent attributes. Conversely, the graphical representation only needs to introduce a single subgraph that is shared by the relevant nodes.

4.2 Hierarchically Policy in the Condor system

The second example of the policy reconciliation framework is used in the context of Condor [9], a distributed high-throughput system designed to efficiently schedule the usage of distributed and heterogeneous resources such as idling CPUs and unused memory. Condor allows resources owners to place various policy requirements on the use of their resources. Our hierarchical DAG structure can succinctly encode Condor security policies. Details of the encoding are very similar to the one discussed in the previous subsection. Appendix D provides details about Condor security policies and their encoding in our framework. The design of the policy infrastructure and its integration with Condor are detailed in the following section, where we discuss how we integrate the policy reconciliation engine into the Condor system.

5 Implementation

5.1 Policy Reconciliation Engine

We have implemented our hierarchical reconciliation algorithm in the *Policy Reconciliation Engine (PRE)*. PRE reconciles (only) pairs of XML-encoded policies. The restriction of PRE to two-policy reconciliation is not a limitation of our approach, but rather an artifact of the initial target systems' point-to-point communication models (IPSec and Condor). We will extend the implementation to allow multi-party policy reconciliation (e.g., Ismene [23], DCCM [11]) as future needs dictate.

As shown in Figure 10, PRE implements an asymmetric requester/responder model. In this model, the requester supplies the relevant policy to the responder. The responder reconciles the received policy with

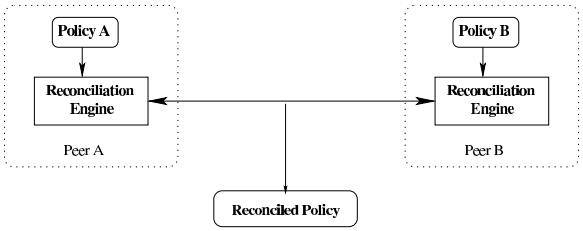


Figure 10: Architecture of Policy Reconciliation Engine (PRE)

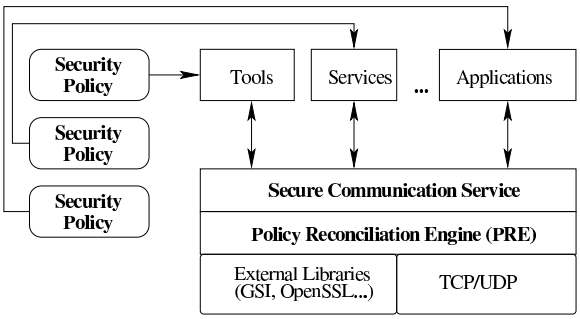


Figure 11: Integrating PRE with Condor

local policies as needed, and the *reconciled policy* is returned to the requester. Both parties subsequently use the reconciled policy to control the session. We chose a requester/responder model because it most faithfully represents contemporary use of policy (e.g., IKE policy negotiation [16]). This model is similar to client/server communication models. Responders, acting as servers, govern access to the communication resources and requesters, acting as clients, submit requests for those resources. In PRE, the responders assert authority over the resources by placing a higher preference on their own (local) policy. Note that the requester may (and often should) validate that the received reconciled policy is consistent with the originally proposed policy. Policy validation interfaces are provided by PRE.

PRE is both a library and a command line tool. Hence, it can be directly integrated into the source code of an application, or used as an external policy processor. The three components of PRE implement its main features: parsing, reconciliation, and validation. The *Parsing Engine* parses the security policy into an internal representation and is used as a preprocessor to the other features. The *Reconciliation Engine* reconciles pairs of hierarchical policies using the algorithms defined in section 3.3. Finally, the *Verification Engine* verifies the correctness of the *Reconciled Policy* with respect to the local security policy (i.e., implements the consistency test described above).

