# Defeasible Security Policy Composition for Web Services*

Adam J. Lee, Jodie P. Boyer, Lars E. Olson, and Carl A. Gunter†
Department of Computer Science
University of Illinois at Urbana-Champaign

## Abstract

The ability to automatically compose security policies created by multiple organizations is fundamental to the development of scalable security systems. The diversity of policies leads to conflicts and the need to resolve priorities between rules. In this paper we explore the concept of *defeasible policy composition*, wherein policies are represented in defeasible logic and composition is based on rules for non-monotonic inference. This enables policy writers to assert rules tentatively; when policies are composed the policy with the firmest position takes precedence. In addition, the structure of our policies allows for composition to occur using a single operator; this allows for entirely automated composition. We argue that this provides a practical system that can be understood by typical policy writers, analyzed rigorously by theoreticians, and efficiently automated by computers. We aim to partially validate these claims here with a formulation of defeasible policy composition for web services, an emerging foundation for B2B commerce on the World Wide Web.

## 1 Introduction

Large-scale security systems typically entail cooperation between domains with differing policies. Developing practical, sound, and automated ways to compose policies to bridge these differences is a long-standing problem. One of the key subtleties is the need to deal with inconsistencies and defaults where one organization proposes a rule on a particular feature and another has a different rule or expresses no rule. A general approach is to assign priorities to rules and observe the rules with the highest priorities when there are conflicts. However, policies with priorities may be difficult to write and understand. For instance, if a priority is represented as a numerical value between 1 and 10, when should a policy designer use a priority of 5 rather than 6? These difficulties may arise when dealing with a partner organization that has its own meanings for these priorities. This problem is related to the concept of "common sense" reasoning, in that an analyst needs to assert policies with a limited degree of commitment, just as a party may need to make a judgment of facts tentatively until greater information reveals faults.

One area in which the need for policy composition in large-scale systems is emerging involves the specification of security policies for *web services*. As a result of the efforts of standards bodies such as OASIS and the World Wide Web Consortium (W3C), SOAP-based web services [37] are becoming more prevalent on the Internet. Web services are being explored and deployed in a wide range of areas, including business-to-business purchasing, web mining [17, 2], grid computing [15], and electronic mail messaging [25]. To address security concerns, OASIS and the W3C have released standards and specifications regarding many aspects of web services security, including confidentiality [32], integrity [13], federation [23], and the establishment of security policies [30].

The WS-SecurityPolicy specification [30] was developed to allow providers of web services to specify the security requirements for the services that they deploy. For instance, a service provider may require that a request contain a particular type of cryptographic token to be used for authentication purposes, or that a certain part of the request be encrypted using a particular algorithm. While the WS-SecurityPolicy specification allows service providers the freedom to specify such policies, policies defined according to this specification lack the semantics necessary for automated composition. Due to rising interest in the formation of dynamic federations and the deployment of web services that must satisfy the security policies of multiple members of these federations, addressing this problem is a matter of increasing importance.

Three principal requirements arise in addressing the need to compose web service security policies. First, the approach must be simple and intuitive to policy designers. Second, the approach should have a formal foundation that allows careful reasoning about the correctness of algorithms and consequences of composition in boundary cases that could be exploited by attackers. Third, enforcement should be efficiently implementable on existing programming platforms and consistent with the existing

*In Formal Methods in Software Engineering (FMSE '06), Alexandria, VA, November 2006. ACM.

†Email addresses: {adamlee,jpboyer,leolson1,cgunter}@uiuc.edu

or proposed standards. These requirements will generally hold in other applications as well.

In this paper, we introduce *defeasible policy composition*. In this system, policy writers construct meta-policies describing both the policy that they wish to enforce and annotations describing their composition preferences. These annotations can indicate whether certain policy assertions are required by the policy writer or, if not, under what circumstances the policy writer is willing to compromise and allow other assertions to take precedence. Meta-policies are specified in defeasible logic, a computationally-efficient non-monotonic logic developed to model human reasoning. Because composition preferences are specified along with the policies in our system, a single composition operator can be used to combine any two meta-policies. This offers several advantages over existing policy composition proposals, which typically define several low-level composition operations and require a human to oversee the composition process by specifying which operations must be used to combine a collection of sub-policies:

- The composition rules for a given policy need only be specified once, rather than each time the policy is composed with other policies.
- Policy composition can be entirely automated. Since composition preferences are encoded in the meta-policies themselves and only one composition operation is provided, an automated composition procedure can combine any arbitrary collection of sub-policies.

We cannot *prove* that this method is the best way to compose policies[1], but defeasible logic has been used successfully in diverse applications and offers a fresh perspective on policy composition. In addition, defeasible policy composition affords us a formal framework within which to reason about the outcomes of policy compositions. To ensure that this technique can be efficiently implemented with existing platforms and standards, our approach involves compilation to policies supported by major platforms. Our study is based on Microsoft .NET Web Service Extensions version 2.0 (WSE 2.0), but other platforms, standards, and web service policy specification systems are likely to be suitable.[2]

The paper is organized as follows. In Section 2, we present a brief overview of WS-SecurityPolicy, defeasible logic, and other related work to set the context in which the policy composition problem emerges. Section 3 presents an abstract model that can be used to reason about the composition of WS-SecurityPolicies. We describe an instantiation of this model based on defeasible logic in Section 4. Section 5 steps through an extended example policy composition using our defeasible logic framework. We discuss our implementation of this logical framework in Section 6 and then examine potential uses of defeasible policy composition in other domains in Section 7. We present our conclusions in Section 8.

[1]This can be compared to a challenge like *proving* that Java is the best general-purpose programming language.

[2]We chose to use WSE 2.0 instead of WSE 3.0 because the latter does not support WS-Policy and related standards. These standards are more expressive than the policy framework that has been implemented in WSE 3.0 and therefore provide a more interesting test case for defeasible policy composition.

# 2 Background and Related Work

In this section, we set the context of the policy composition problem through an overview of the WS-SecurityPolicy specification and defeasible logic. Additionally, we discuss related work in the area of logical policy composition.

## 2.1 WS-SecurityPolicy 1.0

Consider a set of business processes communicating over the Internet via SOAP web services. These software components pass XML messages formatted in SOAP envelopes. Each envelope consists of a header which contains routing information and other meta-data, and a body which contains information about the service being requested. The contents of SOAP envelopes can be protected using an optional security header which may contain security tokens, such as X.509 certificates (public-key certificates in an ISO format) or SAML assertions (security tokens in an XML format as specified by OASIS). This security header can also contain key material or signed digests used to protect the confidentiality and integrity of the enclosed SOAP message. Additionally, the sender may encrypt or sign any or all parts of the enclosed SOAP message.

The service provider may wish to specify that requests directed to their services meet certain security requirements. For instance, they could require that a request contain a particular type token for authentication purposes or that a certain part of the request be encrypted using a specific algorithm. WS-SecurityPolicy is a web service specification developed by Microsoft, IBM, RSA, and VeriSign to give service providers the ability to specify the security requirements of their web services.

