Memory Complexity of Parsimonious Strategy in Automated Trust Negotiation

Indika H. Katugampala  Graduate School of Information Science, Nagoya University
indika@net.itc.nagoya-u.ac.jp
Hirofumi Yamaki  Information Technology Center, Nagoya University
yamaki@itc.nagoya-u.ac.jp
Yukiko Yamaguchi (affiliation as previous author)
yamaguchi@itc.nagoya-u.ac.jp

keywords: Automated Trust Negotiation, Parsimonious strategy, memory complexity

Summary

Automated Trust Negotiation (ATN) has been proposed as a mechanism to establish mutual trust among strangers. While existing fundamental protocols and strategies are shown, this paper focuses on Parsimonious strategy. The most straightforward implementation of Parsimonious strategy has a very high memory consumption which may be problematic when it is used in real world environments. This paper proposes an implementation which keeps all requests in Disjunctive Normal Form (DNF) and further reduces its memory consumption by exploiting the history of the negotiation, while keeping the completeness of the strategy intact. In addition to that, proposed method provides a criterion to detect negotiation failures. Results obtained by means of simulations showed that the method proposed is effective in achieving its goals, without increasing the overall computational overhead. Theoretical analysis of the proposed method is also presented.

1. Introduction

With advancement of the Internet, the number of services that users may use has increased dramatically. However, a major problem when using such services is to decide whether the service provider is trustworthy. Furthermore, service providers may also wish to restrict their services only to trustworthy users. As a solution to this issue, Automated Trust Negotiation (ATN) has been proposed to establish mutual trust among strangers [Winsborough 99, Winsborough 00, Winsborough 02, Winslett 02, Yu 03].

Each entity participating in ATN has digital credentials that certify attributes of the entity. Furthermore, each credential has a policy that defines the prerequisites the other entity must fulfill. During the negotiation, entities exchange credentials according to their policies. If the policy of the service is satisfied during the negotiation, user will get access to the service.

Consider the following exemplary scenario depicted in Figure 1. ABC Computers Inc., who manufactures personal computers and music players, is running a campaign which offers a discount to students who buy a personal computer and a music player at the same time. To qualify for the discount, one should prove that s/he is currently a student at some educational institution by submitting the student’s certificate. ABC Computers Inc. is a manufacturer certified by National Trade Commission who possesses a trade permission certificate. ABC Computers Inc. is ready to disclose the trade permission certificate to anyone who wishes to buy its products.

Bob, who is a university student, has a student’s certificate issued by his university, but does not wish to disclose it to untrusted parties. He considers anyone who can certify his/her identity by a credential issued by a government agency as a trusted party.

Suppose that Bob wants to be eligible for the discount. Under aforementioned circumstances, one way to establish trust is as follows. When Bob asks for the discount, discount service requests him to submit his student’s certificate. Since Bob discloses his student’s certificate only to parties certified by a government agency, he asks for any such credential issued to the service. Since the service possesses a digital certificate issued by National Trade
Commission and service’s policy allows it to be disclosed without any prerequisites, service discloses it at this stage. After Bob has verified the server’s certificate, he discloses his student’s certificate. Discount server verifies Bob’s student’s certificate and allows him to apply for the discount.

As in the above example, during the ATN process, entities participating in the negotiation exchange their policies and/or credentials, and establish mutual trust.

2. Automated Trust Negotiation

2.1 Formal Definition

Throughout this paper, the same notation presented in [Yu 00] is used. We denote the user entity as client, the service entity as server and the target of the negotiation as service. We use the propositional symbols to denote the service, credentials of the client, and credentials of the server. Accordingly, we denote the service by \( S \), credentials of the client by \( C_1, \ldots, C_{n_c} \), credentials of the server by \( S_1, \ldots, S_{n_s} \), where \( n_c \) is the number of credentials possessed by the client and \( n_s \) is the number of credentials possessed by the server. Policy for disclosing credential \( C \) is denoted by \( C \leftarrow F_C(S_1, \ldots, S_k) \), where \( F_C(S_1, \ldots, S_k) \) is an expression involving credentials \( S_1, \ldots, S_k \) possessed by the other entity, boolean operators \( \land, \lor, \) and parentheses as needed. We say the policy of credential \( C \) is satisfied, if the logical expression \( F_C(S_1, \ldots, S_k) \) evaluates to true after substituting propositional symbols of already disclosed credentials by the other entity in disjuncts \( D_1, \ldots, D_l \).