The current implementation of PRE contains about 1000 lines of C/C++ code. All XML processing methods use the Apache Project's `Xerces-C++` [27] Version 2.2.0 library. Internally, each policy is stored as a `DOM` object, and is processed using the standard Xerces DOM API. Source code and documentation for PRE are available for download.

5.2 Integrating Policy Reconciliation Engine with Condor

Much of our work in policy has been motivated by the requirements of the Condor system. As described in Section 4.2, Condor schedules resources based on the client requests and other environmental factors. Every Condor peer has a local security policy that governs the services providing the authenticity, confidentiality, and integrity of the session it supports. We have modified the Condor system to use PRE-based reconciliation to construct the security policy used by each session.

Past versions of Condor defined security policy using flat structures called *ClassAd*s [29]. ClassAds flexibly communicate resource advertisements and client requests. However, we found the structure of ClassAds inherently limiting, i.e., we could not represent the appropriate range of acceptable or preferential policies because of their flat structure. Such statements of policy are, as previously argued, hierarchical in nature. This need for hierarchical policy drove our efforts, and ultimately lead to the development of PRE.

The architecture of integrated PRE with Condor is shown in Figure 11. PRE sits below Condor's Secure Communication Service layer, and is used during session initiation. An in-band PRE protocol is used during the initial session handshake to determine the policy. In this protocol, the client submits the XML policy

and awaits the session defining response. After performing reconciliation, the server returns the reconciled policy to the client. Appendix E gives the XML Document Type Definition (DTD) for the Condor policy.

Currently, Condor does not authenticate the policies or policy exchanges beyond that supported by the underlying transport layer. In general, how and by whom policies are issued and authenticated is an environmental and systems design issue. Environments often require external services for storing and validation of issued policies (e.g., LDAP collections of signed policies). These issues are defined by the larger policy architecture, and is beyond the scope of the current work. Interested readers are referred to [22] for a taxonomy of policy architectures addressing these issues.

5.3 Performance

Because of the relatively small policy size and the restriction to pairwise reconciliation, we did not anticipate the introduction of PRE into Condor would significantly impact performance. We sought to measure these costs through several controlled experiments. These experiments measured the total execution time of the policy negotiation protocol defined in the preceding section. All experiments were executed in an environment consisting of a single Central Manager *server* (333 Mhz duo-processor/Linux RedHat 7.2) and eight *clients* (three Ultra 10 Sparc Sun/Solaris 2.8 and five 750 MHz Pentium III/Linux Redhat 7.2).

The experimental results confirmed our intuition: the average protocol execution (without I/O), for a policy consisting of *authentication*, *integrity* and *secrecy*, only uses about 5.2% of the total execution time. When including I/O overhead, the cost is still small–at about 10% of the overall execution time. Startup cost (i.e. program initialization) is the most dominant factor of the overall execution time, followed closely by overhead incurred from Condor's internal data structures.

5.4 Future work

While the theoretical framework and implementation of our hierarchical policy model have reached maturity, we see further exploration of its application to a wide range of problem domains as essential. Initially, we will seek to integrate PRE with widely used policy systems. This will enable us to explore the ways of exploiting the PRE services in specific and policy reconciliation in general. One such work will realize our IPsec policy in software. Integration with tools such as FreeSwan [28] will provide important data-points in the use of extended policy services, and serve to further demonstrate the power of our approach.

We also seek to apply our work to domains which have immediate, but as yet unaddressed, requirements for policy. For example, reconciliation may play an important role in defining security for peer-to-peer (P2P) systems. Currently, there are few coherent security models for P2P. The egalitarian nature of P2P systems mandate autonomy. Each end-point must be able to assert and realize a set of security requirements deemed important. However, autonomy must be counter-balanced with interoperability. The collection of participants must be able to negotiate a shared view of security. This is precisely the definition of reconciliation. Hence, we claim that the fluid and heterogeneous security models of P2P systems would be well served by our work. Moreover, the clarity and succinctness of the hierarchical model may enable more free and open use of security policy in these large communities.

