RESPONDING TO POLICIES AT RUNTIME IN TRUSTBUILDER

by

Bryan Smith

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

March 2004

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Bryan Smith

Each member of the following graduate committee has read this thesis and by majority vote has been found to be satisfactory in its present form satisfying the thesis requirements for the degree of Master of Science within the Department of Computer Science at Brigham Young University.

Date	Dr. Kent E. Seamons, Chair
Date	Dr. Charles Knutson
Date	Dr. Dennis Ng

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Bryan Smith in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Dr. Kent E. Seamons
Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

RESPONDING TO POLICIES AT RUNTIME IN TRUSTBUILDER

Bryan Smith

Department of Computer Science

Master of Science

Automated trust negotiation is the process of establishing trust between entities with no prior relationship through the iterative disclosure of digital credentials. One approach to negotiating trust is for the participants to exchange access control policies to inform each other of the requirements for establishing trust. When a policy is received at runtime, a compliance checker determines which credentials satisfy the policy so they can be disclosed. In situations where several sets of credentials satisfy a policy and some of the credentials are sensitive, a compliance checker that generates all the sets is necessary to insure that the negotiation succeeds whenever possible. Compliance checkers designed for trust management do not usually generate all the satisfying sets. In this thesis, we present two practical algorithms for generating all satisfying sets given a compliance checker that generates only one set. The ability to generate all of the combinations provides greater flexibility in how the system or user establishes trust. For example, the least sensitive credential combination could be disclosed first. These ideas have been implemented in TrustBuilder, our prototype system for trust negotiation.

ACKNOWLEDGMENTS

I would like to thank my graduate advisor, Dr. Kent E. Seamons, for his time and guidance. I would like to thank my graduate committee, Dr. Charles Knutson and Dr. Dennis Ng, and research peers for their feedback. I would like thank my wife, Gabriel, for her support and patience. Finally, I would like to thank my Heavenly Father for all that I am.

Financial support for this research was provided by DARPA through AFRL contract number F33615-01-C-0336 and through Space and Naval Warfare Systems Center San Diego grant number N66001-01-1-8908. Support was also provided by the National Science Foundation under grant number CCR-0325951 and prime cooperative agreement number IIS-0331707, and The Regents of the University of California.

Contents

1 Introduction **1**

1.1 Thesis Statement 2

1.2 Thesis Organization 3

2 Policy Exchange during Trust Negotiation **5**

3 Satisfying Set Generation **11**

3.1 Policy Modification 12

3.2 Credential Set Modification 15

4 Exploring Alternative Ways to Establish Trust **25**

5 Implementation **27**

5.1 TrustBuilder 27

5.1.1 TrustBuilder1-Relevant Strategy Implementation 27

5.1.2 Negotiation Trees 28

5.1.3 Backtracking Algorithm for Automated Trust Negotiation (BAAT) 29

5.1.4 Policies in Minimal DNF 31

5.2 HTTP 32

6 Related Work **35**

7 Conclusion and Future Work **37**

7.1 Contributions 37

7.2 Future Work 38

A UML Diagrams **45**

List of Figures

2.1 Compliance checker in a traditional mode of operation 7

2.2 Compliance checker that is able to incrementally determine all minimal sets of credentials that satisfy a remote policy 8

3.1 Example TPL policy for community college enrollment scenario . . . 14

3.2 Modified $ValidIDHolder$ role definition after a satisfying set containing an employee ID has been obtained 15

3.3 Lattice containing the subsets of {A, B, C} 16

3.4 Pseudo-code for the SSgen algorithm 19

3.5 Execution time for the SSgen algorithm according to the size of the union of all satisfying sets and the number of satisfying sets. 23

5.1 Pseudo-code for the TrustBuilder1-Relevant Strategy implementation 29

5.2 A negotiation tree in TrustBuilder 30

5.3 BAAT pseudo-code 31

5.4 TPL role definition for role A 32

5.5 Modified TPL rol definition in minimal DNF for role A 33

5.6 HTTP prototype architecture 33

A.1 Use case for trust negotiation with policy exchange 46

A.2 Sequence diagram for TrustBuilder 46

A.3 Class diagram for TrustBuilder 47

Chapter 1 — Introduction

In the physical world, individuals may establish trust by presenting paper credentials, such as a driver's license, passport, employee ID, credit card, etc., to demonstrate properties about themselves that prove their trustworthiness. These credentials serve as letters of introduction between parties with no pre-existing relationship.

In our research, we explore ways to facilitate similar kinds of interactions in the digital world. Traditional approaches to establishing trust on-line were designed for closed systems, where the participants know each other in advance. They frequently rely on identity-based approaches such as a username and password. In open systems like the Internet, these traditional approaches fail because the participants in a transaction are often strangers and are not in the same security domain.

Trust negotiation [17, 18, 19] is a new approach to establishing trust between strangers through the disclosure of digital credentials containing properties of the participants. Digital credentials are signed statements by trusted third parties that assert properties of the credential owner. Credentials are disclosed during trust negotiation to demonstrate trustworthiness. One approach to implementing digital credentials is to use X.509v3 attribute certificates.

A naïve approach to trust negotiation would be for Alice to disclose all her credentials to an unfamiliar server, Bob, whenever Alice makes a request. In the event the service is protected by an access control policy, Bob can check whether or not Alice possesses the requisite credentials. This simple approach is akin to a first-time customer plopping down their wallet or purse on the counter, and inviting the merchant to rifle through its contents to determine whether or not to trust the customer.

Obviously, this is an unacceptable solution to the problem of trust establishment because it completely ignores credential sensitivity.

A more reasonable approach, which considers credential sensitivity, is to first associate an access control policy with each sensitive credential that specifies the credentials that must be received from the other party before the sensitive credential can be disclosed. A trust negotiation begins when one party discloses all non-sensitive credentials. Then the two parties take turns disclosing all the credentials whose access control policies have been satisfied by the other party's disclosed credentials. Eventually, the negotiation either succeeds when trust is established, or fails when one party has nothing further to disclose. A disadvantage to this approach is that credentials may be unnecessarily disclosed.

A third approach is for each party to disclose access control policies to each other that are relevant to the negotiation. Subsequent credential disclosures are then based on a need to know. This thesis focuses on this third approach to trust negotiation.

When Bob receives a policy from Alice during a trust negotiation, he uses a compliance checker to determine two things: 1) whether or not his local credentials satisfy the policy, and 2) which credentials satisfy the policy. Bob needs to know which credentials satisfy the policy so that he knows what he must disclose in order to advance the negotiation. Compliance checkers designed for trust management [4] are usually not designed to provide this second capability.

Even if a compliance checker identifies the credentials that satisfy a policy, this is not enough for trust negotiation to succeed whenever possible.

1.1 Thesis Statement

In this thesis, we explain why a compliance checker must be able to generate all the ways that a policy can be satisfied. Some existing compliance checkers determine only one way that a set of credentials satisfies a policy. We present two practical ways to adapt such a compliance checker so that it is able to generate all the ways that a policy is satisfied. One approach requires rewriting the policy every time the compliance checker is invoked so that prior solutions are excluded in order to force additional solutions to be generated. Another approach involves modifying the set of input credentials each time the compliance checker is invoked in order to obtain all minimal satisfying sets.

Definition 1.1. A *minimal satisfying set* is a set of credentials that satisfies a policy such that no proper subset also satisfies the policy.

Throughout the remainder of this thesis, all references to a satisfying set imply a minimal satisfying set.

Definition 1.2. A *compliance checker* is a function $f : \{C, P\} \Rightarrow S$, where C is a set of credentials, P is a policy, and S is a subset of C that $\models$ (minimally satisfies) P. If C is empty or contains no satisfying sets, then S is empty.

Once a compliance checker is able to produce all the satisfying sets, there are several ways to use this feature during trust negotiation. Generating some or all of the satisfying sets at once allows the negotiation agent to select the order that alternative ways of establishing trust are considered. We introduce several alternatives and discuss the merits of each approach.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents an overview of trust negotiations involving policy exchanges and discusses the requirements this approach imposes on compliance checkers. Chapter 3 presents two approaches for generating all the minimal sets that satisfy an access control policy received during a trust negotiation. In Chapter 4, the three methods to generating satisfying sets and choosing which negotiation path to explore are presented. Chapter 5 contains a discussion of implementation issues encountered while employing these ideas in TrustBuilder. Chapter 6 contains related work and Chapter 7 contains conclusions and future work.

Chapter 2 — Policy Exchange during Trust Negotiation

The following hypothetical example of a trust negotiation illustrates how the participants can learn about each other's requirements for establishing trust by exchanging access control policies. This allows the participants to determine in private whether or not they satisfy the policy, and whether they are willing to disclose credentials that satisfy the policy.

Suppose Alice is registering online for community college. The server, Bob, requires some form of ID from Alice. Bob's access control policy specifies that several forms of ID are acceptable, including a driver's license, employee ID, and a military ID. Alice is employed by a major retailer and is a member of the Army reserve, so she has all three kinds of ID. Her employer requires that her employee ID only be used for business purposes, and Alice only discloses her military ID to authorized government servers.