If credential \( C \) can be disclosed without any credential from the other entity, we denote that policy by \( C \leftarrow true \), and such credential \( C \) is called an unprotected credential. On the other hand, if credential \( C \) can not be disclosed under any circumstances, we denote such a policy by \( C \leftarrow false \).

Even though the service is not a credential, we may denote the access control policy of the service using the same notation mentioned above. As a result of the negotiation, if the policy of the service becomes satisfied, the service will be available to the client, and we say the negotiation is a successful negotiation. Otherwise, we say the negotiation is a failed negotiation.

2.2 Automated Trust Negotiation Strategies

Several negotiation strategies that can be used in ATN have been proposed. The first of its kind, namely Eager strategy and Parsimonious strategy, was proposed in [Winsborough 00]. In Eager strategy, client and server exchange currently disclosable credentials. As the negotiation proceeds, more credentials become disclosable. The client terminates the negotiation if either no new credentials were disclosed by the server or no new credentials became disclosable by newly disclosed credentials from the server. Eager strategy is complete, and it terminates when a successful negotiation is impossible. Maximum length of a negotiation which uses Eager strategy is bounded by the value \( 2\min(n_c, n_s) \).

On the other hand, in Parsimonious strategy, entities first exchange policies and decide whether a successful negotiation exists. Only if it is possible to have a successful negotiation, they start disclosing credentials. Parsimonious strategy is the main focus of this paper and more
details are presented in Chapter 3.

PRUNES [Yu 00] is another strategy for ATN. It is a refined version of complete brute-force backtracking negotiation strategy, and it uses the knowledge of the negotiation gathered so far to prune certain areas of the search space, while keeping the completeness of the strategy intact.

Contrary to above strategies that do not consider user preferences, Dynamic programming approach [Yamaki 05] assigns a cost – which is a positive real number – to each credential and policy to reflect user preference. It calculates a solution with the minimum cost using dynamic programming based on the assumption that both entities are cooperative. Since the solution has the minimum total cost, it assures that no unnecessary credentials or policies are disclosed. One major drawback of this strategy is that it assumes the policies of two parties have no confluences, i.e., there are no policies such that $C_1 \leftarrow S_1 \land S_2$, $S_1 \leftarrow C_2$, $S_2 \leftarrow C_2$, where $C_1, C_2$ are credentials of one entity and $S_1, S_2$ are credentials of the other entity.

Another strategy which takes in to account user preferences is proposed in [Nakatuka 06]. It uses AO$^*$ algorithm and exploits the heuristics of the problem domain to achieve better performance. It assigns a cost to each credential and executes AO$^*$ to find a solution with the minimum cost. This strategy also assumes the cooperativeness of entities.

3. Parsimonious Strategy

In parsimonious strategy [Winsborough 00], first the client requests the service and if the service is anonymous, the service is granted. If not, the server sends the access control policy of the service as the reply. If that policy can not be satisfied by the client, it generates a new counter request, which must be satisfied by the server, for the client to satisfy the server’s policy. This process proceeds until a request that can be satisfied by one entity using its unprotected credentials appears. When such situation occurs, the entity which can satisfy other entity’s counter request, starts to disclose credentials. With the disclosable credentials, it also sends the counter request that it already requested in the previous step. Likewise the previously requested counter requests are replayed with the credentials that satisfy the counter request of the other entity, until access is granted to the service eventually.

The strength of Parsimonious strategy is that it does not disclose any unnecessary credentials. Moreover, in situations where no successful negotiation is possible, it does not disclose any credentials at all.