This paper has discussed reconciliation only in the context of security policy. However, hierarchical policy models are applicable to other problem domains. To illustrate, GRID systems share the resources in heterogeneous environments. Participants in the GRID have diverse policies that govern the resource usage. Agreement is often achieved statically in current GRID systems by mandating the adoption of a single universal policy. This mandate is in direct conflict with the needs of dynamic environments whose resource constraints and requirements change frequently. Hence, policy reconciliation systems such as PRE can help to bridge such a gap between dynamicity and the needs for agreement. Furthermore, there is often a direct dependence between resource requirements and security settings and dynamic policy reconciliation

can act as the agent between the two. For example, a system that handles sensitive data on remote hosts will require some minimum security policy be enforced.

6 Conclusion

Security policy reconciliation is the process of resolving different security policies. In this paper, we presented a formal framework for policy reconciliation. We also presented an efficient algorithm for reconciling different policies. Two distinguishing features of our work are hierarchical representation and preferences. We also implemented a simplified version of our algorithm in a software module called PRE and incorporated it in Condor. Experimental results in the context of Condor clearly demonstrate that for each session the reconciliation overhead is negligible.

References

[1] D. Balenson, D. Branstad, P. Dinsmore, M. Heyman, and C. Scace. Cryptographic Context Negotiation Protocol. Technical report, Network Associates, Inc., 1999.

[2] D. Balenson, D. Branstad, D. McGrew, J. Turner, and M. Heyman. Cryptographic Context Negotiation Template. Technical report, Network Associates, Inc., 1999.

[3] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A novel firewall management toolkit. In *IEEE Symposium on Security and Privacy*, pages 17–31, 1999.

[4] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corperation, Bedford, MA, 1973.

[5] M. Blaze, J. Feigenbaum, and A.D. Keromytis. KeyNote: Trust management for public-key infrastructures. *Lec. Notes in Comp. Sci.*, 1550:59–63, 1999.

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

[7] Matt Blaze, John Ioannidis, and Angelos D. Keromytis. Trust management for IPsec. *Information and System Security*, 5(2):95–118, 2002.

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

[9] Condor. *http://www.cs.wisc.edu/condor/*.

[10] J. DeTreville. Binder, a logic-based security language. In *Symp. on Res. in Sec. and Privacy*, Oakland, CA, May 2002. IEEE Computer Society Press.

[11] P. Dinsmore, D. Balenson, M. Heyman, P. Kruus, C. Scace, and A. Sherman. Policy-Based Security Management for Large Dynamic Groups: An Overview of the DCCM Project. In *DARPA Information Survivability Conference and Exposition*, pages 64–73, 2000.

[12] C.M. Ellison. SPKI requirements. RFC 2692, September 1999.

[13] C.M. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B.M. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, September 1999.

[14] F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

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

[16] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). *http://www.ietf.org/rfc/rfc2409.txt*, 1998.

[17] James Hoagland, Raju Pandey, and Karl Levitt. Security Policy Specification Using a Graphical Approach. Technical Report CSE-98-3, The University of California, Davis Department of Computer Science, June 1998.

[18] Trevor Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[19] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. *http://www.ietf.org/rfc/rfc2401.txt*, 1998.

[20] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A Graph-Based Formalism for RBAC. *Transactions on Information and System Security (TISSEC)*, 5(3):332 – 365, 2002.

[21] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)*, pages 104–111, 1988.

[22] P. McDaniel. *Policy Management in Secure Group Communication*. PhD thesis, University of Michigan, Ann Arbor, MI, August 2001.

[23] P. McDaniel and A. Prakash. Methods and limitations of security policy reconciliation. In *2002 IEEE Symposium on Security and Privacy*, pages 73–87, May 2002.

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

[25] Matunda Nyanchama and Sylvia Osborn. The Role Graph Model and Conflict of Interest. *Transactions on Information and System Security (TISSEC)*, 2(1):3 – 33, 1999.