WS-SecurityPolicy is designed as an extension to WS-Policy [35], a specification used to define the characteristics, capabilities, and requirements for an XML-based web service. In addition, WS-Policy provides a way to combine several assertions using the `ExactlyOne`, `All`, or `OneOrMore` operators. With WS-SecurityPolicy, a developer can specify requirements re-

```
<wsp:Policy xml:base="http://.../policies"
            wsu:Id="P1">
 <wsp:All>
  <wsse:SecurityToken>
   <wsse:TokenType>wsse:Kerberosv5TGT</wsse:TokenType>
  </wsse:SecurityToken>
  <wsse:Integrity>
   <wsse:Algorithm Type="wsse:AlgSignature"
              URI="http://.../xmldsig#rsa-sha1" />
  </wsse:Integrity>
 </wsp:All>
</wsp:Policy>
```

Figure 1: A sample WS-SecurityPolicy document

lating to the security tokens presented to the service, and the confidentiality and integrity of incoming messages. The requirements of the service provider are written as an XML document, an example of which is shown in Figure 1. This example states that the requester must present a Kerberos version 5 ticket-granting ticket and include a RSA-signed SHA1 hash of the message.

WS-SecurityPolicy is implemented as part of the current Microsoft .NET web service extensions (WSE 2.0), which also defines the `policyDocument` XML element [22]. This is used instead of WS-PolicyAttachment [36] to tie a policy to a specific web service. Specifically, `policyDocument` is used to define policies that are applied to service requests, replies, and faults.

## 2.2 Defeasible Logic

Defeasible logic is a type of non-monotonic logic. This family of logics allows previously valid conclusions to be withdrawn in the event that new information is presented. Non-monotonic logics were developed to emulate common-sense reasoning. For example, if we know that Sam is a dog, it is reasonable to assume that he can bark. However, if we later find out that Sam is sick we may want to retract our previous assumption about Sam's ability to bark.

A defeasible theory consists of three parts: facts, rules, and a superiority relationship. Facts are indisputable statements such as $\mathit{terrier}(\mathit{Sam})$, which states that "Sam is a terrier." Defeasible logic has three types of rules:

**Strict Rules** Strict rules are rules in the classical sense, such as "terriers are dogs." Formally:

$$\mathit{terrier}(X) \rightarrow \mathit{dog}(X)$$

**Defeasible Rules** Defeasible rules are used to draw conclusions that may later be retracted. For example "dogs typically bark" is a defeasible rule which is written formally as:

$$\mathit{dog}(X) \implies \mathit{bark}(X)$$

**Defeater Rules** Defeater rules provide contrary evidence to defeasible rules. For example, "if a dog is sick, it might not be able to bark." More formally:

$$\mathit{sick}(X) \rightsquigarrow \neg \mathit{bark}(X)$$

It is important to note that defeater rules cannot be used to draw conclusions, they simply prevent conclusions. A dog being sick is not sufficient evidence to prove that the dog cannot bark, however we do not want to jump to the conclusion that it indeed can bark.

The superiority relationship is a partial ordering of the rules in the defeasible theory. For example:

$$\begin{aligned}
&\mathit{basenji}(\mathit{Jasmine})\\
&\mathit{basenji}(X) \rightarrow \mathit{dog}(X)\\
&r : \mathit{dog}(X) \implies \mathit{bark}(X)\\
&r' : \mathit{basenji}(X) \implies \neg \mathit{bark}(X)\\
&r' \succ r
\end{aligned}$$

Without the superiority relation, we would not be able to conclude either $\mathit{bark}(\mathit{Jasmine})$ or $\neg\mathit{bark}(\mathit{Jasmine})$. The superiority relationship gives precedence to the rule that Basenjis cannot bark and allows us to conclude $\neg\mathit{bark}(\mathit{Jasmine})$.

In the policy composition system presented in this paper, we have chosen to represent WS-SecurityPolicies and rules regarding their composition as defeasible logic theories. We chose to use a non-monotonic logic because human reasoning is, by nature, not monotonic [16]. Humans tend to make decisions based upon partial evidence and revise their conclusions as new facts come to light [31]. We argue that specifying security policies and their composition rules using a non-monotonic logic is more natural than using a monotonic logic, as users need not translate their inherently non-monotonic reasoning into strictly monotonic rules. Although interesting, a user study investigating this argument is outside the scope of this paper.

The system presented in this paper uses defeasible logic, rather than another non-monotonic logic, because defeasible logic has been shown to be computationally efficient [26] and a number of tools exist to automate reasoning about defeasible theories. Additionally, defeasible logic has been used to solve a number of related problems in requirements engineering and legal decision support [4], planning and learning [31], and automated negotiation during auctions and brokered sales [18]. Lastly, according to [5], defeasible logic is at least as expressive as Courteous Logic Programs, which are used in [20, 33] to define and prioritize rules, a problem closely related to security policy composition.

## 2.3 Related Work

Policy specification in defeasible logic, the use of formal methods for web services, and policy composition are three areas closely related to the work presented in this paper. In this section, we present a survey of previous contributions in these areas.

In [19], Governatori, ter Hofstede, and Oaks discuss the use of defeasible logic for a brokered sale and bargaining. Antoniou, Maher, and Billington [3] also use defeasible logic to represent administrative regulations concerning, for example, exam scheduling. Cholvy and Cuppens [12] propose the use of deontic logic, another type of non-monotonic logic, for specifying and reasoning about policies. None of these works consider policy composition.

McDougall, Alur, and Gunter [28, 29] introduced the idea of combining policies using defeasible logic. Their work focuses on the application of this idea to policies specifically designed for financial transactions on a smart payment card, and it is supported by a model that focuses on decision-making based on state generated by transaction histories. The current work instead treats policy merging for long-lived server functions and is strongly influenced by issues like hierarchy and the use of existing standardized policies. The resulting system therefore has a somewhat different underlying model despite the fact that both use defeasible logic.

There has been relatively little work thus far on the application of formal methods to reasoning about WS-SecurityPolicies. TulaFale [8] is a tool that verifies policy-based security from Web services. It has been used to prove security properties for a messaging protocol based on web services [25] and protocols for a web service architecture for collecting medical information about people in their homes [27]. Recent work [9] has demonstrated a web service programming technique that enables rigorous verification together with production quality executables. However, none of these works discuss the composition of policies.

Several other sources describe the problem of policy composition. In particular, Halpern and Weissman [21] present a mechanism for using a fragment of first-order logic to specify a security policy, which accommodates composing policies. They are able to identify when policies can be logically combined without contradiction; however, if contradictions do occur then the policies cannot be combined. Our work allows contradictions to occur, and specifies a framework for resolving them.

Bonatti, Vimercati, and Samarati [11] present a logical framework for the specification and composition of access control policies expressed as (subject, object, access) tuples. They specify several operators such as addition, conjunction, subtraction, and overriding and allow policy administrators to specify the ways in which sub-policies written by different administrative domains within an organization shall be combined. In this paper, we present a similar logical framework, however with a different focus and a different level of abstraction. Our system focuses on combining documents specifying the properties of secure systems, such as the required types of authorization token or levels of encryption, rather than determining a complete list of authorized users. In addition, we allow policy writers to encode their composition preferences in the policies themselves and provide a general composition operator that combines *any* two such policies. Our system can encode operators such as those presented in [11] while also enabling more advanced features such as primitive forms of policy negotiation.

Several other authors (e.g., Bidan and Issarny [10]; Bertino, Jajodia, and Samarati [7]; and Lupu and Sloman [24]) address the problem of conflict resolution as it emerges in access control and management policy composition. Conflicts in access-control policies often concern users that can satisfy different roles or accessible objects matching contradicting rules. Our notion of policies, while certainly related to access-control, takes a different approach by defining the content of a document, rather than specifying access rules. This leads to different types of possible conflicts, requiring different approaches to conflict resolution.

# 3 Framework

In this section, we describe some of the problems that emerge while attempting to compose security policies by examining the possible compositions of several web services security policies. We then present an abstract framework that addresses this problem.