<table>
<thead>
<tr>
<th>Server Policy</th>
<th>Client Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \leftarrow C_1 \lor (C_2 \land C_3)$</td>
<td>$C_1 \leftarrow S_1 \lor S_2$</td>
</tr>
<tr>
<td>$S_1 \leftarrow C_2 \lor C_4$</td>
<td>$C_2 \leftarrow S_5$</td>
</tr>
<tr>
<td>$S_2 \leftarrow C_2 \land C_3$</td>
<td>$C_3 \leftarrow S_3 \land S_4$</td>
</tr>
<tr>
<td>$S_3 \leftarrow \text{true}$</td>
<td>$C_4 \leftarrow S_3 \land S_4$</td>
</tr>
<tr>
<td>$S_4 \leftarrow \text{true}$</td>
<td>$C_5 \leftarrow S_2 \land S_3$</td>
</tr>
<tr>
<td>$S_5 \leftarrow C_5$</td>
<td></td>
</tr>
</tbody>
</table>

3.1 Straightforward Implementation of Parsimonious Strategy

While theoretical aspects of Parsimonious strategy were presented in [Winsborough 00], not much has been mentioned about implementing it. A straightforward way to implement Parsimonious strategy would be to simply substitute opponent’s request by its policies. To explain the straightforward implementation, consider the policy set in Table 1. First few steps of the straightforward implementation would be as follows. Negotiation starts with the client’s request for the service $S$, and the server responds with the counter request $C_1 \lor (C_2 \land C_3)$. Since this can not be satisfied by the client, it generates a new counter request by substituting its policies of $C_1, C_2$, and $C_3$ into the server’s request. Therefore, client’s counter request becomes $(S_1 \lor S_2) \lor ((S_5) \land (S_1 \land S_4))$. Likewise negotiation continues until a request that can be satisfied appears or until a negotiation failure is detected.

Even though the above method is easy to implement, the size of counter requests increase exponentially as the negotiation continues. To reduce this high memory consumption, it is desirable to remove unnecessary elements in counter requests while keeping their logical meanings intact. In the next section, we propose one such method.

3.2 DNF based Implementation of Parsimonious Strategy

This section presents an implementation of Parsimonious strategy, which keeps all counter requests in DNF. Given below are the steps involved in a negotiation which uses DNF based implementation for policies shown in Table 1.

1. Client initiates the negotiation by sending a request to the service $S$.
2. Since the service $S$ is not anonymous, server asks client to satisfy policy $C_1 \lor (C_2 \land C_3)$.
3. Because client can not satisfy server’s request, it generates a counter request by substituting its policies into server’s request and converting it to DNF. Client’s counter request at this step becomes $S_1 \lor S_2 \lor (S_5 \land S_3 \land S_4)$. 
Memory Complexity of Parsimonious Strategy in Automated Trust Negotiation

(4) Since client’s request can not be satisfied, sever requests \( C_2 \lor C_4 \lor (C_2 \land C_3) \lor C_5 \).

(5) Client requests \( S_5 \lor (S_3 \land S_4) \lor (S_5 \land S_3 \land S_4) \lor (S_2 \land S_3) \).

(6) Since this request can be satisfied by the server, it discloses \( S_3, S_4 \) and asks for \( C_2 \lor C_4 \lor (C_2 \land C_3) \lor C_5 \) – the same counter request it requested at step 4.

(7) Client discloses \( C_4 \) and requests \( S_1 \lor S_2 \lor (S_3 \land S_4) \).

(8) Sever discloses \( S_1 \) and requests \( C_1 \lor (C_2 \land C_3) \).

(9) Client discloses \( C_1 \) and requests permission for the service \( S \).

(10) Server issues a ticket for service \( S \).

Once counter requests are maintained in DNF, it is easy to remove unnecessary parts of counter requests as shown below. Following definitions are used while describing how this can be achieved.

**Definition 1.** stricter disjunct

Consider two disjuncts \( t_1 \) and \( t_2 \). If \( t_1 \lor t_2 = t_1 \), then we say that \( t_2 \) is stricter than \( t_1 \) and denote that relationship by \( t_2 \preceq t_1 \).