[26] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A Community Authorization Service for Group Collaboration. In *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, 2001.

[27] The Apache XML Project. Xerces C++ Parser. *http://xml.apache.org/xerces-c*.

[28] The FreeS/WAN Project. Linux FreeS/WAN. *http://www.freeswan.org/*.

[29] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1998.

[30] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[31] Ravi S. Sandhu and Pierrangela Samarati. Access Control: Principles and Practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[32] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 135–186. Elsevier, 1990.

[33] J. Zao, L. Sanchez, M. Condell, C. Lynn, M. Fredette, P. Helinek, P. Krishnan, A. Jackson, D. Mankins, M. Shepard, and S. Kent. Domain Based Internet Security Policy Management. In *DARPA Information Survivability Conference and Exposition*, pages 41–53, 2000.

A Combining Partial Orders

We consider three alternatives for combining partial orders. The first two alternatives are a special case of the last alternative, but we present them separately because these might represent the "common cases".

- **[Case I] Agents totally ordered:** In this case, we assume that there is a total order on agents, i.e., agent i is ordered before all agents j, where $j > i$. Given s and s' in the set S, we define $s \preceq_{1,\ldots,n} s'$ iff there exists a j such that $s \preceq_j s'$ and $s \not\preceq_i s'$ and $s' \not\preceq_i s$ for all $i < j$. In other words, in the "combined" partial order we say that s' is "preferred" over s if agent j prefers s' over s and agents that are ordered before agent j have no preference between s and s'.

- **[Case II] No order between agents:** In this case, we assume no order between agents. Given s and s' in the set S, we define $s \preceq_{1,\ldots,n} s'$ iff for all $1 \leq i \leq n$, $s \preceq_i s'$. In other words, in the "combined" partial order we say that s' is "preferred" over s if all agents prefer s' over s.

- **[Case III] Partial-order between agents:** This case is slightly more complicated. First, we explain the idea using an example. Suppose there are three agents 1, 2, and 3. Assume that 1 is ordered before 2 and 3, but there is no specific order between 2 and 3. In other words, agent 1's preferences should supersede those of agents 2 and 3. In this case, s' is preferred over s if one the following two conditions is satisfied:

- agent 1 prefers s' over s, or
- agent 1 states no preference between s and s', but both agents 2 and 3 prefer s' over s.

This case is formalized below.

A partial order $\preceq$ on a finite set S will be represented as a directed graph $G[\preceq] = (S, E)$, where $s_1 \succeq s_2$ iff there is a path from s_1 to s_2 in $G[\preceq]$. We call the partial order $\preceq$ a *strict partial order* iff its graph $G[\preceq]$ is acyclic.

Next we provide a a linear time algorithm to compute the combined partial order. Assume that we are given a strict partial order $\preceq$ over the finite set $\{1, 2, \cdots, n\}$. Intuitively, $i \succeq j$ means that agent i's preferences are given precedence over j's preferences. Recall that we are given a finite set S and $\preceq_i$ is a partial order on S given by agent i. We will give an "operational" semantics for the combined partial order $\preceq_{1,2,\cdots,n}$, i.e., we provide an algorithm $\mathcal{A}$ to compute $\preceq_{1,2,\cdots,n}$.

Let $G[\preceq]$ be the directed graph which gives the partial order over the set of agent indices $\{1, 2, \cdots, n\}$. Further, we assume that $G[\preceq]$ is a directed acyclic graph or DAG. Our algorithm $\mathcal{A}$ maintains two data structures: a graph G_2 with set of vertices S and a boolean array $mark[\cdot]$ with n elements. At the end, G_2 represents the combined partial order $\preceq_{1,2,\cdots,n}$.

Initially: G_2 has empty set of edges, and we set $mark[i] = 0$ for all $1 \leq i \leq n$.