Using semi-automated trust negotiation, Alice and Bob (or more precisely, their trust negotiation agents) could interact as follows. Alice requests to enroll in community college. Bob responds with an access control policy that requests that Alice submit a digital ID. Alice evaluates the policy and determines that she can satisfy Bob's policy in three different ways. Up to this point, Alice's trust negotiation agent completed these steps automatically on her behalf. At this point, the agent notifies Alice interactively that she must submit one of her three forms of ID, and Alice manually selects the form of ID that she is most willing to disclose.

With automated trust negotiation, Alice's trust negotiation agent relieves her of the need to manually select which credentials to disclose. Instead, credential disclosure is controlled by access control policies that specify the credentials the other party must first disclose in order to receive a sensitive credential. The following is one way the negotiation between Alice and Bob might proceed, and assumes a compliance checker that generates solutions to a policy in random order. Alice requests to enroll in community college. Bob responds with an access control policy that requests that Alice submit some form of ID. Alice evaluates the policy and determines that she can satisfy the policy using her military ID. Since it is sensitive, Alice discloses the access control policy to request that Bob demonstrate that he is an authorized military server. Bob notifies Alice that he cannot satisfy her policy. Alice then determines that her employee ID satisfies the policy. Since it is also sensitive, Alice requests that Bob demonstrate he is an authorized server representing Alice's employer, or someone that her employer trusts. Bob notifies Alice that he cannot satisfy this policy. Finally, Alice determines that her driver's license satisfies the policy, and she discloses her credential to Bob, allowing her to complete the enrollment process. Note in this example that the negotiation could have been greatly simplified if Alice had considered her driver's license before the other more sensitive credentials. We will return to this point in Chapter 4.

There are two distinct modes of operation required by a compliance checker during trust negotiation. The first mode is the traditional way a compliance checker functions in a trust management environment, as shown in Figure 2.1. Alice's negotiation agent invokes the compliance checker to determine whether to grant Bob access to a sensitive resource. Alice's negotiation agent provides the compliance checker with Alice's access control policy for the resource, the credentials Bob has disclosed, and possibly other information that Alice has made available locally (e.g., time of day, proof of her age, the results of Alice's checks for revocation of the credentials Bob has submitted, etc.). The compliance checker produces a Boolean result indicating whether or not the credentials satisfy the policy. PolicyMaker has a compliance checker that returns a justification when access is denied [5]. KeyNote supports the specification of a user-defined justification whenever the result is false [3]. During trust negotiation, the compliance checker adopts this first mode of operation to determine whether to grant access to a sensitive service or to disclose a sensitive credential or policy.

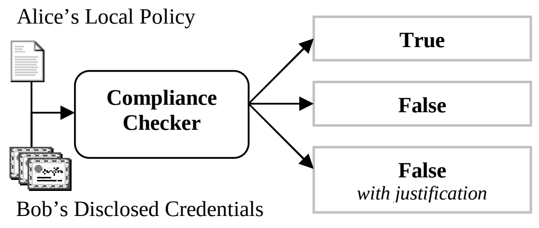

Figure 2.1: Compliance checker in a traditional mode of operation

The second mode of operation required of a compliance checker during trust negotiation is not usually necessary in traditional trust management environments, but is required if policies are disclosed during negotiation. Suppose Bob requests a sensitive resource and does not supply the necessary credentials. Rather than simply deny the request, Alice can disclose the access control policy governing the sensitive resource in order to guide the negotiation to a successful conclusion. Upon receipt of Alice's access control policy, Bob can make use of his compliance checker according to the diagram in Figure 2.2. Bob's negotiation agent invokes the compliance checker to determine whether Bob has credentials cached locally that satisfy Alice's disclosed policy. The compliance checker accepts Alice's disclosed policy and Bob's local credentials as input, and possibly other information from Bobs environment (time of day, etc.). The compliance checker returns a Boolean result indicating whether Bob has credentials that satisfy Alice's policy. In addition, Bob's negotiation agent needs the compliance checker to return a set of local credentials that satisfies Alice's policy so that those credentials may be disclosed during the negotiation. In some cases, Bob may have more than one set of credentials that satisfies Alice's policy. However, some of Bob's credentials in a satisfying set may be sensitive, and Alice may not be able to qualify for access to them. Thus, Bob's compliance checker must be able to determine all minimal combinations of credentials that satisfy Alice's policy in order to exhaust all possibilities for establishing trust. REFEREE is an example of an early trust management system that returns a justification when a policy is satisfied [7].

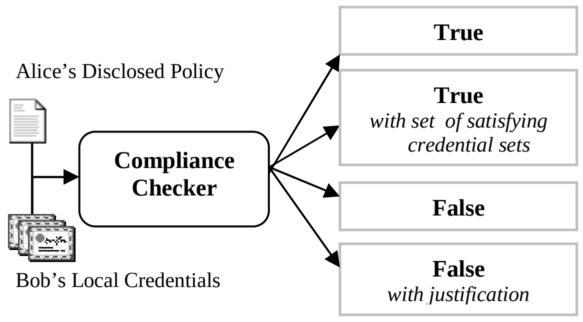

Figure 2.2: Compliance checker that is able to incrementally determine all minimal sets of credentials that satisfy a remote policy

Compliance checkers can be categorized according to the modes of operation supported. A type-1 compliance checker is the traditional type of compliance checker used in trust management systems, which determines whether to grant access to a protected resource. This type lacks support for the second mode of operation described previously. A type-2 compliance checker returns only a single set of credentials that satisfy the policy. The runtime engine for the IBM Trust Establishment (TE) system [9] is an example of this type of compliance checker. A type-3 compliance checker returns all the sets of credentials that satisfy the policy and is required by trust negotiation when exchanging policies [15].

A trust negotiation is said to be complete if it succeeds whenever possible. In order for the trust negotiation to be complete, Alice's compliance checker must be capable of determining all the ways that her credentials satisfy Bob's policy. If her compliance checker does not generate all minimal satisfying credential sets, then trust negotiation may be incomplete. This situation can occur when the following two conditions are met. First, the set of satisfying credentials returned by Alice's compliance checker contains credentials that are governed by policies that Bob cannot satisfy. Second, there is another set of satisfying credentials that Alice's compliance checker is unable to identify that are governed by policies that Bob can satisfy. When both of these conditions occur, Alice and Bob will not be able to establish trust even though it is theoretically possible for trust to be established.

Chapter 3 — Satisfying Set Generation

Most compliance checker implementations are either type-1 or type-2. They are not designed to satisfy the needs of policy exchange during trust negotiation, and so they do not generate all the sets that satisfy a policy.

TrustBuilder is our prototype trust negotiation system under development at BYU. The initial implementation of TrustBuilder leveraged the IBM TE system to provide a policy language for expressing the credential combinations necessary to establish trust and a compliance checker for making trust decisions. The TE system is freely available on IBM AlphaWorks, but the source code is proprietary.

Our experience using TE for trust negotiation was the impetus for the research presented in this paper. TE has a type-2 compliance checker. In order for trust negotiation to be complete when policies are exchanged, the compliance checker must generate all the minimal sets that satisfy a policy. One way to generate these sets using the TE compliance checker is a brute force approach. Given a policy P and a set of local credentials L, the compliance checker is invoked $2^{|L|} - 1$ times for each set from the power set of L. The disadvantage of this approach is that the cost grows exponentially with the number of local credentials.

In the remainder of this section, we discuss and analyze two approaches for generating all the sets that satisfy a policy, given a type-2 compliance checker. These approaches, known as policy modification and credential set modification, are more efficient than the brute force approach.

3.1 Policy Modification

The first approach to generating all of the sets that satisfy a policy is to repeatedly invoke the compliance checker with the same set of local credentials, but modify the policy to exclude all the satisfying sets that have already been generated. The process begins with no known satisfying sets. The compliance checker is invoked with the set of local credentials L and the policy P. If a set of credentials S_1 is returned by the compliance checker, then P is modified at runtime to exclude S_1 as follows: P AND NOT (P_1) where P_1 is a conjunction of all the credentials in S_1. Note that the set S_1 is excluded, but the individual credentials that comprise S_1 can still be members of other minimal sets that satisfy P. The compliance checker is invoked a second time with the set of local credentials and the modified policy. After each invocation of the compliance checker, the policy is further modified according to the form P AND NOT (P_1 OR ... OR P_n) where n is the number of satisfying sets generated thus far. The process of policy modification and compliance checker invocation continues until the compliance checker returns an empty set, which indicates that all satisfying sets have been generated. Using this approach, all of the satisfying sets will be generated after $N+1$ invocations of the compliance checker where N is the number of satisfying sets.

The following illustrates how TrustBuilder adopts this approach using the XML-based Trust Policy Language (TPL) supported by the IBM TE system [9]. The TE system maps a subject to a role based on the subject's credentials, a role-assignment policy established by the owner of the resource, and the roles of the issuers of the credentials. The role assignment policy is a set of TPL role definitions. TPL role definitions contain a collection of ⟨GROUP⟩ tags. Inside the ⟨GROUP⟩ tag, ⟨RULE⟩ tags are used to state the requirements for role membership. If any of the rules are satisfied, then the supplicant is a member of the group. A RULE entity contains a series of INCLUSION tags and FUNCTION tags. An INCLUSION entity specifies constraints on credentials, such as type and issuer. Further constraints on credentials can be specified within a FUNCTION entity.