**Definition 2.** independent disjuncts, dependent disjuncts

If neither \( t_1 \) is stricter than \( t_2 \) nor \( t_2 \) is stricter than \( t_1 \), we say that \( t_1 \) and \( t_2 \) are independent disjuncts. If \( t_1, t_2 \) are not independent disjuncts, they are dependent disjuncts.

The following theorem uses above definitions to present a method to reduce the number of disjuncts in a counter request.

**Theorem 1.** Assume disjuncts \( t_1, t_2 \) appear in a counter request such that \( t_1 \) is stricter than \( t_2 \). Then removing disjunct \( t_1 \) from the counter request does not affect the completeness of Parsimonious strategy.

Outline of the proof is as follows. We consider two counter requests \( R, R' \), where \( R' \) is derived by removing \( t_1 \) from \( R \). First we consider the case that a disjunct other than \( t_1 \) leads to a successful negotiation and show that \( R' \) also leads to a successful negotiation.

Then we consider the case that \( t_1 \) leads to a successful negotiation and prove that \( R' \) also leads to a successful negotiation by using the mathematical induction on \( k \): the number of counter requests created by both parties before a satisfiable counter request appears.

Complete proof is given below.

**Proof.** Let’s denote the counter request before removing \( t_1 \) by \( R \), and the counter request after removing \( t_1 \) by \( R' \). Assume that a disjunct other than \( t_1 \) in \( R \) leads to a successful negotiation. Since the same disjunct appears in \( R' \), \( R' \) also leads to a successful negotiation.

Now consider the case that \( t_1 \) leads to a successful negotiation. We prove that the theorem holds by mathematical induction on \( k \): the number of counter requests created by two entities before a satisfiable counter request appears.

When \( k = 0 \), it means that \( R \) can be satisfied by the other entity without making any counter requests. In other words, \( t_1 \) in \( R \) evaluates to true at the other entity. Since \( t_1 \) is stricter than \( t_2 \), \( t_1 \lor t_2 = t_2 \) (by definition) and \( t_1 = true \) implies that \( t_2 = true \). Which means \( R' \) also evaluates to true and leads to a successful negotiation. Therefore, the result holds when \( k = 0 \).

Assume that the theorem holds for \( k = 0, \ldots, p \in \mathbb{N}^+ \). Now consider the case where \( k = p + 1 \). Also assume that disjunct \( t_i = C_{t,1} \land \ldots \land C_{t,m_i} \) (where \( C_{t,1}, \ldots, C_{t,m_i} \) are credentials of the opponent entity) and opponent entity’s policy for \( C_{t,j} \) is \( C_{t,j} \leftarrow \tau_{j,1} \lor \ldots \lor \tau_{j,q} \). When creating a counter request at the other entity, disjunct \( t_i \) is expanded as follows.

\[
t_i = C_{t,1} \land C_{t,2} \land \ldots \land C_{t,m_i} \\
= (\tau_{i,1} \lor \tau_{i,2} \lor \ldots \lor \tau_{i,qj}) \land \tau_{m_i,1} \lor \tau_{m_i,2} \lor \ldots \lor \tau_{m_i,qm_i} \\
= (\tau_{1,1} \lor \tau_{2,1} \lor \ldots \lor \tau_{m_i,1}) \land (\tau_{1,2} \lor \tau_{2,2} \lor \ldots \lor \tau_{m_i,2}) \lor \ldots \lor (\tau_{1,q} \lor \tau_{2,q} \lor \ldots \lor \tau_{m_i,qm_i})
\]