## 3.1 A Motivating Example

Suppose that Alice and Bob each work in different departments within the same organization. In addition, suppose that Carol manages both of these departments. We now imagine a scenario in which Alice and Bob collaborate on a cross-departmental project to develop a web service. It is clear that this web service must abide by Alice's, Bob's, and Carol's security policies, as Alice and Bob developed the web service jointly and both individuals fall under the management of Carol. Consider the case in which all three individuals have different requirements for the security of SOAP requests sent to this web service, specifically, those listed in the first column of Figure 2.

Because all three individuals have different requirements, there are several conflicts that must be resolved before their policies can be composed. First, since Al-

| Policies | Annotations |
|---|---|
| Alice | |
| security token | |
| X.509 certificate | unless otherwise overruled |
| issued by MyUniv/CS | |
| used to encrypt/decrypt session key | |
| encryption | |
| single-use, generated by client | encrypted by X.509 key |
| 3-DES session key | unless otherwise overruled |
| Bob | |
| security token | |
| SAML assertion | in case of low computing capacity so X.509 isn't required |
| issued by MyUniv/CS | |
| encryption | |
| any algorithm | |
| session key encrypted with password-based algorithm | |
| Carol | |
| security token | |
| either SAML or X.509 | or stricter requirements |
| issued by MyUniv | any MyUniv organization |
| message integrity | |

Figure 2: Three policies to be merged

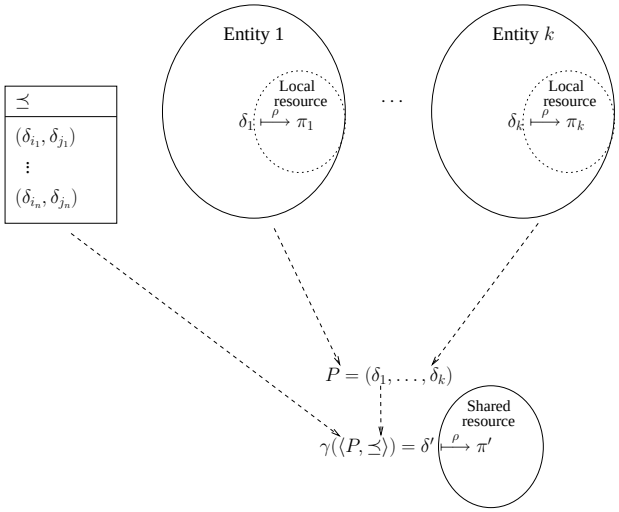


Figure 3: Overview of mathematical framework

ice and Bob have no precedence over each other, which type of security token should we require? If we use a conjunction semantics for composition, we would require both types of security tokens and require that the session key be encrypted twice, once for each algorithm, which may be unnecessary. If we use a disjunction semantics, we would allow either a certificate or a SAML assertion or both. Note that either security token, by itself, fails to satisfy both Alice's and Bob's policies.

In addition, Carol's policy should certainly take precedence over Alice's and Bob's policies in case of any conflicts, but what exactly does that mean? If Bob requires a SAML assertion and Carol allows either a SAML assertion or an X.509 certificate, do we simply require a SAML assertion since it satisfies both policies? Or, is Carol actually insisting that X.509 certificates always be accepted, and that her policy should overrule Bob's? Similarly, since Alice requires security tokens to be issued by MyUniv/CS and Carol only requires them to be issued by MyUniv, which assertion should be used? Or, should the merged policy require two X.509 certificates, one from each issuer?

The second column of Figure 2 shows how annotations help eliminate these ambiguities and specify how the policies should be combined. One of the annotations for Bob's policy even includes some context about the client's computing capacity to be considered. The composite policy, in this case, depends on this context information. In the case of a low-capacity client, a SAML assertion with generic shared-key encryption using a password-based key and message integrity will be used, otherwise an X.509 certificate with 3-DES encryption and message integrity will be used.

## 3.2 Mathematical Framework

We now introduce an abstract model that allows us to reason about the composition of security policies, such as WS-SecurityPolicies, and addresses the types of problems illustrated in our example. Let us denote the set of all such security policies by $\Pi$. Rather than each entity storing a security policy, $\pi \in \Pi$, as its internal policy representation, we propose that it instead stores a semantically-enhanced policy, $\delta$. We will denote the set of all such policies by $\Delta$. A policy $\delta \in \Delta$ is a meta-policy that describes both the requirements enforced by $\delta$ and the means through which $\delta$ should be composed with other meta-policies.

To maintain compatibility with currently deployed standards, we cannot require that applications understand a meta-policy directly. Instead, we define a projection function, $\rho : \Delta \rightarrow (\Pi \cup \perp)$. For a given $\delta \in \Delta$, $\rho(\delta)$ gives us the unique $\pi \in \Pi$ enforced by $\delta$. If $\delta$ cannot be projected onto a $\pi \in \Pi$, $\rho$ will return the value $\perp$.

Because meta-policies specify the way that they are to be combined with other policies, we can define a single composition operation in this system. Let us define an ordered policy structure as a pair $\langle P, \preceq \rangle$ where $P \in 2^{\Delta}$ is a set of meta-policies and $\preceq$ is a partial ordering on $P$. Let $\mathbb{P}$ be the collection of all such ordered policy structures. The existence of $\preceq$ allows us to account for organizational hierarchies or other superiority relationships that exist between policies or their authors. Note that the policy hierarchy induced by $\preceq$ can take the form of a total ordering of policies, an ad-hoc peering arrangement, or some arbitrary structure, allowing complete flexibility in the situations in which our framework is applicable. We then define $\gamma : \mathbb{P} \rightarrow \Delta$ to be the function which produces

a composite meta-policy from an ordered policy structure. Because the annotations (such as those in the second column of Figure 2) already specify how the policy should combine with other policies, $\gamma$ is the only composition operator.

Figure 3 illustrates how a group of $k$ entities can use the operations described in this section. Each entity $j$ contains a local meta-policy $\delta_j$. To enforce this policy on some local resource, the entity simply computes $\rho(\delta_j) = \pi_j$ and uses its existing protection framework to enforce $\pi_j$. However, if the group of $k$ entities wishes to define a security policy to apply to some resource shared among them, the composition of their individual security policies must be computed. Given a partial order, $\preceq$, on the set of meta-policies $P = \{\delta_1, ..., \delta_k\}$, the entities can compute their composite meta-policy $\gamma(\langle P, \preceq \rangle) = \delta'$. The group can then use $\delta'$ to compute the security policy $\rho(\delta') = \pi'$ that will protect their joint resource.

The precise generation of $\preceq$ is outside of the scope of this paper, though one could imagine it being agreed upon by the $k$ entities at the time that their federation is formed. For example, in the case where $\preceq$ reflects an organizational hierarchy, it may be mandated by corporate policy.

# 4 Policy Composition Model

In this section, we describe an instantiation of the policy composition framework described in Section 3.2 based on defeasible logic. Our instantiation is compatible with WSE 2.0, although this framework could be applied to other domains as well. In the context of this instantiation, we first present the syntax for a set of defeasible logic atoms expressing WS-SecurityPolicy assertions. These atoms will be used to construct full defeasible logic expressions. We next discuss the structure of meta-policies in the set $\Delta$, and define algorithms for $\rho$ and $\gamma$ appropriate to our instantiation. We then show that any finite ordered policy structure that can be described in this model can be composed in an automated fashion to form a single meta-policy, $\delta'$.