In the community college enrollment scenario discussed previously in Section 2, Alice has three identification credentials: an employee ID, a military ID, and a driver's license. Using TPL, the college server (Bob) requires that a person be a member of the $ValidIDHolder$ group to register. An example of a TPL policy for this scenario is shown in Figure 3.1.

The TE compliance checker is designed to return a single set that satisfies the policy. Suppose Alice invokes the compliance checker with the policy contained in Figure 3.1 and her three identification credentials as input. Assume the compliance checker first returns a set containing Alice's employee ID. In order for Alice to determine if there are additional satisfying sets using policy modification, the TPL policy is modified to exclude the set containing Alice's employee ID. Figure 3.2 illustrates how the $ValidIDHolder$ group can be modified to eliminate Alice's employee ID as a satisfying set, and force the compliance checker to return another satisfying set, if one exists.

Assume the modified policy in Figure 3.2 is fed to the compliance checker along with Alice's credentials, and her military ID is returned as another satisfying set. The policy can be further modified to also exclude Alice's military ID as a solution. The next invocation returns the set containing Alice's driver's license. Finally, the policy is modified to also exclude her driver's license, and the next invocation of the compliance checker returns the empty set because there are no more satisfying sets.

The policy modification approach results in fewer calls to the compliance checker

```
<GROUP NAME= "self" ></GROUP>

<GROUP NAME="Company ">
      <RULE>
            <INCLUSION ID="compcert" TYPE="TrustedCompany"
              FROM="self"/>
      </RULE>
</GROUP>

<GROUP NAME= "USArmedForces" >
      <RULE>
            <INCLUSION ID="usAFcert" TYPE="USArmedForces" FROM="self"/>
      </RULE>
</GROUP>

<GROUP NAME= "State" >
      <RULE>
            <INCLUSION ID="statecert" TYPE="State" FROM="self"/>
      </RULE>
</GROUP>

<GROUP NAME="ValidIDHolder">
      <RULE>
            <INCLUSION ID="empIDcert" TYPE="EmployeeID"
              FROM="Company"/>
      </RULE>
      <RULE>
            <INCLUSION ID="mIDcert" TYPE="MilitaryID"
              FROM="USArmedForces"/>
      </RULE>
      <RULE>
            <INCLUSION ID="dlcert" TYPE="DriversLicense" FROM="State"/>
      </RULE>
</GROUP>
```

Figure 3.1: Example TPL policy for community college enrollment scenario

compared to the brute force approach, especially when there is a large number of local credentials and only a few sets that satisfy a policy. As the number of satisfying sets grows, the complexity of the modified policy increases, requiring additional processing each time the compliance checker is invoked. Experiments using the TrustBuilder implementation of policy modification indicate that the added overhead for generating and evaluating modified policies using the TE system is negligible for typical problem sizes. For example, we developed two test scenarios involving 50 local credentials, policies with 4 or 5 satisfying sets each, and each satisfying set consisting of 2 to 3 credentials. The cost of each policy modification was approximately .02 seconds, with no noticeable increase in policy evaluation time. The experiments were run on a 1.4 GHz Pentium 4 processor with 512 MB of RAM.

```
<GROUP NAME="ValidIDHolder">
    <RULE>
                <INCLUSION ID="empIDcert" TYPE="EmployeeID" FROM="Company"/>
                <FUNCTION>
                  <AND>
                   <NE>
                     <FIELD ID="empIDcert" NAME="issuerName"/>
                     <CONST>CompanyX</CONST>
                   </NE>
                    <NE>
                     <FIELD ID="empIDcert" NAME="X509serialNo"/>
                     <CONST>2345</CONST>
                   </NE>
                  <\AND>
                </FUNCTION>
       </RULE>
    <RULE>
                <INCLUSION ID="mIDcert" TYPE="MilitaryID"
                  FROM="USArmedForces"/>
    </RULE>
    <RULE>
                <INCLUSION ID="dlcert" TYPE="DriversLicense" FROM="State"/>
    </RULE>
</GROUP>
```

Figure 3.2: Modified $ValidIDHolder$ role definition after a satisfying set containing an employee ID has been obtained

In order to adopt the policy modification approach, the policy language must support the proper semantics for the AND and NOT operators. The recipient of the original policy is not able to treat the policy as a black box, but must be capable of rewriting the policy according to the satisfying sets that are generated.

3.2 Credential Set Modification

The second approach to generating all of the sets that satisfy a policy is to repeatedly invoke the compliance checker with the same policy, but modify the input set of local credentials based on the satisfying sets that have already been generated. In this section, we present the Satisfying Set Generation (SSgen) algorithm, which accepts a policy and a set of credentials and returns all the credential sets that minimally satisfy the policy.

The SSgen algorithm assumes a restricted form of policies according to the following definition.

Definition 3.1. A *policy* P is a disjunction of rules, where rules are conjunctions of credentials. A rule specifies a minimal satisfying set.

By definition, a policy cannot be (A AND B) OR A because $A \subset \{A, B\}$.

A lattice can be formed from the subsets of the set of local credentials L according to the following rule: $Y \rightarrow^* X$, iff $Y \subseteq X$. Figure 3.3 illustrates a lattice formed from the set $\{A, B, C\}$. The SSgen algorithm uses the lattice to determine the sequence of sets with which to invoke the compliance checker.

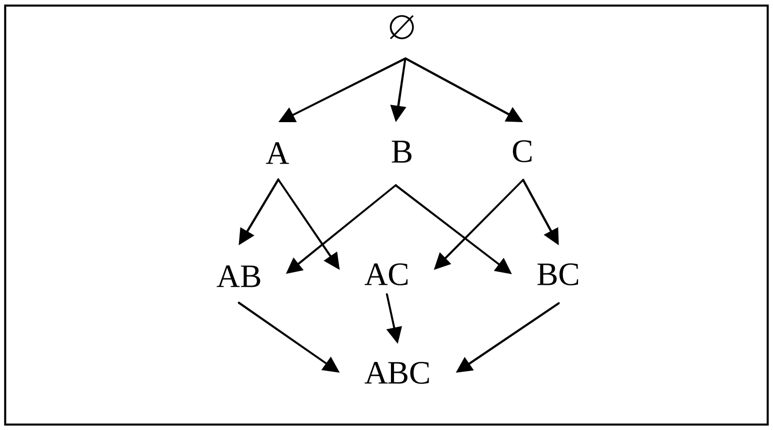

Figure 3.3: Lattice containing the subsets of {A, B, C}

A naïve implementation of the brute force approach to generating all the satisfying sets is to conduct a breadth-first traversal of the lattice, invoking the compliance checker for each non-empty set in the lattice. This results in $2^{|L|} - 1$ invocations of the compliance checker. This is impractical when $|L|$ is large.

The following properties provide insights into ways to improve on the naïve brute force algorithm. We will refer to sets in the lattice in Figure 3.3 for illustrative purposes.

Property 1. A search for a satisfying set can begin at the bottom of the lattice, and invoke the compliance checker with the set of all local credentials $\{A, B, C\}$. This is especially significant when there are no satisfying sets because the first call to the compliance checker will fail and the search is finished. If there are any satisfying sets, the first call to the compliance checker will find one.

Property 2. When the compliance checker returns a satisfying set, by definition, no proper subset of that set can also be a satisfying set. These proper subsets can be eliminated from the search space. For example, suppose the first call to the compliance checker returns the set $\{A,B\}$. The sets $\emptyset$, $\{A\}$, and $\{B\}$ can be eliminated.

Property 3. When the compliance checker returns a satisfying set, by definition, no proper supersets of that set can also be a satisfying set. These proper supersets can be eliminated from the search space. For example, suppose the first call to the compliance checker returns the set $\{A\}$. The sets $\{A, B\}$, $\{A, C\}$, and $\{A, B, C\}$ can be eliminated.

Property 4. When the compliance checker returns an empty set, by definition, no proper subset of the input set can be a satisfying set. These proper subsets can also be eliminated from the search space. For example, suppose the compliance checker is called with the set $\{A, C\}$ as input. If no satisfying set is returned, the sets $\{A\}$ and $\{C\}$ can be eliminated.

Property 5. Suppose the compliance checker returns a satisfying set S containing m members. To find additional satisfying sets, each proper subset of S can be combined with all the local credentials not in S and then passed to the compliance checker. If there are any more satisfying sets besides S, this approach will find one in at most $2^m - 1$ calls to the compliance checker. If no other satisfying set is found, then all other sets in the lattice will be eliminated during the process.

Property 6. In a lattice fashioned after the one shown in Figure 3.3, all nodes that are a distance i from the empty set comprise all of the i-subsets (i.e., subsets containing i members) in the lattice. For example, the nodes in Figure 3.3 that are a distance of 2 from the empty set form all the 2-subsets $\{A, B\}$, $\{A, C\}$, and $\{B, C\}$. For a lattice and a given i, suppose all of the i-subsets are input to the compliance checker and fail to return a satisfying set, or are proper subsets of failed input sets. According to Property 4, no other subset in the lattice with less than i members is a satisfying set. The remaining subsets are eliminated from consideration.