Iteration: Our algorithm $\mathcal{A}$ repeats the following steps until for all $1 \leq i \leq n$, $mark[i] = 1$.

- Find all unmarked nodes I of G which do not have incoming edges from an unmarked vertex, i.e., I is defined as

  $$\{k \mid mark[k] = 0, \text{ and there does not exist } j \text{ such that } mark[j] = 0 \text{ and there is an edge from } j \text{ to } k\}$$

  Mark all nodes in I, i.e., for all $i \in I$, $mark[i] = 1$.

- Add an edge from $s \rightarrow s'$ to G_2 if the following two conditions are satisfied:
  - edge $s \rightarrow s'$ does not already exist in G_2, and
  - for all $i \in I$, $s \succeq_i s'$, i.e., all agents in I "agree" that s "precedes" s'.

Data structures and computational complexity: We assume that graphs corresponding to the various partial orders are stored in a data structure such that it takes *constant time* to decide whether $s \preceq s'$. For example, given a graph $G[\preceq]$ for the partial order $\preceq$, we can first compute the transitive closure of the graph and store reachability information in a hash table. Let $\preceq$ be the partial order over the finite set $\{1, 2, \cdots, n\}$ and $G[\preceq]$ be the associated graph. We allocate an array $mark$ with n elements and initialize all its elements to 0. We maintain a list $sourceList$ of all elements in $\{1, 2, \cdots, n\}$ that do not have an incoming edge in $G[\preceq]$ from an unmarked vertex. Moreover, we maintain an auxiliary array $inDegree[\cdot]$ of n elements, where $inDegree[i]$ is the number of incoming edges to i from unmarked vertices. The list $sourceList$ and array $inDegree[i]$ can be initialized by inspecting the graph $G[\preceq]$. The iteration step of algorithm $\mathcal{A}$ specific to the data structures are as follows:

- Add an edge from $s \rightarrow s'$ to G_2 if the following two conditions are satisfied:
  - edge $s \rightarrow s'$ does not exist in G_2, and
  - for all $i \in sourceList$, $s \succeq_i s'$, i.e., all agents in $sourceList$ "agree" that s "precedes" s'.

- Mark all elements that appear in the list $sourceList$. If an element i is marked, then for all edges $i \rightarrow j$ decrement $inDegree[j]$. If $inDegree[j]$ becomes 0, add j to a temporary list $tmpList$.

- Copy $tmpList$ to $sourceList$.

Let the size $|G|$ of a graph G be the sum of the number of vertices and edges in G. It is easy to see that the complexity of our algorithm is linear in the sum of the sizes of the graph or is

$$O(|G^{\star}[\preceq]| + \sum_{i=1}^{n} |G[\preceq_i]|) \ .$$

Note: In the complexity analysis given above, we are ignoring the cost of computing transitive closure of the graphs associated with the partial orders. Graph $G^{\star}[\preceq]$ is the transitive closure of $G[\preceq]$.

B Details of the Reconciliation Algorithm

This section presents the details of the algorithm for reconciling n policies briefly described in Section 3.3. Operations for intersecting two sets, computing the conjunction of two conditions, and resolving two partial orders are considered primitive operations, i.e., $O(1)$ operations. This makes the presentation simpler and abstracts away from specific representation issues, e.g., any polynomial time algorithm for these operations will suffice for our discussion.