Accordingly, request \( R = t_1 \lor t_2 \lor \ldots \lor t_n \) becomes,

\[
R = \bigvee_{1 \leq r_1 \leq q_1} \bigvee_{1 \leq r_m \leq q_m, 1 \leq q_m \leq m} \left( \tau_{1,1} \land \tau_{2,1} \land \ldots \land \tau_{m_1,1} \right) \lor \ldots \lor \left( \tau_{1,2} \land \tau_{2,2} \land \ldots \land \tau_{m_2,r_2} \right) \lor \ldots \lor \left( \tau_{1,q} \land \tau_{2,q} \land \ldots \land \tau_{m_m,r_m} \right)
\]
Now we prove that if a satisfiable counter request appears in $S$, then $S'$ also contains such a counter request. First consider the case in which $R_i (i \geq cur)$ evaluates to true and the disjunct evaluates to true in $R_i$ derived from a disjunct other than $t_{cur}$. Since that disjunct should also present in $R'_{cur}$, $R_i$ evaluates to true.

Now consider the case in which $R_i$ evaluates to true and the disjunct evaluates to true in $R_i$ derived from $t_{cur}$. Assume the number of counter requests created by both entities between requests $R_{pre}$, $R_{cur}$ is $d(d \in \mathbb{E}ven \ d \geq 2)$. Therefore, $R_{cur} = R_{pre+d}$. It can be observed that, if a request which evaluates to true appears in sequence $S$, then that request should be either $R_{cur+d}$ or a later request. Otherwise, negotiation should have not lasted until $R_{cur}$.

Therefore, we prove if request $R_{cur+d+k} (k \in \mathbb{N})$ evaluates to true in sequence $S$, then a request in $S'$ also evaluates to true, by using mathematical induction on $k$.

Consider the case $k = 0$, where $R_{cur+d}$ has a disjunct which evaluates to true and that disjunct derived from $t_{cur}$. Since $t_{cur+1} \geq t_{pre}$, for every $t_{cur}$ derived from $t_{cur+1}$ there exists a disjunct $t_{pre}$ derived from $t_{pre+1}$ such that $t_{cur+1} = t_{pre}$, where $t_{cur} \in R_{pre+1}, t_{pre} \in R_{pre+i} (1 \leq i \leq d)$. Because $R_{cur+d}$ evaluates to true, by the definition of stricter disjunct, $R_{pre+d}$ must also evaluate to true. That disjunct can not be $t_{cur}$, since $R_{cur}$ in $S$ would become a satisfiable request. Therefore, a disjunct other than $t_{cur}$ becomes satisfiable, which means request $R'_{cur}$ in $S'$ also becomes satisfiable.

Assume the result holds for $k = 0, \ldots, p \left( \in \mathbb{N}^+ \right)$. Consider the case $k = p+1$. When comparing $R_{pre+1}$ with $R_{cur+1}$, for all disjuncts $t_{cur} \in R_{cur+1}$ derived from $t_{cur+1}$, there exists a disjunct $t_{pre} \in R_{pre+1}$ derived from $t_{pre+1}$, such that $t_{cur} \geq t_{pre}$. Since the number of counter requests in $S$ from $R_{cur+1}$ until a satisfiable request appears is $d + p$, using induction hypothesis we can remove all disjuncts derived from $t_{cur+1}$ without affecting completeness. Let us denote this new counter request, which also leads to a successful negotiation, by $R''$. Since $R''$ does not have any disjuncts derived from $t_{cur+1}$, it is equal to $R'_{cur+1}$, meaning that a counter request in $S'$ also becomes satisfiable.

Example 2. Consider disjunct $S_2 \land S_3$ in step 5 and $S_2$ in step 3 of the exemplary negotiation in Section 3-2. Since $S_2 \land S_3$ is stricter than $S_2$, according to Theorem 2 it is safe to remove $S_2 \land S_3$ from the counter request in step 5.

§1 Negotiation Failure Detection

Applying Theorem 2 may sometimes result in a counter request which has no disjuncts, which means that a successful negotiation is not possible. Therefore, in addition
to reducing the memory consumption, Theorem 2 provides a method to detect negotiation failures.