## 4.1 Syntax for WS-SecurityPolicy Assertions

Rather than reasoning about atomic propositions, we define complex structures, to represent the WS-SecurityPolicy assertions defined in [30]. Each assertion corresponds to a structure, with its defined sub-elements as attributes. While we will only define structures for the `SecurityToken`, `Integrity`, and `Confidentiality` assertions, we could easily define the `SecurityHeader` and `MessageAge` structures following the same pattern. The `Visibility` assertion is somewhat different, because it is logically equivalent to the negation of the `Confidentiality` assertion. We leave this for future work.

We define the `securitytoken` structure as:

```
securitytoken(TokenType, TokenIssuer, {Claims},
   {Ext})
```

where each attribute gives the value of the corresponding tags of the `SecurityToken` assertion. The `integrity` and `confidentiality` structures are defined similarly:

```
integrity({Algorithms}, TokenInfo, {Claims},
   {MessageParts}, {Ext})

confidentiality({Algorithms}, KeyInfo,
   {MessageParts}, {Ext})
```

For example, the following structure is used to represent an X.509 certificate issued by MyUniv. We use the free variables `C` and `E` to indicate that there are no restrictions on the token's claims or extensible elements:

```
securitytoken('x509', 'myuniv', C, E)
```

Similarly, to require an RSA enveloped signature on the entire message using a security token issued by MyUniv, we might use the following structure:

```
integrity({algorithm('signature','rsa'),
   algorithm('transform','enveloped')},
   securitytoken(T,'myuniv',C1,E1), C2,
   {messageparts('xpath',S,'/'), E2})
```

Note that we abuse the notation of predicate logic in two ways. First, we allow set-valued attributes, such as {`Claims`}. Second, the `TokenInfo` and `KeyInfo` attributes are complex structures rather than atomic values. In fact, WS-SecurityPolicy allows these values to be `SecurityToken` assertions themselves; thus, we allow these attributes to be `securitytoken` structures. Similarly, the `Algorithms` and `MessageParts` attributes are sets of complex values. It is possible to make the notation conform to regular predicate logic: for example, we could define a separate predicate for `Claims`, with an attribute that links it to the `securitytoken` structure. However, we think the chosen notation is more readable.

In WSE 2.0, separate policies may be defined for request, reply and fault messages. We define variations of the above structures that allow users to specify that a particular assertion is bound only to a particular class of messages (e.g., the `request_securitytoken` structure describes security token requirements for request messages only). These variations allow users to specify request, reply, and fault policies that are dependent on one another, for example, "I will accept UsernameTokens in the request policy if we require X.509 certificates in the response policy." A structure without a `request_`, `reply_`, or `fault_` prefix applies to all three policy types.

$\delta_{req}$:

```
hassecuritytoken,hasencryption → satisfied
securitytoken('SAML','myuniv/CS',C,E)
        → hassecuritytoken
⋮
```

$\delta_{reas}$:

```
mobile ⇒ securitytoken('SAML','myuniv/CS',C,E)
securitytoken('SAML',I,C,E)
        ⇝ ¬securitytoken('X509',I1,C1,E1)
⋮
```

Figure 4: Bob's policy represented in defeasible logic

## 4.2 Defeasible Logic Policy Representation

Though the above structures can be used in logical statements, deciding the semantics of defeasible rules based on these structures is a non-trivial problem. For example, if we can prove `request_securitytoken('x509', 'myuniv', C, E)`, does it indicate that requests containing an X.509 certificate issued by MyUniv are merely acceptable, or that such a certificate is required?

We solve this problem by defining $\delta$ as a tuple of theories $\langle \delta_{req}, delta_{reas} \rangle$. $\delta_{req}$ defines the requirements of the policy, and $\delta_{reas}$ defines the "reasoning" behind the policy, or the annotations of each assertion. We also allow the use of custom-defined propositions to account for any context information. As an example, Figure 4 illustrates a portion of Bob's policy from Figure 2. The part of $\delta_{req}$ shown states that a security token and encryption are required, and that a SAML assertion issued by MyUniv/CS is a valid security token. $\delta_{reas}$ includes a custom proposition, `mobile`, which, if true, helps conclude that a SAML assertion will be accepted and prevents the policy from requiring an X.509 certificate.

## 4.3 The $\rho$ Function

Recall from Section 3.2 that the $\rho$ function is used to project a meta-policy, $\delta$, onto the security policy, $\pi$, that is to be enforced. In our logical instantiation of the framework presented in Section 3.2, $\rho$ projects our logical meta-policies onto WSE 2.0 `policyDocuments` that describe the set of security policies protecting access to a given web service. This function can be computed automatically by using existing reasoning tools, and allows our framework to be applied directly to a concrete specification and implementation, namely WSE 2.0. The $\rho$ function has two distinct phases: logical derivation and XML generation. Figure 5 presents pseudocode for $\rho$.

The first step of the logical derivation phase is to determine the set of all conclusions, $\mathcal{C}$, that can be derived from

```
1: Function ρ(δ = ⟨δ_req, δ_reas⟩ ∈ Δ) =
2: {# Phase 1: Logical Derivation}
3: Let C = the set of all conclusions that can be derived from δ_reas
4: Let S = {} {# S will be the set of sets of conclusions that
   satisfy δ_req}
5: for all C ∈ 2^C do
6:    Let δ'_req = δ_req ∪ C
7:    if we can derive satisfied in δ'_req then
8:       S = S ∪ {C}
9:    end if
10: end for
11: if S == {} then
12:    return ⊥ {# Cannot output a valid π}
13: end if
14:
15: {# Phase 2: Generate XML}
16: Output(<policyDocument> with namespace declarations
    and mappings header)
17: I = ⋂ S
18: for all prefix ∈ {request, response, fault} do
19:    Output(prefix-specific header tags)
20:    I' = {i ∈ I | i is applicable to messages of type prefix}
21:    if I' ≠ {} then
22:       Output(<All>)
23:       for all i ∈ I' do
24:          Output( DefeasibleToAssertion(i) )
25:       end for
26:       Output(</All>)
27:    end if
28:    Output(<ExactlyOne>)
29:    for all S ∈ S do
30:       Output(<All>)
31:       for all s ∈ (S \ I) applicable to messages of type prefix
          do
32:          Output( DefeasibleToAssertion(s) )
33:       end for
34:       Output(</All>)
35:    end for
36:    Output(</ExactlyOne>)
37:    Output(prefix-specific footer tags)
38: end for
39: Output(</policyDocument>)
```

Figure 5: Pseudocode for the $\rho$ function

$\delta_{reas}$, the reasoning theory of $\delta$. This operation is linear in the number of propositions in the system, as shown in [26]. Lines 5–10 determine all of the subsets of $\mathcal{C}$ that satisfy the requirements of $\delta$. For each $C \in 2^{\mathcal{C}}$, we construct a new requirements theory, $\delta'_{req}$, which contains the old requirements theory, $\delta_{req}$, along with one fact for each conclusion in the set $C$, as shown on line 6. If we can conclude `satisfied` in this new $\delta'_{req}$, then we add the set $C$ to the set of sets $\mathcal{S}$. If $\mathcal{S}$ is empty after this process completes for each subset of $\mathcal{C}$, then we cannot generate a WS-SecurityPolicy that meets the requirements of the meta-policy $\delta$, so $\rho$ returns the value $\perp$.