Property 7. Suppose the compliance checker returns a satisfying set S containing m members. If S is the only satisfying set in the lattice, this can be determined in $m+1$ calls to the compliance checker based on Properties 1, 5, and 6. The first call to the compliance checker returns set S (Property 1). The next m calls to the compliance checker involve sets containing the $(m-1)$-subsets of S (Property 5). If the m calls to the compliance checker fail to produce another satisfying set, all other subsets in the lattice are eliminated (Property 6).

The SSgen algorithm (see Figure 3.4) is based on the properties listed above. The following theorem states that the SSgen algorithm finds all the satisfying sets for a given policy.

```
L is the set of local credentials
P is a policy
B is the set of known minimal satisfying sets, which can be empty
E is the set of sets known to contain no subsets that satisfy P, which can be empty
U is union of all sets in B
An n-subset is a subset that contains n members.
$\mathcal{P}(U)$ returns the power set of U

1  J = complianceChecker(L, P)
2  if (J is not empty)
3      $B = B \cup \{J\}$
4      Let $S = \mathcal{P}(U)$
5      Let $n = |U|$
6      while ($n > 0$)
7          Let T be all the (n-1)-subsets $\in S$
8          For each set, $D \in T$
9              $A = D \cup (L \setminus U)$
10             if (A is not a superset or a subset of a set $\in B$ and is not a subset of a set $\in E$)
11                 J = complianceChecker(A, P)
12                 if (J is empty)
13                     $E = E \cup \{A\}$
14                 else
15                     if ($J \setminus U$ != $\phi$)
16                         goto line 3 because |U| will increase
17                     $B = B \cup \{J\}$
18         if ($\forall t \in T$ ($t \cup (L \setminus U) \subseteq E$))
19             n = 0
20         else
21             n = n-1
22     end-while
23 end-if
```

Figure 3.4: Pseudo-code for the SSgen algorithm

Theorem 1. *$R \subseteq L$ and $R \models P$ if and only if $R \in B$, where B is the set of sets returned by the SSgen algorithm, L is the set of local credentials, P is a policy, and R is an arbitrary set of credentials.*

Proof. First, let $R \in B$. In the SSgen algorithm, the code to add R to B can only be reached by sets containing subsets of U unioned with $L \setminus U$ that satisfy P. The compliance checker on Lines 1 and 11 returns J such that $J \models P$ or J is empty. Lines 3 and 17 add J to B only if J is not empty nor already in B. Thus, if $R \in B$ then $R \subseteq L$ and $R \models P$.

Conversely, let $R \subseteq L$ and $R \models P$. Suppose to the contrary that $R \notin B$. In the SSgen algorithm, there are eight ways for R to not be in B. Lines 1 and 4-9 only consider sets containing subsets of U unioned with $L \setminus U$ for addition to B. Therefore, the first two cases that will cause R to not be in B are (1) R contains a non-empty proper subset of $L \setminus U$ and (2) $R \subset U$. Line 10 checks for sets that are not a superset nor a subset of a set in B and not a subset of a set in E. As a result, there are five more cases: (3) $R \subset$ a set in B, (4) $R \in B$, (5) $R \supset$ a set in B, (6) $R \subset$ a set in E, and (7) $R \in E$. Lines 1 and 11 invoke the compliance checker with a set and P and places the return value in J. Thus, another case is (8) R does not satisfy P. The pseudo-code in Line 13 adds sets that caused the compliance checker to return the empty set to E. By definition of a compliance checker, the empty set is only returned if the input set of credentials contain no subsets that satisfy the policy. Lines 3 and 17 add sets that are not empty nor in B to B. However, the cases that R is empty and $R \in B$ are already covered. Line 18 cause sets to not be considered if all the n-subsets contain no subsets that satisfy P. Nonetheless, the case that $R \subset$ a set in E is covered. Given the above eight cases, lets consider each case separately and show that each end in a contradiction.

Case 1: R contains a non-empty proper subset of $L \setminus U$. All minimal satisfying sets in R such that $R \cap (L \setminus U)$ is not empty and not equal to $L \setminus U$ are also in sets included in S. Therefore, any R that minimally satisfy P will be in B, which contradicts $R \notin B$.

Case 2: $R \subset U$. If R is a proper subset of U, then either $R \in B$, $R \supset$ a set in B, $R \subset$ a set in B, R is the empty set, or R is a mixture of credentials from different sets in B such that R is neither a superset nor subset of a set in B. $R \in B$ contradicts $R \notin B$. $R \supset$ a set in B contradicts $R \models P$. $R \subset$ a set in B contradicts $R \models P$. R is the empty set contradicts $R \models P$. All minimal satisfying sets in R such that R is a mixture of credentials from different sets in B such that R is neither a superset nor subset of a set in B are also in sets included in S. Therefore, any R that minimally satisfy P will be in B, which contradicts $R \notin B$.

Case 3: $R \subset F$, where F is a set in B. If a superset of R minimally satisfies P, then, by definition, R cannot satisfy P, which contradicts $R \models P$.

Case 4: $R \in B$ contradicts $R \notin B$.

Case 5: $R \supset D$, where D is a set in B. By the definition of a policy, if a subset of R minimally satisfies P then R cannot satisfy P because that contradicts the fact that a satisfying set contains the smallest number of credentials that minimally satisfy P. But this contradicts the fact that $R \models P$.

Case 6: $R \subset$ a set in E. If R is a proper subset of a set in E then R also does not satisfy P because otherwise the set would not be in E. But this contradicts the fact that $R \models P$.

Case 7: $R \in E$ contradicts $R \models P$.

Case 8: R does not satisfy P contradicts $R \models P$. $\square$

The complexity of the SSgen algorithm is $O(2^{|U|})$ where U is the union of the satisfying sets. Although this worst case performance is exponential in the size of U, the algorithm is practical when $|U|$ is small and the number of satisfying sets is modest.

Figure 3.5 shows performance characteristics of an implementation of SSgen in TrustBuilder. All experiments were run on a 1.4 GHz Pentium 4 processor with 512 MB of RAM. Each test case consisted of a set of local credentials L where $|L| = 50$. The size of U varies from 1 to 24 credentials.

For a single satisfying set, the algorithm determines that there is only one satisfying set in well under one second in all cases. In cases where there is only a single satisfying set, note that the complexity of the SSgen algorithm is $O(|U|)$.

The maximum number of satisfying sets for a given U is $\frac{n!}{(r!(n-r)!}$, where $n = |U|$ and $r = |U|/2$. This case results in the most invocations to the compliance checker. The experiments show that SSgen starts to become impractical at approximately 10 credentials. However, since for 10 credentials there will be 252 satisfying sets, it is unlikely that an access control policy will ever reach that level of complexity in practice.

Another significant data point is the case where there are U satisfying sets of one member each. This results in an exhaustive search of each lattice of size 1 to $|U|$ followed by a single invocation of the compliance checker. The graph shows that the running time is below one second as $|U|$ grows to 10 credentials, increases to 2 seconds as $|U|$ grows to 15 credentials, and then becomes impractical as $|U|$ increases beyond 15 credentials.

The graph also shows the performance when there are two satisfying sets. Each set is approximately three-fourths the size of U. The running time is below one second as $|U|$ grows to 16 credentials, increases to 3 seconds as $|U|$ grows to 19 credentials, and then becomes impractical as $|U|$ increases beyond 19 credentials.

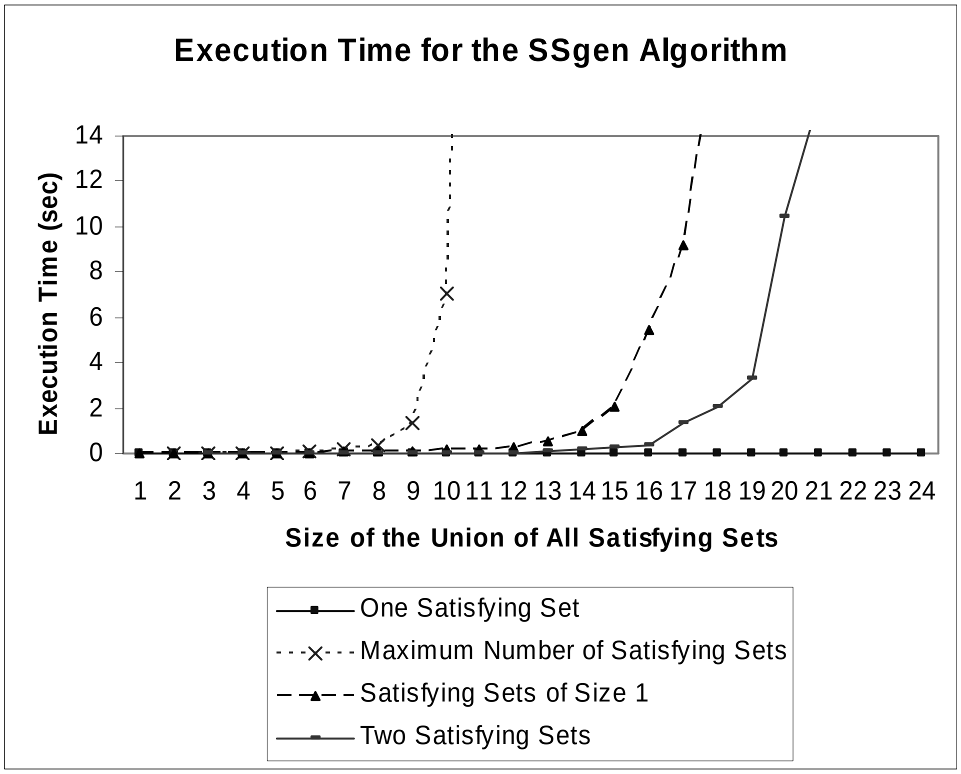

Figure 3.5: Execution time for the SSgen algorithm according to the size of the union of all satisfying sets and the number of satisfying sets.