First, we focus on computing the conjunction of 2 policies P_1 and P_2 for a schemata $S = (N, E, root)$. The time complexity for this operation is $O(|N| + |E|)$, i.e., for each node n the number of primitive operations is bounded by a constant and there are at most N nodes. Now consider n policies $P_1, \cdots, P_n$, where $n = 2^m$. In this case, we first compute $P_1 \wedge P_2$, $P_3 \wedge P_4$, $\cdots$, $P_{n-1} \wedge P_n$. After that, we are left with $\frac{n}{2}$ policies. We again form groups of two and perform the conjunction, which leaves us with $\frac{n}{4}$ policies. Continuing this way, we can compute $\bigwedge_{i=1}^{n} P_i$ in $\log_2(n)$ steps. The number of conjunctions we perform are $\frac{n}{2} + \frac{n}{4} + ... + 1$. So we perform $O(n)$ conjunctions and each conjunction has time complexity $O(|N| + |E|)$. Therefore, the time complexity to compute $\bigwedge_{i=1}^{n} P_i$ is $O(n(|N| + |E|))$. The general case, where n is not a power of 2, can be solved in an analogous manner with same asymptotic time complexity.

Assume that we have computed the combined policy $P = \bigwedge_{i=1}^{n} P_i$. Next we describe an algorithm $findMPI(S, P)$ to compute an MPI for policy P on a schemata $S = (N, E, root)$.

Primitives and data structures: The algorithm maintains a "cache" $Results$ of old results, i.e., if $findMPI(S', P')$ is called, then the result of the algorithm along with the arguments are stored in the cache $Results$. We assume that given a k-tuple of conditions $C = \langle c_1, \cdots, c_k \rangle$, there is a function $pick(C)$ that return a k-tuple of values $\langle v_1, \cdots, v_k \rangle$ such $v_i \models c_i$ (for $1 \leq i \leq k$).

Algorithm description: For clarity, we describe a recursive algorithm; however, a non-recursive version of the algorithm can be easily implemented using work-lists. If the schemata S is empty, then we simply return an empty instance. There are two cases based on whether $root$ is a $\vee$ or $\wedge$ node. The proof of correctness is by induction on the depth of the tree schemata and is interleaved with the description of the algorithm. The algorithm maintains the invariant that $findMPI(S, P)$ returns an MPI for policy P on schema S. The complexity of the algorithm given below is easily seen to be $O(|N| + |E|)$. Therefore, the entire reconciliation algorithm has time complexity $O(n(|N| + |E|))$.

- If $findMPI(S, P)$ is in the cache $Results$, return immediately with the result; otherwise proceed to the next two steps.

- $root$ **is an $\wedge$-node.**
  Let the set of successors $succ(root)$ of $root$ be $\{n_1, \cdots, n_k\}$, and S_i be the tree sub-schemata of

T that is rooted at n_i ($1 \leq i \leq k$). Similarly, let P_i be the sub-policy of P rooted at n_i. Let $findMPI(S_i, P_i)$ return a MPI I_i. The instance I for P is constructed by attaching $I_1, \cdots, I_k$ to the $root$. Add the result along with the arguments to the cache $Results$.
Correctness: Let I be the instance returned by the function $findMPI(S, P)$. Let I' be another instance of the schema S. We need to prove that $I \not\preceq I'$. Let I_i (I'_i) be the sub-instance of I (I') rooted at n_i (the i-th successor of the root). Using the induction hypothesis we know that $I_i \not\preceq I'_i$. From the definition of MPI it follows that $I \not\preceq I'$.

- $root$ **is a $\vee$-node.**
The algorithm picks a successor $n \in S(root)$ which is a maximal element in the partial order $pref(n)$. Let S_n and P_n be the sub-schemata and sub-policy rooted at n. By the induction hypothesis, $findMPI(S_n, P_n)$ returns an MPI I_n for P_n. In this case, $findMPI(T, P)$ returns an instance I, which is I_n attached to $root$. Add the result along with the arguments to the cache $Results$.
Correctness: Let I be the instance returned by the function $findMPI(S, P)$, and I' be another instance of the schema S. We need to prove that $I \not\preceq I'$. Assume the contrary, i.e., $I \preceq I'$. Let the successor of $root$ in I and I' be n and n'. From the definition of the partial order $\preceq_P$ there are two cases:
(*Case 1:*) In the partial order $pref(root)$ of policy P, $n \preceq n'$. However, $findMPI(.,.)$ picks a successor that is maximal in the set $S(root)$. This contradicts the fact that $n \preceq n'$.
(*Case 2:*) $n = n'$ and $I_n \preceq I_{n'}$, where I_n and $I'_{n'}$ are sub-instances of I and I' rooted at n and n' respectively. However, this contradicts the induction hypothesis for $findMPI(S_n, P_n)$.