If $\mathcal{S}$ is non-empty we proceed to generate the XML

1: **Function** $\gamma^*(\alpha = \langle \alpha_{req}, \alpha_{reas} \rangle \in \Delta, \beta = \langle \beta_{req}, \beta_{reas} \rangle \in \Delta, \preceq) =$
2: **if** $\alpha \preceq \beta$ **then**
3: $\gamma^*(\beta, \alpha, \preceq)$
4: **end if**
5: Rewrite rule labels in $\alpha$ and $\beta$ to ensure uniqueness
6: **if** $\beta \preceq \alpha$ **then**
7: {# $\alpha$ is of higher priority}
8: Extend the rule priority partial ordering to allow defeaters in $\alpha$ to block conclusions from $\beta$
9: Extend the rule priority partial ordering to allow conclusions in $\alpha$ to take precedence over conclusions that they can scope in $\beta$
10: **end if**
11: $\delta'_{reas} = \alpha_{reas} \cup \beta_{reas}$
12: $\delta'_{req} = \alpha_{req} \cup \beta_{req}$
13: **return** $\delta' = \langle \delta'_{req}, \delta'_{reas} \rangle$

Figure 6: Pseudocode for the $\gamma^*$ function

representation of the `policyDocument`. This process takes place in lines 15–39 of Figure 5. Note that the function *Output()* writes its argument to the `policyDocument` being generated and the function *DefeasibleToAssertion()* carries out the straightforward task of converting a logical fact to its corresponding WS-SecurityPolicy assertion. For each possible message type (request, response, or fault), the loop on lines 18–38 generates the portion of the `policyDocument` relating to this message type. Lines 17–27 extract the intersection of the possible ways to satisfy $\delta_{req}$ and generate a corresponding `<All>` clause for the current message type. Lines 25–33 generate an `<ExactlyOne>` clause describing the requirements needed in addition to those addressed by the above `<All>` clause.

## 4.4 The $\gamma$ Function

Given a finite set $P \subset \Delta$ of policies to compose and $\preceq$, a partial ordering on $P$, the function $\gamma$ composes the meta-policies in $P$ to construct a new meta-policy $\delta'$. Here we present the details of a function, $\gamma^*$, which composes two meta-policies, given a priority relation between them (see Figure 6). We then discuss how to define $\gamma$ through repeated applications of $\gamma^*$. As with $\rho$, this function can be computed automatically though the use of existing tools, allowing the composition of meta-policies to occur without human intervention.

Composing two meta-policies $\alpha, \beta \in \Delta$ is not a difficult task. In the case that $\alpha$ and $\beta$ are not related by the partial ordering $\preceq$, we simply ensure that the rule labels in the two reasoning theories $\alpha_{reas}$ and $\beta_{reas}$ are unique and collect these rules to form a new reasoning theory. We then create a requirements theory that enforces the requirements of both $\alpha_{req}$ and $\beta_{req}$.

The case in which $\beta \preceq \alpha$ is similar, although it requires extensions of the rule priority partial order defined in $\alpha_{reas}$. This partial order needs to be extended to give each defeater rule $d \in \alpha_{reas}$ precedence over defeasible rules in $\beta_{reas}$ which can possibly be defeated by $d$. Additionally, the rule priority partial order needs to be extended to give each rule concluding $c$ in $\alpha_{reas}$ precedence over each rule in $\beta_{reas}$ whose conclusion can be scoped by either $c$ or $\neg c$.

It can be shown that $\gamma^*$ can be used iteratively to merge any set of meta-policies $P$. We first show that given a finite ordered policy structure which forms a set of trees, we can recursively define the partial function $\hat{\gamma} : \mathbb{P} \rightarrow \Delta$ that iteratively applies $\gamma^*$ to compute the composition of all meta-policies in the structure. Specifically, $\hat{\gamma} : \langle P, \preceq \rangle \mapsto \delta'$, where $\delta'$ is defined as follows:

**Base case ($|P| = 2$):** Let $P = \{\delta_1, \delta_2\}$. Then $\delta' = \gamma^*(\delta_1, \delta_2, \preceq)$.

**Recursive case ($|P| = n > 2$):** Here we must consider three possible cases.

- **Case 1:** $P$ contains two meta-policies, $\delta_1$ and $\delta_2$, which are unrelated to any $\delta_i \in P$ by $\preceq$, i.e., $\delta_1$ and $\delta_2$ are singletons. In this case, let $\delta_{(1,2)} = \gamma^*(\delta_1, \delta_2, \preceq)$. Let $P' = (P \setminus \{\delta_1, \delta_2\}) \cup \{\delta_{(1,2)}\}$. Now, $|P'| = n - 1$ and we can set $\delta' = \hat{\gamma}(\langle P', \preceq \rangle)$.
- **Case 2:** $P$ contains two meta-policies, $\delta_1$ and $\delta_2$, such that $\delta_1$ is the parent of $\delta_2$, $\delta_1$ has no children other than $\delta_2$, and $\delta_2$ is a leaf of the ordered policy structure. In this case, let $\delta_{(1,2)} = \gamma^*(\delta_1, \delta_2, \preceq)$. Let $P' = (P \setminus \{\delta_1, \delta_2\}) \cup \{\delta_{(1,2)}\}$. If $\delta_1$ had no parent, let $\preceq' = \preceq$. Otherwise, because $\preceq$ is a tree structure, there is a single $\delta_k$ such that $\delta_1 \preceq \delta_k$, so let $\preceq' = \preceq$, with the additional relationship $\delta_{(1,2)} \preceq' \delta_k$. Now, $|P'| = n - 1$ and we can set $\delta' = \hat{\gamma}(\langle P', \preceq' \rangle)$.
- **Case 3:** $P$ contains three meta-policies, $\delta_1$, $\delta_2$, and $\delta_3$, such that $\delta_2$ and $\delta_3$ are children of $\delta_1$, and $\delta_2$ and $\delta_3$ are leaves of the ordered policy structure. In this case, let $\delta_{(2,3)} = \gamma^*(\delta_2, \delta_3, \preceq)$. Let $P' = (P \setminus \{\delta_2, \delta_3\}) \cup \{\delta_{(2,3)}\}$ and let $\preceq' = \preceq$ with the additional relationship $\delta_{(2,3)} \preceq' \delta_1$. Now, $|P'| = n - 1$ and we can set $\delta' = \hat{\gamma}(\langle P', \preceq' \rangle)$.

Given that any ordered policy structure which forms a collection of trees can be composed using $\gamma^*$, we argue that this is sufficient to show that any arbitrary ordered policy structure in $\mathbb{P}$ can be composed through iterative applications of $\gamma^*$, using the following sketched algorithm of $\tau : \mathbb{P} \rightarrow \mathbb{P}$:

If an ordered policy structure, $\langle P, \preceq \rangle \in \mathbb{P}$, does not form a collection of trees, then at least one meta-policy has multiple parent nodes. That is, there exists some $\delta_1, \ldots, \delta_n$ such that $\delta_1 \preceq \delta_2, \ldots, \delta_1 \preceq \delta_n$, and $\forall\, (1 \leq \{i, j\} \leq n)\ (\delta_i \neq \delta_j)$. This means that the entity whose policy is described by $\delta_1$ is subordinate to $\delta_2, \ldots, \delta_n$. Assume, without loss of generality, that $\delta_1$ is the lowest such node in the ordered policy structure

To convert this section of $\langle P, \preceq \rangle$ to a tree, we clone the subtree rooted at $\delta_1$ $(n-1)$ times and alter $\preceq$ such that each clone is subordinate to exactly one of $\delta_2, \ldots, \delta_n$. This operation does not modify the meaning of the partial order since each parent still has precedence over the rules in its clone of the subtree rooted at $\delta_1$. To create an ordered policy structure that forms a collection of trees from our starting structure, $\langle P, \preceq \rangle$, we can inductively apply this reasoning starting at the leaves of the ordered policy structure, eliminating all nodes with multiple parents. Let $\tau(\langle P, \preceq \rangle)$ be the resulting policy structure.