Chapter 4 — Exploring Alternative Ways to Establish Trust

This chapter presents three ways to utilize a type-3 compliance checker so that alternative ways to establish trust can be explored during a trust negotiation.

First, when a policy is received during a negotiation, the compliance checker can immediately generate all the minimal satisfying sets. This allows the trust negotiation agent the flexibility to determine the order in which each set will be utilized to attempt to establish trust. The set ordering can be determined using heuristics, such as streamlining the negotiation or limiting the amount of sensitive information that is disclosed. For instance, the sets could be ordered by the number of sensitive credentials each set contains so that freely available sets will be considered first, possibly causing the negotiation to immediately succeed instead of pursuing additional rounds of negotiation to unlock sensitive credentials in another satisfying set. Another option is to prompt the user to provide guidance interactively regarding the order to explore satisfying sets based on the user's privacy preferences. Also, instead of considering each satisfying set individually, all the satisfying sets can be merged into a single set. This approach places a premium on reducing the number of rounds during a negotiation and is focused less on limiting the amount of information disclosed.

Second, the compliance checker may generate some of the sets that satisfy the policy, and then processes those sets using the same approaches that were discussed in the previous paragraph (e.g., order the sets according to sensitivity, prompt the user for guidance, and merge the sets). The reason for only generating some of the sets is to place limits on the amount of effort expended to generate the sets when there are many solutions. This can reduce the overall cost of the negotiation in case a successful negotiation can be reached without having to incur the cost to generate all of the sets. The amount of effort can be controlled by a threshold set on the amount of computation time expanded on set generation. Another method is to generate sets until the first set of non-sensitive credentials is returned, which could immediately lead to a successful negotiation without generating additional satisfying sets. If none of the sets leads to a successful negotiation, then the compliance checker must continue to generate additional satisfying sets in order for the trust negotiation to be complete.

Third, the compliance checker may generate one satisfying set at a time through an iterator-style interface. This approach avoids unnecessary generation of satisfying sets. However, the compliance checker dictates the order that satisfying sets are explored to establish trust. When a satisfying set leads to an unsuccessful negotiation, backtracking is employed to return to the previous point in the negotiation where the unsuccessful set was generated so that another attempt to generate a new satisfying set can be made.

Chapter 5 — Implementation

The two approaches to generating all satisfying sets and the three methods to explore alternative ways to establish trust are general purpose solutions that are suitable for integration into a wide range of applications and protocols. In order to demonstrate their feasibility for implementing a complete strategy for trust negotiation, we created a prototype for web applications. In the remainder of this chapter, we will discuss the major design issues encountered while developing the prototype.

5.1 TrustBuilder

The prototype augments the TrustBuilder [18] system, currently under development in the Internet Security Research Lab at BYU. The main component of TrustBuilder is the security agent, which mediates access to local resources (i.e., credentials, policies, or services). The security agent uses a strategy to determine what is disclosed and to accept disclosures from the other party. A method in the TrustBuilder interface allows policies to be passed for evaluation by its compliance engine and the strategy only discloses relevant credentials and policies. A tree structure allows backtracking, and two methods perform policy and credential set modification to obtain all satisfying sets.

5.1.1 TrustBuilder1-Relevant Strategy Implementation

We implemented a complete strategy that utilizes the received policies and limits disclosures to those relevant to the negotiation. The strategy is an implementation of TrustBuilder1-Relevant Strategy [19]. The only difference between our implementation and the TrustBuilder1-Relevant Strategy is our implementation does not make use of denial policies. Instead, we assume that if the credential or its governing policy is never disclosed that the other party either does not have the credential or will not disclose the credential under any circumstance.

The implementation of the TrustBuilder1-Relevant Strategy (Figure 5.1) takes the set of all received policies P, the set of all received credentials H, and the requested resource R, which can be null if the other party is not requesting a resource. In Figure 5.1, nR is initialized to contain an empty object, and F is initialized to be an empty set. H is then used to obtain the set A, which will contain all the roles relevant to R authenticated by the other negotiating party thus far in the negotiation. If a role G appears in the policy governing credential I, we say that G is relevant to I. Next R is checked to see if it is null. If it is not null, then the policy governing R is evaluated based on A. If the evaluation is true, then the authenticated attribute in nR is set to true meaning the access is authorized to the requested resource. If the evaluation is false, then Rs governing policy is placed in $nR.policiesToDisclose$. Each policy D in P is subsequently passed to the compliance checker along with the set of local credentials L with E being returned. All the credentials in E that are not already in F are added to F, which is the current set of relevant credentials. Afterward, all the credentials in F that are not already in S are added to S, which is the set of all relevant credentials. Next, H is used to obtain the authenticated roles relevant to the credentials in S, which is added to A. Finally, the policies governing the credentials in S are evaluated based on A. If the policy evaluates to true, then the credential that the policy governs is added to $nR.credsToDisclose$. Otherwise, the policy is added to $nR.policiesToDisclose$.

5.1.2 Negotiation Trees

Definition 5.1. A *negotiation tree* is a finite tree satisfying the following conditions:

1. The root corresponds to the requested resource.

```
P is the set of all received polices
H is the set of all received credentials
S is the set of all the relevant credentials
L is the set of all local credentials
R is the requested resource

Let F be initialized to an empty set
Let nR be initialized to an empty instance of NegotiationResponse
Let A be all the roles authenticated by H
if (R is not null)
    Let B be the policy governing R.
    if B evaluates to true based on A
        Set nR.authenticate to true
    if B evaluates to false based on A
        nR.policiesToDisclose = nR.policiesToDisclose $\cup$ B
For each policy, D, in P
    E = complianceChecker(L, D)
    if (E is not empty and $E \notin F$)
        $F = F \cup (E \setminus F)$
if (F is not empty and $F \notin S$)
    $S = S \cup (F \setminus S)$
For each credential, C, in S
    Let G be the policy governing C
    if G evaluates to true based on A
        nR.credsToDisclose = nR.credsToDisclose $\cup$ C
    if G evaluates to false based on A
        nR.policiesToDisclose = nR.policiesToDisclose $\cup$ G
return nR
```

Figure 5.1: Pseudo-code for the TrustBuilder1-Relevant Strategy implementation

2. All other nodes correspond to a policy and contain a set of satisfying credential sets for that policy.

All child nodes in a negotiation tree contain a policy that governs a credential contained in a credential set that satisfies the policy in the parent node. Figure 5.2 shows an example of a negotiation tree.

5.1.3 Backtracking Algorithm for Automated Trust Negotiation (BAAT)

In concert with using either of the two approaches for satisfying set generation, a depth-first search can be done to explore all possible negotiations—some of which are successful and some of which are unsuccessful. Upon reaching the end of an unsuccessful negotiation, BAAT (see Figure 5.3) receives the current negotiation tree and the last node added to the tree, which is referred to as the current node. If the current node is not the root of the tree, then a search is performed beginning at the current node. The search checks if the current node has been visited. If the