4. Memory Complexity of Parsimonious Strategy

This chapter presents the analysis of the memory complexity of implementation presented in Section 3.3. It is worth noting that AND/OR graph could be used to represent policies of a given entity. For example, Figure 2 shows how server’s policies in Table 1 can be represented by an AND/OR graph. It uses two kinds of nodes: AND nodes (depicted by rectangles) to represent disjuncts, OR nodes (depicted by circles) to represent credentials. AND/OR graph in Figure 2 can be stored in memory using adjacency-list representation as shown in Figure 3.

To compute the memory complexity of the adjacency-list representation, assume that \( m_c, m_s \) correspond to number of distinct disjuncts in client’s policies, server’s policies respectively. Since the client can have maximum of \( n_c + n_s + 1 + m_c \) nodes, the size of adjacency array is \( O(n_c + n_s + m_c) \). Furthermore, it can have at most \( m_c(n_c + n_s) \) edges. Therefore, total number of entries in adjacency lists is of order \( O(m_c(n_c + n_s)) \). It results in the memory complexity of adjacency-list representation of policies at the client side to be \( O(m_c(n_c + n_s)) \). Similarly, it can be shown that the memory complexity at the server side is \( O(m_s(n_c + n_s)) \).

The following lemma, which states the maximum number of independent disjuncts that can be created from a fixed number of credentials, is used when analyzing the memory complexity of Parsimonious strategy.

**Lemma 1.** Maximum number of independent disjuncts that can be created from \( n \) literals is \( \binom{n}{\lfloor \frac{n}{2} \rfloor} \).

Due to space limitations, only the outline of the proof of Lemma 1 is given. First we depict the partial order relationship defined by strictness among disjuncts using a Hasse diagram. In the Hasse diagram there are \( 2^n \) disjuncts (including the disjunct that has no literals, which means logical true) and \( n! \) distinct paths. Here, by a path we mean a sequence \( t_1, \ldots, t_n \), where \( t_1, \ldots, t_n \) are disjuncts satisfying the following conditions: \( t_1 = \text{true} \) (i.e. bottom of the lattice), \( t_n = \text{top} \) of the lattice, and \( t_{i+1} \geq t_i \forall i = 1, \ldots, n-1 \). Also, we can notice that two disjuncts are dependent if and only if there exists a path which contains both terms. It is easy to see that we can choose \( \binom{n}{\lfloor \frac{n}{2} \rfloor} \) disjuncts which are independent to each other. Therefore, we show that it is impossible to select more disjuncts than that while keeping the independent property intact. We can observe that any disjunct with \( k \) literals belongs to \((n-k)!k!\) paths. Also, inequality \((n-k)!k! \geq (n-\lfloor \frac{n}{2} \rfloor)!\lfloor \frac{n}{2} \rfloor!\) holds for \( 0 \leq k \leq n \). If we choose \( \binom{n}{\lfloor \frac{n}{2} \rfloor} \) + 1 disjuncts, there will be at least \( \left( \binom{n}{\lfloor \frac{n}{2} \rfloor} + 1 \right) (n-\lfloor \frac{n}{2} \rfloor)! \cdot \lfloor \frac{n}{2} \rfloor! \) paths, which means at least two disjuncts belong to the same path, therefore those disjuncts are dependent.

Using Lemma 1, we can derive Theorem 3.

**Theorem 3.** Client side memory complexity of Parsimonious strategy is \( O((n_c + n_s)2^n) \). Server side memory complexity of Parsimonious strategy is \( O((n_c + n_s)2^n) \).

**Proof.** Consider the client side first. As described earlier, memory complexity of storing policies at the client side is \( O(m_c(n_c + n_s)) \). During the negotiation phase each entity stores the requested policies by itself. As shown in Section 3.2, Section 3.3, it is not necessary to store two disjuncts which are dependent. According to Lemma 1, number of such disjuncts is bounded by \( \binom{n_s}{\lfloor \frac{n_s}{2} \rfloor} \), and length of each disjunct is at most \( n_s \). Therefore, memory consumption of client during the negotiation phase is at most \( n_s(n_c + n_s) \). During the credential exchange phase, credentials of the server get disclosed and client has to store them. Memory required to store this information is bounded by \( n_s \). Therefore, the total memory consumption is \( (m_c(n_c + n_s)) + n_s(n_c + n_s) \). Knowing that \( m_c \leq
\[
\left(\frac{n_s}{2} \right) \text{ and } \left(\frac{n_c}{2} \right) < 2^n, \text{ memory complexity of the client becomes } O\left((n_c + n_s)2^n\right). \]
Similarly, the memory complexity of the server is \(O\left((n_c + n_s)2^n\right)\).