Although this algorithm sketch suggests an exponential increase in the running time of $\gamma$ due to the duplication of subtrees, in practice this can be avoided. We can interleave the composition and restructuring of the policies in $\langle P, \preceq \rangle$ by composing the subtrees into a single node before they are cloned.

Finally, we define the algorithm $\gamma : \mathbb{P} \rightarrow \Delta$, the composition operator for any finite ordered policy structure, as $\gamma : \langle P, \preceq \rangle \mapsto \hat{\gamma}(\tau(\langle P, \preceq \rangle))$. This function is well-defined because $\tau$ returns a tree-based policy structure, for which $\hat{\gamma}$ is always defined.

In this section, we have presented an instantiation of the reasoning framework described in Section 3.2 based on defeasible logic for use with WSE 2.0. Additionally, we have shown that it is possible to compose any finite ordered policy structure that can be expressed in this logical model. The composition and projection processes can be carried out in an automated fashion, allowing for the immediate use of our logical policy composition framework with unmodified WSE 2.0-compliant web services. In the following section, we present an extended example illustrating the processes of meta-policy composition and projection.

# 5 An Extended Example

In this section, we present an extended example of how to compose two policies using the defeasible logic framework presented in Section 4. In this example, suppose that Alice and Bob are peers developing a web service using WSE 2.0. Each party has different requirements for the security-relevant properties of messages sent to and from their service.

Figure 7 shows Alice's requirements and reasoning theories. For clarity, we have omitted the `Ext` and `Claims` fields from the structures since they are not used in this example. Alice requires that all messages contain a security token and have an integrity guarantee. Note that no logical structures contain a prefix as discussed in Section 4.1, meaning that Alice has the same requirements for all message types. Alice will accept either an X.509 certificate or both a SAML assertion and a username token. She will accept either an RSA signature with a token issued by MyUniv or an HMAC-SHA1 signature with a username token. R1 through R3 of Alice's reasoning theory state that she will accept either an X.509 certificate issued by MyUniv, a SAML assertion with any issuer, or a username token with any issuer. R4 states that if we are able to conclude the use of an X.509 certificate then we would like to use an RSA signature to protect the integrity of the message. R5 is similar to R4. R7 and R8 state that if we can use X.509 certificates then we do not want to be able to conclude the use of either SAML assertions or username tokens. R9 and R10 prevent both integrity algorithms from being concluded. R6 deserves special attention. R6 uses a context-based predicate, `mobile`, to defeat the use of an X.509 certificate. This means that Alice will allow the use of an authentication token other than an X.509 certificate only if the service is meant to be accessed by mobile devices.

```
Requirements:
hassecuritytoken,hasintegrity → satisfied.
securitytoken('x509','myuniv') → hassecuritytoken.
securitytoken('saml',I),securitytoken('unt',I)
    → hassecuritytoken.
integrity({algorithm('signature','rsa'), algorithm('transform','enveloped')},
        securitytoken(T,'myuniv'),{messageparts('wsse:path',S,'wsp:Body()
        wsp:Header(soap:Header) wse:Timestamp() wse:UsernameToken()
        wse:Addressing()')})
  → hasintegrity.
integrity({algorithm('signature','hmac-sha1'),algorithm('transform','enveloped')},
        securitytoken('unt',I), {messageparts('wsse:path',S,'wsp:Body()
        wsp:Header(soap:Header) wse:Timestamp() wse:UsernameToken()
        wse:Addressing()')})
  →hasintegrity.

Reasoning:
R1: {} ⟹ securitytoken('x509','myuniv').
R2: {} ⟹ securitytoken('saml',I).
R3: {} ⟹ securitytoken('unt',I).
R4: securitytoken('x509',I)
        ⟹ integrity({algorithm('signature','rsa'),
                algorithm('transform','enveloped')},
                securitytoken(T,I), M).
R5: securitytoken('unt',I)
        ⟹ integrity({algorithm('signature','hmac-sha1'),
                algorithm('transform','enveloped')},
                securitytoken('unt',I), M).
R6: mobile ⇝ ¬securitytoken('x509',I).
R7: securitytoken('x509',I) ⇝ ¬securitytoken('saml',I).
R8: securitytoken('x509',I) ⇝ ¬securitytoken('unt',I).
R9: integrity({algorithm('signature','rsa')}, securitytoken(T,'myuniv'), M)
        ⇝ ¬integrity(algorithm('signature','hmac-sha1'), S, M).
R10: integrity({algorithm('signature','hmac-sha1')}, securitytoken('unt',I), M)
        ⇝ ¬integrity(algorithm('signature','rsa'), S, M).
R6 > R1. R7 > R2. R8 > R3.
```

Figure 7: Alice's policy, $\delta_a$

Requirements:
hassecuritytoken,hasconfidentiality → satisfied.
securitytoken('x509','myuniv/cs') → hassecuritytoken.
securitytoken('saml','myuniv/cs/securitygroup') → hassecuritytoken.
confidentiality({algorithm('encryption','rsa')}, securitytoken(T,'myuniv/cs'),
{messageparts('wsse:path',S,'wsp:Body()')})
→ hasconfidentiality.
confidentiality({algorithm('encryption','aes128cbc')}, securitytoken('unt',I)),
{messageparts('wsse:path',S,'wsp:Body()')})
→ hasconfidentiality.

Reasoning:
mobile.
R1: {} ⟹ securitytoken('x509','myuniv/cs').
R2: {} ⟹ securitytoken('saml','myuniv/cs/securitygroup').
R3: securitytoken('x509',I)
⟹ confidentiality({algorithm('encryption','rsa')},
securitytoken('x509',I),
{messageparts('wsse:path',S,'wsp:Body()')}).
R4: {} ⟹ confidentiality({algorithm('encryption','aes128cbc')},
securitytoken(T,I),
{messageparts('wsse:path',S,'wsp:Body()')}).
R5: securitytoken('x509',I) ⇝ ¬securitytoken('saml',I1).
R6: confidentiality({algorithm('encryption','rsa')}, S, M)
⇝ ¬confidentiality({algorithm('encryption','aes128cbc')}, S1, M1).
R6 > R2.

Figure 8: Bob's policy, $\delta_b$

Bob's security policies are shown in Figure 8. Like Alice, Bob uses the same requirements for all message types. Bob requires the use of either an X.509 certificate or a SAML assertion as a security token. Additionally, he would like the body of each message to be encrypted using either RSA or 128-bit AES. R1 and R2 of Bob's reasoning theory state that Bob will accept either a X.509 certificate or a SAML assertion and R3 states that if an X.509 certificate is accepted then he does not want a SAML assertion. R3 and R4 state Bob's reasoning about the confidentiality of the message body. Bob would like to use RSA only if an X.509 certificate is sent with the message. R5 and R6 prevent the conclusion of multiple security tokens and multiple confidentiality algorithms. It is also important to note that Bob has `mobile` as a fact in his theory.

Figure 9 shows the result of running $\gamma^*$ on Alice's and Bob's security policies. Notice that because Alice and Bob are peers, there are no new superiority relationships among the rules. The final result of the projection is a policy that requires two security tokens: a SAML assertion and a username token. Additionally, all messages must include an HMAC-SHA1 signature, and the body of the message must be encrypted with 128-bit AES. Because the policy applies to all types of messages the default policy for request, reply and fault are the same. Both Alice's and Bob's requirements theories accept this policy. The result of the projection function, $\rho$, is shown in Figure 10.