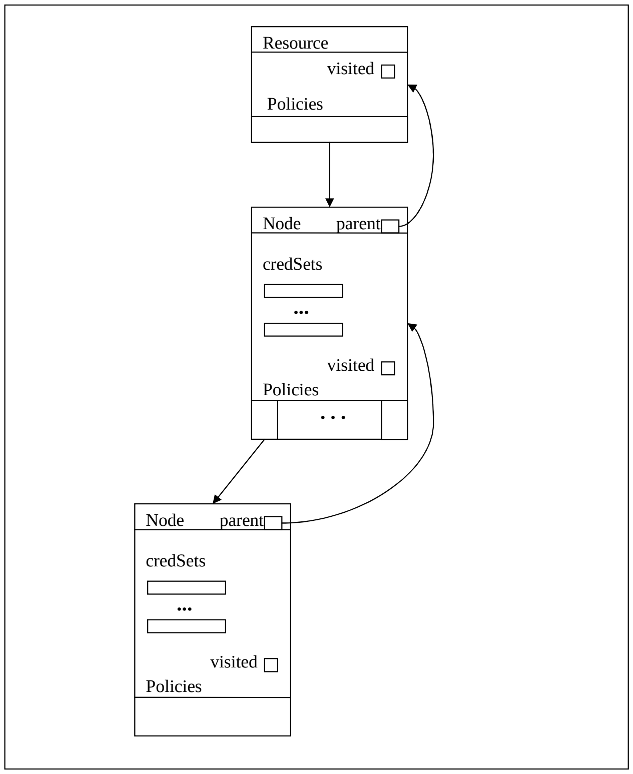

Figure 5.2: A negotiation tree in TrustBuilder

current node has been visited, the siblings are checked first and then the parent. This search continues until an unvisited node is found. If the search ends at the root of the tree, then an empty *NegotiationResponse* object is returned. Otherwise, the satisfying set generation algorithm is invoked with the policy represented by the node and the local credential set. If the satisfying set generation algorithm finds another satisfying set, then the new satisfying set is added to $satisfyingCredSet$. Finally, the policies governing disclosure of the credentials in the new satisfying credential set are evaluated. If a policy is satisfied, then that credential is placed into *negResponse*. Otherwise the policy is placed into *negResponse*. If the satisfying set generation algorithm does not find another satisfying set, then the node is mark visited and the backtracking algorithm is recursively invoked with the input negotiation tree and the visited node.

```
BAAT (negTree, currentNode)

Let L contain all local credentials.
Let negTree contain the current NegotiationTree
Let currentNode be the point at which backtracking will start
A node contains the set of credential sets that satisfy the policy represented by the node in
  satisfyingCredSet

Let negResponse contain an empty NegotiationResponse
If (currentNode != the root node)
    Do a breadth-first search of negTree starting at currentNode until non-visited node is
        found.
    Let N contain the node that was stopped at during the breadth-first search.
    If (N != the root node)
        Let credSet contain the satisfyingCredSet of N
        Let P contain the policy represented by N
        S = satisfyingSetGeneration(P, L)
        If (S ∉ credSet)
          satisfyingCredSet = satisfyingCredSet ∪ {S}
          For each credential, C, in S
              Evaluate the access control policy based on the authenticated roles of other party.
              If evaluation is true
                Place C in negResponse.credsToDisclose
              else
                Place the policy of C in negResponse.policiesToDisclose
        else
          Set N.visited to true
          BAAT (negTree, N)
Return negResponse
```

Figure 5.3: BAAT pseudo-code

5.1.4 Policies in Minimal DNF

Definition 3.1 defined a policy as a disjunction of conjunctions of minimal satisfying sets. In order for the SSgen algorithm to function properly, the policy must be in minimal disjunctive normal form (mDNF). TrustBuilder represents access control policies that govern resources as Boolean expressions with roles as operands (e.g., role expressions). The credentials required for role membership are specified in IBM's Trust Policy Language (TPL) [9]. Since a TPL policy may not be in mDNF, TrustBuilder supports policy conversion into mDNF as follows. First, the role expression is modified by replacing each role name with a credential expression that specifies the credentials that are required to authenticate to that role, according to the TPL

```
 <GROUP NAME= "A" >
 <RULE>
   <INCLUSION FROM="C" ID="cert1" TYPE="F" />
 </RULE>
</GROUP>

 <GROUP NAME= "B" >
 <RULE>
   <INCLUSION FROM="D" ID="cert2" TYPE="G" />
 </RULE>
 <RULE>
   <INCLUSION FROM="E" ID="cert3" TYPE="H" />
 </RULE>
 </GROUP>
```

Figure 5.4: TPL role definition for role A

policy. Second, the resulting credential expression is converted to mDNF using Quine-McClusky minimization [12, 14]. Third, a new TPL policy is constructed according to the mDNF expression. Finally, the governing policy is revised to correspond to the new TPL policy. The following example illustrates policy conversion in TrustBuilder. Suppose the policy governing a resource is the role expression A AND B. The TPL policy specifying the membership requirements for role A and role B is shown in Figure 5.4.

Substituting the corresponding credential expression for each role in the role expression yields cert1 AND (cert2 OR cert3), which is not in mDNF. In this case, putting the credential expression into mDNF is as simple as distributing the AND over the OR, which produces the expression (cert1 AND cert2) OR (cert1 AND cert3). The TPL policy in mDNF corresponding to this expression is shown in Figure 5.5. Based on this new TPL policy, the new access control policy for the resource is G1.

5.2 HTTP

Web browsers use HTTP to communicate with a web server. HTTP is a client-server protocol running over TCP/IP, which follows a request-reply paradigm in which

```
<GROUP NAME= "G1" >
<RULE>
  <INCLUSION FROM="C" ID="cert1" TYPE="F" />
  <INCLUSION FROM="D" ID="cert2" TYPE="G" />
</RULE>
<RULE>
  <INCLUSION FROM="C" ID="cert3" TYPE="F" />
  <INCLUSION FROM="E" ID="cert4" TYPE="H" />
</RULE>
</GROUP>
```

Figure 5.5: Modified TPL rol definition in minimal DNF for role A

the client always initiates a transaction. The HTTP specification supports two forms of authentication: `Basic` and `Digest`. To support trust negotiation, an authentication header was defined of type `TrustNegotiation` [10] to transport credentials and policies in an HTTP message. In addition to creating a new authentication type, the web browser must be able to support trust negotiation. This can be done in several ways. One way uses a proxy server [10] that can intercept HTTP messages and pass messages to TrustBuilder if they are related to trust negotiation. By using a proxy server, the web browser does not have to be aware of TrustBuilder. An overview of this architecture is illustrated in Figure 5.6.

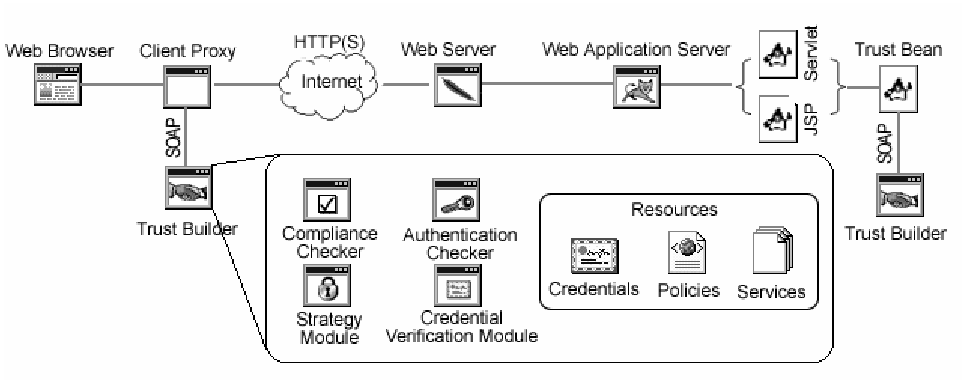

Figure 5.6: HTTP prototype architecture

Chapter 6 — Related Work

There are only a small number of compliance checkers that meet the requirement of returning the set of credentials that satisfy the policy passed to the compliance checker. The compliance checkers that are able to meet this requirement are: Rule-controlled Environment For Evaluation of Rules, and Everything Else (REFEREE) [7], Portfolio and Service Protection Language (PSPL) [6], Trust Establishment (TE) [9], Role-based Trust management (RT) [13] and χ-TNL [2].

Chu et al. [7] present a general purpose execution environment for Web applications that require trust called REFEREE. Policies are treated as programs, which can be invoked and return an answer with justification. The justification is a statement or list of statements. A statement consists of two elements; the context or source of the statement and the content of the statement. A statement can be viewed as an attribute credential with the context being the issuer if the content contains some form of subject identification and states at least one attribute about the subject. It is unclear whether REFEREE returns all the sets of credentials that satisfy the policy. It also has a prototype implementation.

Bonatti and Samarati [6] present a uniform formal framework for specifying service access and information disclosure control. This framework was developed with trust negotiation in mind, so it meets the majority of the requirements of trust negotiation including a compliance checker that returns all the possible satisfying sets of credentials. Currently, no prototype implementation is available.

Herzberg et al. [9] present the Trust Policy Language, an XML-based language used primarily to map users to roles. The language is supported in the Trust Establishment (TE) system, which can determine a user's role based on certificates and TPL policies. TE is an example of a compliance checker that returns only one satisfying set. We incorporated TE in our TrustBuilder prototype because it supported many features needed in trust negotiation. The algorithms presented in this thesis effectively transform the TE compliance checker into a type-3 compliance checker.

Li et al. [13] introduce RT, a family of Role-based Trust management languages for representing credential and policies in distributed authorization. An initial prototype implementation of trust negotiation based on the first RT language provides completeness during trust negotiation for unstructured credentials [16] and is an example of a type-3 compliance checker. The runtime engine is still under development to support the remaining languages in the RT family. We plan to explore the suitability of RT as a decision engine for TrustBuilder.

Bertino et al. [2] present X-TNL, an XML-based language for conducting trust negotiation. X-TNL provides a medium to transport information about the negotiating parties called a certificate. A certificate can be either a credential or a declaration. A credential is list of properties of a negotiating party certified by a Certificate Authority. A declaration contains helpful information (e.g., policies) for the negotiation process. X-TNL is part of a more extensive project called Trust-X, which extended from X-Sec [1]. X-Sec had no need to generate all satisfying credential sets, but the development of X-TNL introduces this need.

Chapter 7 — Conclusion and Future Work

Access control policies can be exchanged between the participants in an on-line trust negotiation to inform each other of the requirements for establishing trust. A compliance checker is a runtime engine that answers the question of whether or not a set of credentials satisfies an access control policy. When a policy is received at runtime, a compliance checker determines which credentials satisfy the policy so they can be disclosed. In situations where several combinations of credentials satisfy a policy and some of the credentials are sensitive, a compliance checker that generates all the combinations is necessary to insure that the negotiation is complete (i.e., succeeds whenever possible). Compliance checkers designed for trust management do not usually generate all the ways a policy is satisfied.