5. Evaluation of the Proposed Method by Simulation

In this chapter, the results obtained by simulations using WS-ATN [Yamaki 08] – an ATN framework for web services platform – is presented. First in Section 5.1, the method used to generate random policy sets is described. Then in Section 5.2 ~ Section 5.5, experimental results are presented.

5.1 Random Input Generation for Simulation

Inputs for simulation were generated using Procedure 1 and Procedure 2. Procedure 1 generates policies for both parties, with the help of Procedure 2 which generates a single policy. Procedure 1 receives the total number of credentials \(n\) as a parameter and values of \(n_s, n_c\) are determined using a pseudo random generator such that the conditions \(1 \leq n_s \leq n - 1\) and \(1 \leq n_c \leq n - 1\) are satisfied. These conditions guarantee that all generated policy sets have at least one credential for each entity. Then, it generates policy for service \(S\) (line 9), and the condition in line 8 ensures that the policy of \(S\) is not empty, i.e. service is not anonymous. Line 13 and line 16 call Procedure 2 in order to generate remaining policies. Condition in line 11 ensures that there exists at least one unprotected credential, which is a necessary condition to have a successful negotiation.

Procedure 2 receives a set of credentials of the opponent entity as the parameter. Firstly, it decides which of the opponent’s credentials are to be included in the policy. Probability of being included is 0.5, independently of other credentials. Secondly, it creates disjuncts using the credentials selected in the previous step. Whether a particular credential is included in a disjunct is decided with probability 0.5, independently of other credentials.

§ 1 Generating Policies with Different Complexities

Although it is not explicitly stated in Procedure 2, after a new disjunct is added (line 16), the procedure ensures that existing disjuncts are mutually independent. There are two ways to maintain this property.

(1) Remove any stricter disjuncts after adding a new disjunct

(2) Remove any less stricter disjuncts after adding a new disjunct

The use of above mentioned methods only affects the distribution of the number of credentials in a disjunct. Disjuncts generated by method (1) has less number of credentials than disjuncts generated by method (2). Therefore, method (1) generates policy sets which are easy to satisfy, while method (2) generates those which are difficult to satisfy.

\begin{algorithm}
\caption{GenerateInputGraphs(n)}
\begin{algorithmic}
\State \(n_s \leftarrow \text{random}() \%(n-1) + 1\)
\State \(n_c \leftarrow n - n_s\)
\State \(S \leftarrow \emptyset\)
\ForAll {i = 1 to \(n_s\)}
\State \(S_i \leftarrow \emptyset 1 \leq i \leq n_s\)
\State \(C_j \leftarrow \emptyset 1 \leq j \leq n_c\)
\State \(\text{ClientCreds} \leftarrow \{C_1\} \cup \ldots \cup \{C_{n_c}\}\)
\State \(\text{ServerCreds} \leftarrow \{S_1\} \cup \ldots \cup \{S_{n_s}\}\)
\While {\(S = \emptyset\)}
\Do {\(S_i = \emptyset\) and \(C_j = \emptyset\)}
\For {i = 1 to \(n_s\)}
\State \(S_i \leftarrow \text{CreatePolicy}(	ext{ClientCreds})\)
\EndFor
\For {j = 1 to \(n_c\)}
\State \(C_j \leftarrow \text{CreatePolicy}(	ext{ServerCreds})\)
\EndFor
\EndWhile
\EndFor
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\caption{CreatePolicy(L)}
\begin{algorithmic}
\State \(Q \leftarrow \emptyset\)
\State \(S \leftarrow \emptyset\)
\ForAll {c \in L}
\If {\(\text{random}() \% 2 = 0\)}
\State \(S \leftarrow S \cup \{c\}\)
\EndIf
\EndFor
\State \(\text{size} \leftarrow |S|\)
\For {i = 1 to \(\text{size}\)}
\State \(T \leftarrow \emptyset\)
\ForAll {c \in S}
\If {\(\text{random}() \% 2 = 0\)}
\State \(T \leftarrow T \cup \{c\}\)
\EndIf
\EndFor
\State \(Q \leftarrow Q \cup \{T\}\)
\EndFor
\State \(\text{return } Q\)
\end{algorithmic}
\end{algorithm}
5.2 Memory Consumption Improvements