# 6 Implementation

To test the logical reasoning framework presented in Section 4, we have implemented a subset of this framework

R1_1: {} ⟹ securitytoken('x509','myuniv').
R1_2: {} ⟹ securitytoken('saml',I).
R1_3: {} ⟹ securitytoken('unt',I).
R1_4: securitytoken('x509',I)
⟹ integrity({algorithm('signature','rsa'),
algorithm('transform','enveloped')},
securitytoken(T,I), M).
R1_5: securitytoken('unt',I)
⟹ integrity({algorithm('signature','hmac-sha1'),
algorithm('transform','enveloped')},
securitytoken('unt',I), M).
R1_6: mobile ⇝ ¬securitytoken('x509',I).
R1_7: securitytoken('x509',I) ⇝ ¬securitytoken('saml',I).
R1_8: securitytoken('x509',I) ⇝ ¬securitytoken('unt',I).
R1_9: integrity({algorithm('signature','rsa')}, securitytoken(T,'myuniv'), M)
⇝ ¬integrity(algorithm('signature','hmac-sha1'), S, M).
R1_10: integrity({algorithm('signature','hmac-sha1')}, securitytoken('unt',I), M)
⇝ ¬integrity(algorithm('signature','rsa'), S, M).
R1_6 > R1_1. R1_7 > R1_2. R1_8 > R1_3.

R2_1: {} ⟹ securitytoken('x509','myuniv/cs').
R2_2: {} ⟹ securitytoken('saml','myuniv/cs/securitygroup').
R2_3: securitytoken('x509',I)
⟹ confidentiality({algorithm('encryption','rsa')},
securitytoken('x509',I),
{messageparts('wsse:path',S,'wsp:Body()')}).
R2_4: {} ⟹ confidentiality({algorithm('encryption','aes128cbc')},
securitytoken(T,I),
{messageparts('wsse:path',S,'wsp:Body()')}).
R2_5: securitytoken('x509',I) ⇝ ¬securitytoken('saml',I1).
R2_6: confidentiality({algorithm('encryption','rsa')}, S, M)
⇝ ¬confidentiality({algorithm('encryption','aes128cbc')}, S1, M1).
R2_6 > R2_2.

Figure 9: The result of $\gamma^*(\delta_a, \delta_b, \preceq)$

that has the ability to reason about `SecurityToken` assertions. Our implementation consists of a collection of Perl scripts that embody the functionality of $\gamma^*$ and interact with the defeasible reasoning engine along with a Java program that implements the $\rho$ function. For our defeasible reasoning engine, we use the Deimos tool, developed at Griffith University [34]. As discussed in Section 4.1, our syntax for the logical forms of WS-SecurityPolicy assertions overloads the predicate syntax used by Deimos and therefore must first be preprocessed into a format understood by Deimos; we also have implemented an automated translator to facilitate this.

We feel that our instantiation of the framework presented in Section 3 illustrates that defeasible policy composition can meet the goals set forth in Section 1. Our use of defeasible logic makes the policies both relatively easy to specify and human readable. We were able to implement $\gamma$ in such a way as to allow policy composition to be performed automatically and our implementation of $\rho$ allows the merged policy to be projected into a format that can be understood directly by WSE 2.0.

# 7 Web Services and Beyond

While the bulk of this paper focuses on using defeasible logic to combine WS-SecurityPolicy documents, the methods presented in this paper are also applicable to

```
<policyDocument xmlns="...Policy">
 <mappings>
   <endpoint uri="...Service1.asmx">
     <defaultOperation>
       <request policy="#MergedPolicies" />
       <response policy="#MergedPolicies" />
       <fault policy="#MergedPolicies" />
     </defaultOperation>
   </endpoint>
 </mappings>
 <policies xmlns:wsu="..."
           xmlns:wssp="..."
           xmlns:wsp="...">
  <wsp:Policy wsu:Id="MergedPolicies">
    <wsp:All>
      <wsse:SecurityToken>
        <wsse:TokenType>
          wsse:SAMLAssertion
        </wsse:TokenType>
        <wsse:TokenIssuer>
          myuniv/cs/securitygroup
        </wsse:TokenIssuer>
      </wsse:SecurityToken>
      <wsse:SecurityToken>
        <wsse:TokenType>
          wsse:UsernameToken
        </wsse:TokenType>
      </wsse:SecurityToken>
      <wsse:Integrity>
        <wsse:Algorithm Type="wsse:AlgSignature"
                        URI="...hmac-sha1"/>
        <wsse:Algorithm Type="wsse:AlgTransform"
                        URI="...enveloped-signature"/>
        <wsse:TokenInfo>
          <wsse:SecurityToken>
            <wsse:TokenType>
              wsse:UsernameToken
            </wsse:TokenType>
          </wsse:SecurityToken>
        </wsse:TokenInfo>
        <wsse:MessageParts Dialect="...wsse:path">
          wsp:Body() wsp:Header(soap:Header) wse:Timestamp()
          wse:UsernameToken() wse:Addressing()
        </wsse:MessageParts>
      </wsse:Integrity>
      <wsse:Confidentiality>
        <wsse:Algorithm Type="wsse:AlgSignature"
                        URI="...aes128_cbc"/>
        <wsse:KeyInfo>
          <wsse:SecurityToken>
            <wsse:TokenType>
              wsse:UsernameToken
            </wsse:TokenType>
          </wsse:SecurityToken>
        </wsse:KeyInfo>
        <wsse:MessageParts Dialect="...wsse:path">
          wsp:Body()
        </wsse:MessageParts>
      </wsse:Confidentiality>
    </wsp:All>
  </wsp:Policy>
 </policies>
</policyDocument>
```

Figure 10: The resulting `policyDocument`

many other areas. In this section, we discuss additional applications of defeasible policy composition within the realm of web services and an example application outside of the web services domain.

### 7.1 Reliable Messaging Policies

In February 2005, the WS-ReliableMessaging [14] specification, which focuses on the reliable delivery of messages, was introduced. As a complement to this specification, WS-RM Policy [6] allows system designers to make assertions about their requirements with respect to message delivery. WS-RM Policy allows policy writers to make assertions regarding timeouts and retransmission intervals as well as acknowledgement intervals for a particular web service. Our defeasible policy composition system could be easily adapted to handle the composition of WS-RM Policy documents. Additionally, it would be very reasonable to expect that the requirements for security policies would have an effect on the requirements for reliable messaging. For example, if the policy writers want to use a heavy-duty encryption algorithm, it would be important to ensure that inactivity timeouts account for this. Within our system, it is trivial to write assertions that would allow an WS-RM Policy to be created with respect to requirements for security.

### 7.2 Firewall Policies

Another possible use of defeasible policy composition involves merging firewall policies. Consider the case where a research lab shared by members of two different departments within a university is protected by a firewall. The researchers themselves would likely want some control over the types of traffic allowed to pass through the firewall, as would the university and the network operations groups within their departments. Proposed firewall policies and composition preferences could be expressed in defeasible logic, while the partial ordering $\preceq$ can account for any organizational structure. From these policies, an overall policy for the firewall which accounts for each user's preferences could be created using the methods discussed in this paper. Additionally, the efficiency of defeasible logic implies that the firewall policy could be regenerated often, allowing for the inclusion of time-sensitive firewall rules. An area of closely related work involves the examination of complex firewall policies to locate possible conflicts [1]. Our technique is orthogonal to this, in that the techniques presented in [1] can be applied to the composite policy generated using defeasible policy composition to detect, for instance, extraneous rules.