7.1 Contributions

In this thesis, we presented two practical algorithms for generating all credential combinations that satisfy a policy given a compliance checker that generates only one combination. The policy modification approach requires rewriting the policy every time the compliance checker is invoked so that prior solutions are excluded in order to force additional solutions to be generated. The credential set modification approach involves modifying the set of input credentials each time the compliance checker is invoked in order to obtain all minimal satisfying sets.

The policy modification approach requires that an implementation be able to interpret and manipulate policies according to the semantics of the policy language. The credential set modification approach is policy language independent, although it requires policies to be expressed in minimal disjunctive normal form.

We incorporated these approaches in TrustBuilder, our prototype system for trust negotiation. TrustBuilder uses the compliance checker provided by the IBM Trust Establishment system that was designed for trust management. The extensions to TrustBuilder demonstrate one way to adapt existing compliance checkers to meet the needs of trust negotiation. The reason all satisfying sets need to be generated is because some of the sets may not lead to a successful negotiation. One reason failure can occur is that one or more of the credentials in a set is sensitive, and the other party is not authorized to receive them. If the sensitive credentials belong to another satisfying set, another failure will occur. Thus, one optimization in the search for additional satisfying sets is to remove from consideration the sensitive credentials that the other party is not authorized to receive. This will cut the search space in half for each sensitive credential.

Once a compliance checker is able to produce all the minimal satisfying sets, there are several ways to use that feature during trust negotiation. Generating some or all of the satisfying sets at once allows the negotiation agent to select the order that alternative ways of establishing trust are considered. We introduced several alternatives and discussed the merits of each approach. These ideas have also been implemented in TrustBuilder. To our knowledge, this is the first example of a trust negotiation system that generates potential solutions to establishing trust and prioritizes them according to specific criteria. These ideas can be incorporated directly into the type-3 compliance checkers that are being implemented to support trust negotiation.

7.2 Future Work

Disclosing access control policies during trust negotiation raises privacy issues, because policies themselves can contain sensitive information. Hidden credentials [11] address this problem by encrypting a policy so that it can only be understood if it is fulfilled. We are currently exploring ways to efficiently process these encrypted policies.

We plan to experiment with emerging compliance checker implementations designed for trust negotiation, including the RT runtime system [13] and the PeerTrust system [8] designed for trust negotiation in the Semantic Web.

Bibliography

[1] Elisa Bertino, Silvana Castano, and Elena Ferrari. On specifying security policies for web documents with an XML-based language. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, pages 57–65, Chantilly, VA, May 2001. ACM Press.

[2] Elisa Bertino, Elena Ferrari, and Anna Squicciarini. X-TNL: An XML-based language for trust negotiation. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 81–84, Como, Italy, June 2003. IEEE Computer Society Press.

[3] Matt Blaze, Joan Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote Trust Management System Version 2. In *Internet Draft RFC 2704*, 1999.

[4] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Oakland, CA, May 1996. IEEE Computer Society Press.

[5] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance-checking in the PolicyMaker trust management system. In *Proceedings of the 2nd International Conference on Financial Cryptography (FC'98)*, volume 1465 of *Lecture Notes in Computer Science*, pages 254–274, Anguilla, British West Indies, 1998. Springer-Verlag.

[6] Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *Proceedings of the 7th ACM Conference on Computer*

and Communications Security (CCS-7), pages 134–143, Athens, Greece, November 2000. ACM Press.

[7] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for Web applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.

[8] Rita Gavriloaie, Wolfgang Nejdl, Daniel Olmedilla, Kent E. Seamons, and Marianne Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the Semantic Web. In *Proceedings of the 1st European Semantic Web Symposium*, Heraklion, Greece, May 2004.

[9] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 2–14, Oakland, CA, May 2000. IEEE Computer Society Press.

[10] Adam Hess and Kent E. Seamons. An access control model for dynamic client content. In *Proceedings of the 8th ACM Symposium on Access Control Models and Technologies*, pages 207–216, Como, Italy, June 2003. ACM Press.

[11] Jason Holt, Robert Bradshaw, Kent E. Seamons, and Hillarie Orman. Hidden credentials. In *Proceedings of the 2nd ACM Workshop on Privacy and Electronic Society*, pages 1–8, Washington, DC, October 2003. ACM Press.

[12] Edward L. McCluskey Jr. Minimization of boolean functions. *Bell System Techinical Journal*, 35(6):1417–1444, November 1956.

[13] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Sympo-*

sium on Security and Privacy, pages 114–130, Oakland, CA, May 2002. IEEE Computer Society Press.

[14] Willard Van Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59(8):521–531, October 1952.

[15] Kent E. Seamons, Marianne Winslett, Ting Yu, Bryan Smith, Evan Child, Jared Jacobsen, Hyrum Mills, and Lina Yu. Requirements for policy languages for trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (Policy 2002)*, pages 68–79, Monterey, CA, June 2002. IEEE Computer Society Press.

[16] William H. Winsborough and Ninghui Li. Towards practical automated trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (Policy 2002)*, pages 92–103, Monterey, CA, June 2002. IEEE Computer Society Press.

[17] William H. Winsborough, Kent E. Seamons, and Vicki E. Jones. Automated trust negotiation. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, volume I, pages 88–102, Hilton Head, SC, January 2000. IEEE Press.

[18] Marianne Winslett, Ting Yu, Kent E. Seamons, Adam Hess, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating Trust on the Web. *IEEE Internet Computing Special Issue on Trust Management*, 6(6):30–37, November/December 2002.

[19] Ting Yu, Marianne Winslett, and Kent E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies in automated trust

negotiation. *ACM Transactions on Information and System Security*, 6(1):1–42, February 2003.

Appendix A — UML Diagrams

1. Use case for trust negotiation with policy exchange
2. Sequence diagram for TrustBuilder
3. Class diagram for TrustBuilder