In this section, the effectiveness of the method proposed in Section 3.3 to reduce the memory consumption of Parsimonious strategy is presented. Simulations were done for varying \( n \) — the total number of credential of both parties — and the number of total nodes stored in memory was observed. For each value of \( n \), 1000 inputs were used. Simulations were done using implementations which uses the history of negotiation and which does not use the history of negotiation. Variation of the average of total number of nodes stored in memory with the \( n \) for both implementations are shown in Figure 4. As it is clear from Figure 4, the method presented in Section 3.3 is effective in reducing the memory consumption of Parsimonious strategy.

5.3 Detecting Negotiation Failures

To figure out how effectively the history of negotiation can be used to detect negotiation failures, the results of failed negotiations are summarized in Figure 5. In Figure 5, \( x \)-axis is \( 2\min(n_c, n_s) + 2 \) and \( y \)-axis denotes the number of exchanged messages. Values shown are averages of 139 samples for each value of \( x \)-axis. Theoretical message limit after which the client can determine a negotiation failure is \( 2\min(n_c, n_s) + 2 \). From Figure 5, it is clear that by using the condition proposed in Section 3.3, it is possible to detect negotiation failures at early stages of the negotiation.

5.4 Computational Overhead due to usage of the History of the Negotiation

The effects of using the history of negotiation are two folds. Since disjuncts can be removed from a counter request according to Theorem 2, it reduces the number of disjuncts that could appear in future counter requests; hence reduces the number of disjunct comparisons that have to be executed when utilizing Theorem 1. On the other hand, applying Theorem 2 itself increases the number of disjunct comparisons. Therefore, it is difficult to determine whether using the history of negotiation increases the computational cost or not. To evaluate the computational overhead, the number of disjunct comparisons occurred during negotiations was observed and the results are shown in Figure 6. As it is clear from Figure 6, the method proposed in Section 3.3 achieves its goals without increasing the overall computational cost.

5.5 Effects of the Complexity of Policies

To observe the improvements in memory consumption under different policy sets with varying complexities, the memory consumptions of policy sets with different complexities were observed. The complexities of policy sets were changed using the method described in Section 5.1.1 and the results are shown in Figure 7. The results reveal that for small values of \( n \), implementation presented in
Section 3.3 has a slightly higher memory consumption when it is used with difficult policies. Nevertheless, for large values of \( n \), it has low memory consumption when it is used with complex policies. This suggests that implementation proposed in Section 3.3 has a lower memory consumption for policies that are more complex, for large values of \( n \).

6. Conclusion

This paper focused on implementation and analysis of Parsimonious strategy. Particularly, implementation details of Parsimonious strategy and methods to reduce its memory consumption without compromising the completeness were presented. The worst case memory complexity of the proposed implementation was also analyzed. Furthermore, how the proposed method can be used to detect negotiation failures was described.

Simulations were used to evaluate the effectiveness of the proposed method. Experimental results revealed that using the history of the negotiation reduces the memory consumption, detects negotiation failures in early stages of the negotiation, and achieves these goals without increasing the overall computational cost. Finally, the variation of memory consumption under different policy sets with varying complexities was observed. Simulation results showed that the proposed implementation has a lower memory consumption for policies that are more complex, for large values of \( n \).

References


Received December 20, 2009.