C IPsec Graphical Policy Schemata

The following is a simplified IPsec policy schemata for the IKE example in section 4.

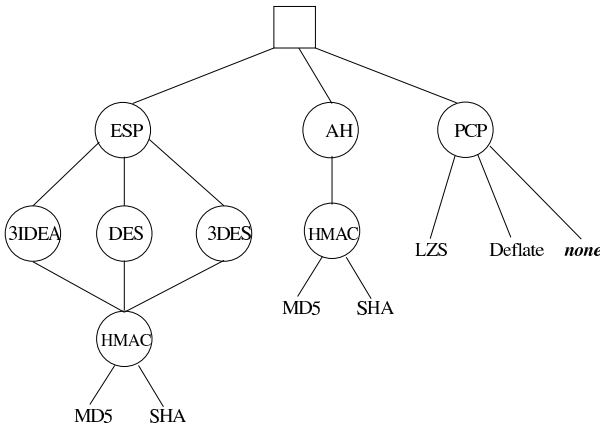


Figure 12: IPsec Policy Schemata Example

D Details of Policy in the Condor System

Figure 13 shows a two Condor security policies and their encoding in our framework. Policy 1 indicates that `authentication` is required, and acceptable *methods of authentication* are `Kerberos` and `GSS`, in that order. On the other hand, `secrecy` is only `optional`, with `3DES` and `BLOWFISH` as the acceptable algorithms. Policy 2 is similar to policy 1 except that `GSS` is the only acceptable authentication method and `secrecy` is always required. Existing Condor system uses the ClassAd structure to represent the policies, as shown by the two examples on the top of the figure. In the tree representation, the square node indicates