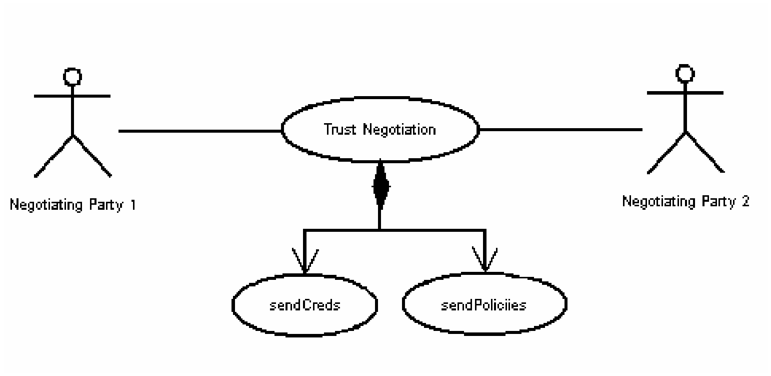

Figure A.1: Use case for trust negotiation with policy exchange

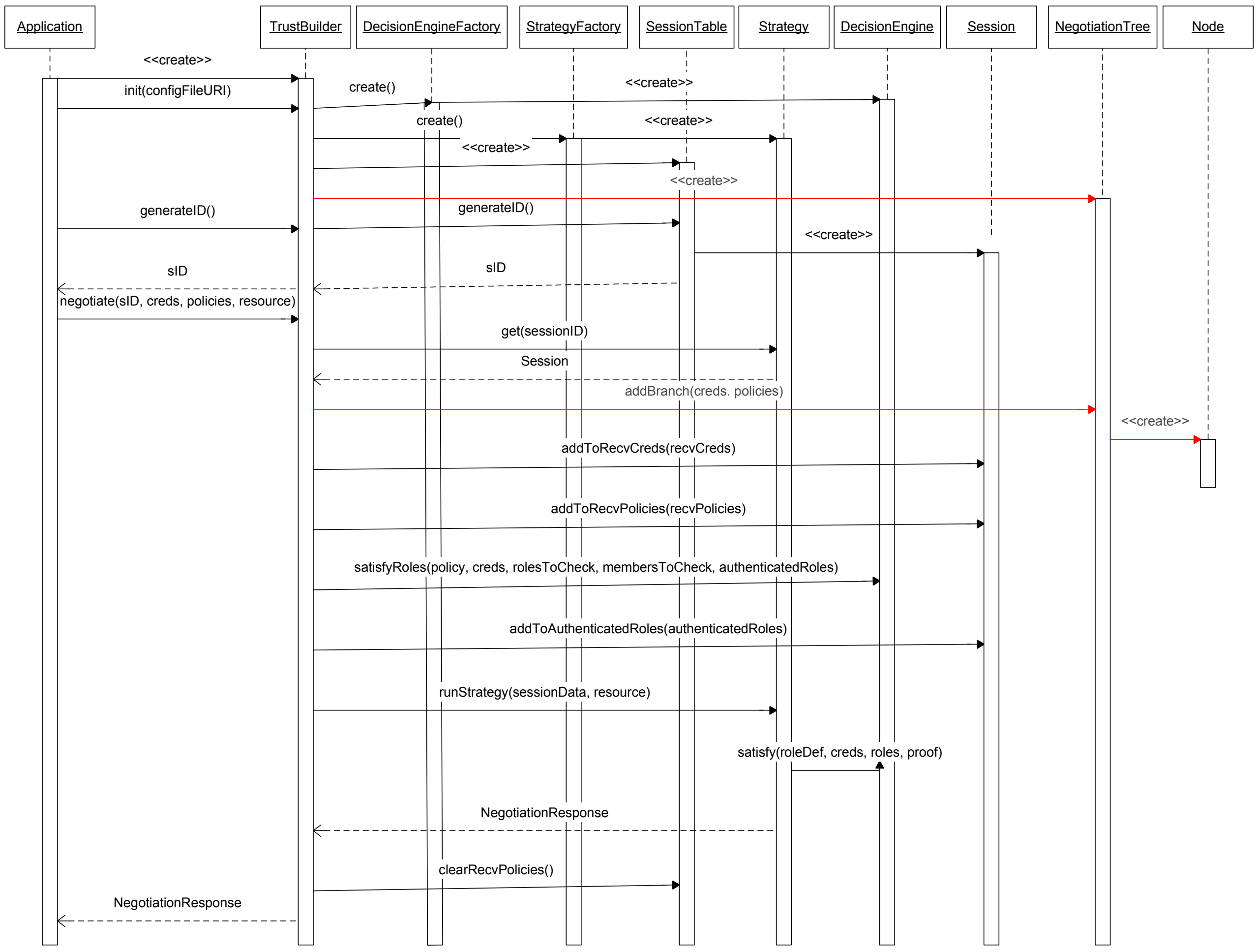

Figure A.2: Sequence diagram for TrustBuilder

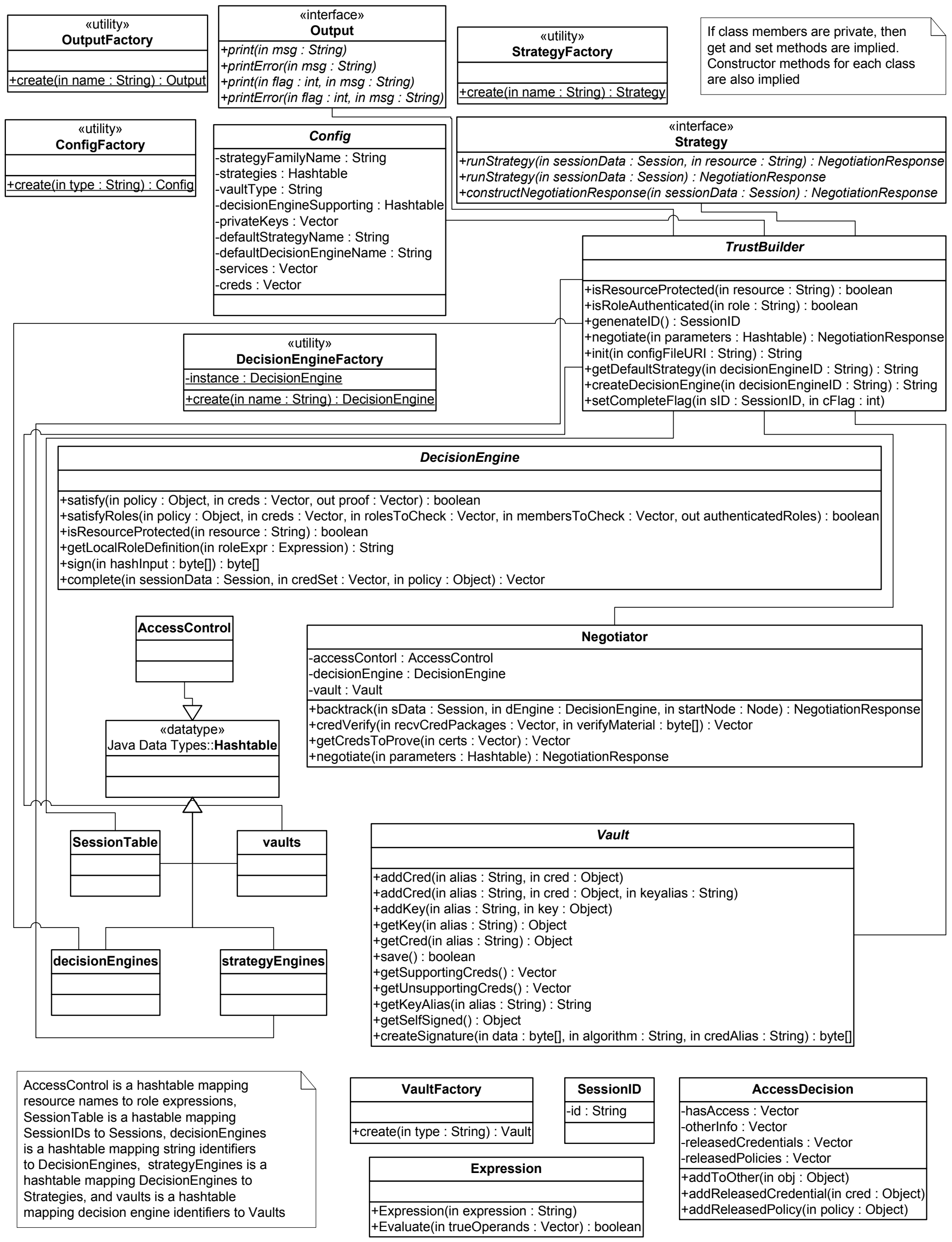
«utility» OutputFactory
+create(in name : String) : Output
«interface» Output
+print(in msg : String)
+printError(in msg : String)
+print(in flag : int, in msg : String)
+printError(in flag : int, in msg : String)
«utility» StrategyFactory
+create(in name : String) : Strategy
If class members are private, then get and set methods are implied. Constructor methods for each class are also implied
«utility» ConfigFactory
+create(in type : String) : Config
Config
-strategyFamilyName : String
-strategies : Hashtable
-vaultType : String
-decisionEngineSupporting : Hashtable
-privateKeys : Vector
-defaultStrategyName : String
-defaultDecisionEngineName : String
-services : Vector
-creds : Vector
«interface» Strategy
+runStrategy(in sessionData : Session, in resource : String) : NegotiationResponse
+runStrategy(in sessionData : Session) : NegotiationResponse
+constructNegotiationResponse(in sessionData : Session) : NegotiationResponse
TrustBuilder
+isResourceProtected(in resource : String) : boolean
+isRoleAuthenticated(in role : String) : boolean
+genenateID() : SessionID
+negotiate(in parameters : Hashtable) : NegotiationResponse
+init(in configFileURI : String) : String
+getDefaultStrategy(in decisionEngineID : String) : String
+createDecisionEngine(in decisionEngineID : String) : String
+setCompleteFlag(in sID : SessionID, in cFlag : int)
«utility» DecisionEngineFactory
-instance : DecisionEngine
+create(in name : String) : DecisionEngine
DecisionEngine
+satisfy(in policy : Object, in creds : Vector, out proof : Vector) : boolean
+satisfyRoles(in policy : Object, in creds : Vector, in rolesToCheck : Vector, in membersToCheck : Vector, out authenticatedRoles) : boolean
+isResourceProtected(in resource : String) : boolean
+getLocalRoleDefinition(in roleExpr : Expression) : String
+sign(in hashInput : byte[]) : byte[]
+complete(in sessionData : Session, in credSet : Vector, in policy : Object) : Vector
AccessControl
Negotiator
-accessContorl : AccessControl
-decisionEngine : DecisionEngine
-vault : Vault
+backtrack(in sData : Session, in dEngine : DecisionEngine, in startNode : Node) : NegotiationResponse
+credVerify(in recvCredPackages : Vector, in verifyMaterial : byte[]) : Vector
+getCredsToProve(in certs : Vector) : Vector
+negotiate(in parameters : Hashtable) : NegotiationResponse
«datatype» Java Data Types::Hashtable
SessionTable
vaults
Vault
+addCred(in alias : String, in cred : Object)
+addCred(in alias : String, in cred : Object, in keyalias : String)
+addKey(in alias : String, in key : Object)
+getKey(in alias : String) : Object
+getCred(in alias : String) : Object
+save() : boolean
+getSupportingCreds() : Vector
+getUnsupportingCreds() : Vector
+getKeyAlias(in alias : String) : String
+getSelfSigned() : Object
+createSignature(in data : byte[], in algorithm : String, in credAlias : String) : byte[]
decisionEngines
strategyEngines
AccessControl is a hashtable mapping resource names to role expressions, SessionTable is a hastable mapping SessionIDs to Sessions, decisionEngines is a hashtable mapping string identifiers to DecisionEngines, strategyEngines is a hashtable mapping DecisionEngines to Strategies, and vaults is a hashtable mapping decision engine identifiers to Vaults
VaultFactory
+create(in type : String) : Vault
SessionID
-id : String
AccessDecision
-hasAccess : Vector
-otherInfo : Vector
-releasedCredentials : Vector
-releasedPolicies : Vector
+addToOther(in obj : Object)
+addReleasedCredential(in cred : Object)
+addReleasedPolicy(in policy : Object)
Expression
+Expression(in expression : String)
+Evaluate(in trueOperands : Vector) : boolean

Figure A.3: Class diagram for TrustBuilder