## 8 Conclusions

In this paper, we investigated the use of defeasible logic to perform automated policy composition. We defined an abstract framework that augments a policy with meta-data describing how it should be composed with other such meta-policies. Because the composition preferences are encoded in meta-policies, we require a single composition operator. This allows us to define a fully-automated composition procedure that also takes into account both resource-specific context information and a partial ordering defined among the meta-policies. We then showed how this model can be applied to the composition of web

services security policies.

We then presented an instantiation of this model for the composition of web service security policies. We showed that this model can be used to compose any finite set of meta-policies that can be expressed in the syntax presented in Section 4. This composite meta-policy can be projected onto a WS-SecurityPolicy enforceable by Microsoft's web services extensions, WSE 2.0. We have implemented a subset of this logical framework that can be used to reason about the composition of policies containing `securitytoken` predicates and have successfully used these policies in WSE 2.0 web services. We also discussed additional applications both inside and outside of the realm of web services.

## Acknowledgements

Lee was supported by the NSF under grants IIS-0331707, CNS-0325951, and CNS-0524695 and by a Motorola Center for Communications Graduate Fellowship. Boyer was partially supported by the MacArthur Foundation. Boyer and Gunter were partially supported by NSF Grant CCR-0208996 and ONR Grant N00014-04-1-0562.

## References

[1] E. S. Al Shaer and H. H. Hamend. Discovery of policy anomalies in distributed firewalls. In *IEEE INFOCOMM*, 2004.

[2] Amazon web services. Web Page, Jan. 2006. `www.amazon.com/gp/aws/landing.html`.

[3] G. Antoniou, D. Billington, and M. J. Maher. On the analysis of regulations using defeasible rules. In *HICSS '99: Proceedings of the Thirty-second Annual Hawaii International Conference on System Sciences-Volume 6*, page 6033, 1999.

[4] G. Antoniou and A. Ghose. What is default reasoning good for? applications revisited. In *32nd Hawaii International Conference on System Sciences*, Jan. 1999.

[5] G. Antoniou, M. J. Maher, and D. Billington. Defeasible logic versus logic programming without negation as failure. *Journal of Logic Programming*, 42(1):47–57, 2000.

[6] S. Batres and C. Ferris (Editors). Web services reliable messaging policy assertion(WS-RM Policy). Specification, Feb. 2005. `msdn.microsoft.com/library/en-us/dnglobspec/html/WS-RMPolicy.pdf`.

[7] E. Bertino, S. Jajodia, and P. Samarati. Supporting multiple access control policies in database systems. In *IEEE Symposium on Security and Privacy*, pages 94–109, 1996.

[8] K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *11th ACM conference on Computer and Communications Security*, pages 268–277, Oct. 2004.

[9] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *Computer Security Foundations Workshop (CSFW 06)*, Venice, Italy, July 2006. IEEE.

[10] C. Bidan and V. Issarny. Dealing with multi-policy security in large open distributed systems. In *European Symposium on Research in Computer Security (ESORICS)*, pages 51–66, 1998.

[11] P. Bonatti, S. D. C. di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *7th ACM Conference on Computer and Communications Security (CCS '00)*, pages 164–173, Nov. 2000.

[12] L. Cholvy and F. Cuppens. Analyzing consistency of security policies. In *18th IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.

[13] D. Eastlake and J. Reagle (Chairs). W3C XMLDSig working group. Web Page, Jan. 2006. `www.w3.org/Signature/`.

[14] C. Ferris and D. Langworth (Editors). Web services reliable messaging protocol(WS-ReliableMessaging). Specification, Feb. 2005. `msdn.microsoft.com/library/en-us/dnglobspec/html/WS-ReliableMessaging.p%df`.

[15] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure Working Group, Global Grid Forum*, Jun. 2002.

[16] K. Frankish. Non-monotonic inference. In *The Encyclopedia of Language and Linguistics*. Elsevier, second edition, 2005.

[17] Google web APIs. Web Page, Jan. 2006. `www.google.com/apis/`.

[18] G. Governatori, A. H. M. ter Hofstede, and P. Oaks. Defeasible logic for automated negotiation. In P. Swatman and P. M. Swatman, editors, *Proceedings of CollECTeR*, 2000.

[19] G. Governatori, A. H. M. ter Hofstede, and P. Oaks. Is defeasible logic applicable? In G. Antoniou and G. Governatori, editors, *Proceedings of the 2nd Australasian Workshop on Computational Logic*, pages 47–62, Brisbane, January 2001. Queensland University of Technology.

[20] B. N. Grosof, Y. Labrou, and H. Y. Chan. A declarative approach to business rules in contracts: courteous logic programs in XML. In *ACM Conference on Electronic Commerce*, pages 68–77, 1999.

[21] J. Halpern and V. Weissman. Using first-order logic to reason about policies. In *IEEE Computer Security Foundations Workshop (CSFW'03)*, Jun. 2003.

[22] S. Horrell. Web services enhancements 2.0 support for WS-Policy. Web Page, July 2004. `msdn.microsoft.com/library/en-us/dnwse/html/wse2wspolicy.asp`.

[23] C. Kaler and A. Nadalin (Editors). Web services federation language (WS-Federation). Specification, Jul. 2003. `www-106.ibm.com/developerworks/webservices/library/ws-fed/`.

[24] E. C. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999.

[25] K. D. Lux, M. J. May, N. L. Bhattad, and C. A. Gunter. WSEmail: Secure internet messaging based on web services. In *International Conference on Web Services*, Orlando, FL, July 2005.

[26] M. J. Maher. Propositional defeasible logic has linear complexity. *Theory and Practice of Logic Programming*, 1(6):691–711, 2001.

[27] M. J. May, W. Shin, C. A. Gunter, and I. Lee. Securing the drop-box architecture for assisted living. In *Formal Methods in Software Engineering (FMSE '06)*, Alexandria, VA, November 2006. ACM.

[28] M. McDougall, R. Alur, and C. A. Gunter. A model-based approach to integrating security policies for embedded devices. In *ACM EMSOFT*, Sept. 2004.

[29] Michael McDougall. *Modeling and Analyzing Integrated Policies*. PhD thesis, University of Pennsylvania, 2004.

[30] A. Nadalin (Editor). Web services security policy language (WS-SecurityPolicy). Web Services Specification, 2002. `www.verisign.com/wss/WS-SecurityPolicy.pdf`.

[31] D. Nute. Defeasible logic. In *14th International Conference on Applications of Prolog*, Oct. 2001.

[32] J. Reagle (Chair). W3C XML encryption working group. Web Page, Jan. 2006. `www.w3.org/Encryption/2001/`.

[33] D. M. Reeves, M. P. Wellman, B. N. Grosof, and H. Y. Chan. Automated negotiation from declarative contract descriptions. In *17th National Conference on Artificial Intelligence, Workshop on Knowledge-Based Electronic Markets (KBEM)*, Jul. 2000.

[34] A. Rock. Deimos: A query answering defeasible logic system. Technical report, Griffith University, Mar. 2004. `www.cit.gu.edu.au/~arock/defeasible/doc/Deimos-long.pdf`.

[35] J. Schlimmer (Editor). Web services policy framework (WS-Policy). Web Services Specification, 2004. `ftp://www6.software.ibm.com/software/developer/library/ws-policy.pdf`.

[36] C. Sharp (Editor). Web services policy attachment (WS-PolicyAttachment). Specification, Sept. 2004. `msdn.microsoft.com/library/en-us/dnglobspec/html/ws-policyattachment.as%p`.

[37] SOAP version 1.2. W3C Recommendation, Jan. 2006. `www.w3.org/TR/soap12`.