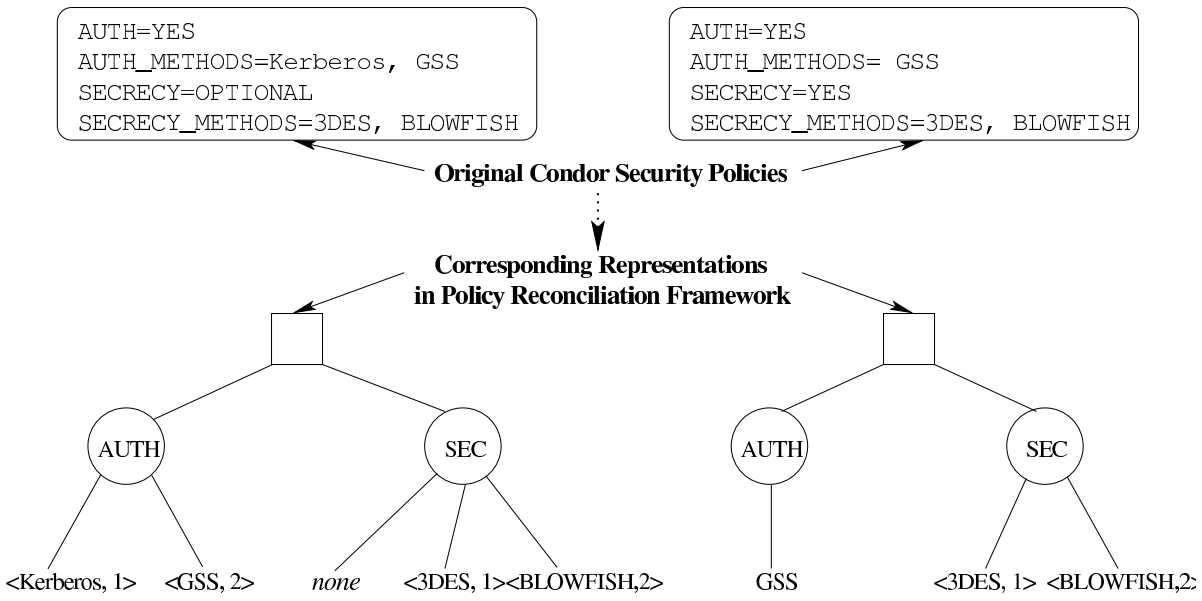


Figure 13: Condor Security Policy Example

the cases where both authentication and secrecy need to be considered for any communication. The policy on the left uses the *none* option to represent the fact that secrecy, while needs to be considered, is optional. For the policy on the right, secrecy is always required since it does not include *none* as one of its children.

In addition, the *preference* feature of the reconciliation framework is also demonstrated through the example shown in Figure 13. In each one of the Condor policies, an mechanism that appears earlier in the list has higher preference than the ones that appear later in the list. For example, in the first policy, the authentication mechanism `Kerberos` has a higher order of preference than `GSS`. This is annotated in the hierarchical policy through an integer `ID` attribute. The smaller the ID is, the higher the preference becomes. For example, in the corresponding hierarchical policy representation, `Kerberos` has ID `1`, while `BLOWFISH` has ID `2` [3]. The special keyword *none* always has the lowest preference and therefore does not have an ID associated with it. For policies that do not need the preference feature, all mechanisms are annotated with the same ID to indicate that either one of them can be selected during reconciliation.

As was the case with IPsec, our framework is applicable to other type of policies. As shown through the examples in Figure 13, the hierarchical representation can express Condor's security policies in a more concise and unambiguous manner. The hierarchical relationship among various components of the policies can be expressed clearly using the DAG structure, while the same cannot be said for the original Condor policy representation. Although here we only address how to apply the framework to Condor's security policies, we can also apply the framework to support other types of policies in Condor. For instance, resource owners in Condor can use similar policies to described acceptable users for their resources. Likewise, resource users can use policies to describe their preferred resources. The process of matching among the resources and users is called *match making* and it can be also modeled using our framework.

[3]The IDs are normalized before reconciliation in order to avoid inconsistency among different representations.

E DTD for IPSec and Condor Policies

Condor Security Policy DTD Example

```
<?xml encoding="iso-8859-1"?>
<!-- @version -->
<!ELEMENT Policy (Authentication, Integrity,
                  Confidentiality, Duration)>
<!ELEMENT Authentication (Algorithm)+>
<!ELEMENT Integrity (Algorithm)+>
<!ELEMENT Confidentiality (Algorithm)+>
<!ELEMENT Duration (#PCDATA)>
<!ATTLIST Duration unit (minute|second) #REQUIRED>
<!ELEMENT Algorithm (#PCDATA)>
<!ATTLIST Algorithm id CDATA #REQUIRED>
```

IPSec Policy DTD Example

```
<?xml encoding="iso-8859-1"?>
<!-- @version -->
<!ELEMENT Policy (Proposal)+>
<!ELEMENT Proposal (AH+, ESP+, PCP+)>
<!ATTLIST Proposal id CDATA #REQUIRED>
<!ELEMENT AH (AH_Algorithm)+>
<!ELEMENT ESP (ESP_Algorithm)+>
<!ELEMENT PCP (PCP_Algorithm)+>
<!ELEMENT AH_Algorithm (name)>
<!ATTLIST AH_Algorithm KeyLength CDATA #REQUIRED>
<!ELEMENT ESP_Algorithm (ESP_NAME)>
<!ELEMENT ESP_NAME (name, AH_Algorithm)>
<!ATTLIST ESP_Algorithm KeyLength CDATA #REQUIRED>
<!ELEMENT PCP_Algorithm (name)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT key_length (#PCDATA)>
```