

Automatically Checking Group Diffie-Hellman  
and XOR in the Strand Space Model  
BSc Thesis

Erik Jongsma  
LIACS, Leiden ([ejongsma@liacs.nl](mailto:ejongsma@liacs.nl))  
Supervisor: Dr. Tom Chothia  
CWI, Amsterdam ([chothia@cwi.nl](mailto:chothia@cwi.nl))

September 18, 2008

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Related work</b>	<b>2</b>
2.1	Strand Spaces . . . . .	3
2.2	Diffie-Hellman . . . . .	5
2.3	Group Diffie-Hellman . . . . .	6
2.4	Constraint Solving in Prolog . . . . .	6
2.5	Other programs . . . . .	7
<b>3</b>	<b>Adding Diffie-Hellman</b>	<b>8</b>
3.1	Regular Diffie-Hellman . . . . .	8
3.2	Group Diffie-Hellman . . . . .	8
3.2.1	Logic rules . . . . .	9
3.2.2	Implementation . . . . .	12
3.3	Adding XOR . . . . .	15
<b>4</b>	<b>Example protocols and attacks</b>	<b>16</b>
<b>5</b>	<b>Conclusion</b>	<b>19</b>
5.1	Further Work . . . . .	19

## 1 Introduction

Security protocols are all around us. We use them when we surf the internet, when we withdraw money from an ATM, etc. Most people do not realize the importance of the fact that all of these communications should be secure. However, a lot of research has been done in the areas of cryptography and security in general. In this thesis, we will focus on security protocols.

Designing secure protocols by hand is a difficult task. This is in part because it is sometimes very hard to spot a possible attack on your own protocol. Many protocols have been shown to have flaws which were overlooked by the original designers. Therefore, there has been an increased interest in automated analysis of protocols. All kinds of automated tools have been created, based on different formal models. Formal models are used to abstract a protocol away from its implementation. This is important, because if a protocol can be broken in a formal model, none of its implementations will ever be safe from attacks. Using formal models and automated tools, a lot of protocols previously believed to be secure have been broken.

One of the formal methods used to reason about protocols is the strand space model [FHG99]. This is a very nice model, in which abstracted protocols look very much like the original versions. There is a protocol analyzer that uses this model, written in Prolog [MS01, CE02]. The goal of this thesis is to describe how we have adapted this analyzer to incorporate Diffie-Hellman [DH76] and XOR operations. Since a lot of real life protocols use at least one of these operations, these adaptations enable the program to analyze a much larger suite of protocols.

We will start by discussing the related work in this area. Then, we will continue with an explanation of how we have added these operations to the analyzer. Where necessary, we will prove that the rules the program uses are correct. Finally, we will demonstrate that previously known attacks are found by our adaptation of the program.

## 2 Related work

In this section, we will present the work related to what we have done. First, we will define strand spaces and related notions. Then, we will talk about Diffie-Hellman key exchange and how the Diffie-Hellman operation can be added to strand spaces. Group Diffie-Hellman will also be explained. Finally, we will discuss the Prolog program we have worked with. We will also briefly describe other efforts to analyze Diffie-Hellman-based protocols using different formal models and tool support.

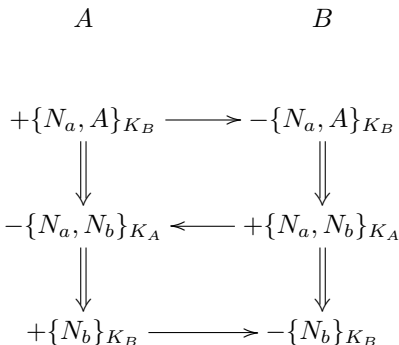
## 2.1 Strand Spaces

As already described in the introduction, the strand space method is a formal method used to analyze security protocols. This model is due to Fabrega, Herzog and Guttman [FHG99]. We will use the following example protocol:

1.  $A \rightarrow B : \{N_a, A\}_{K_B}$
2.  $B \rightarrow A : \{N_a, N_b\}_{K_A}$
3.  $A \rightarrow B : \{N_b\}_{K_B}$

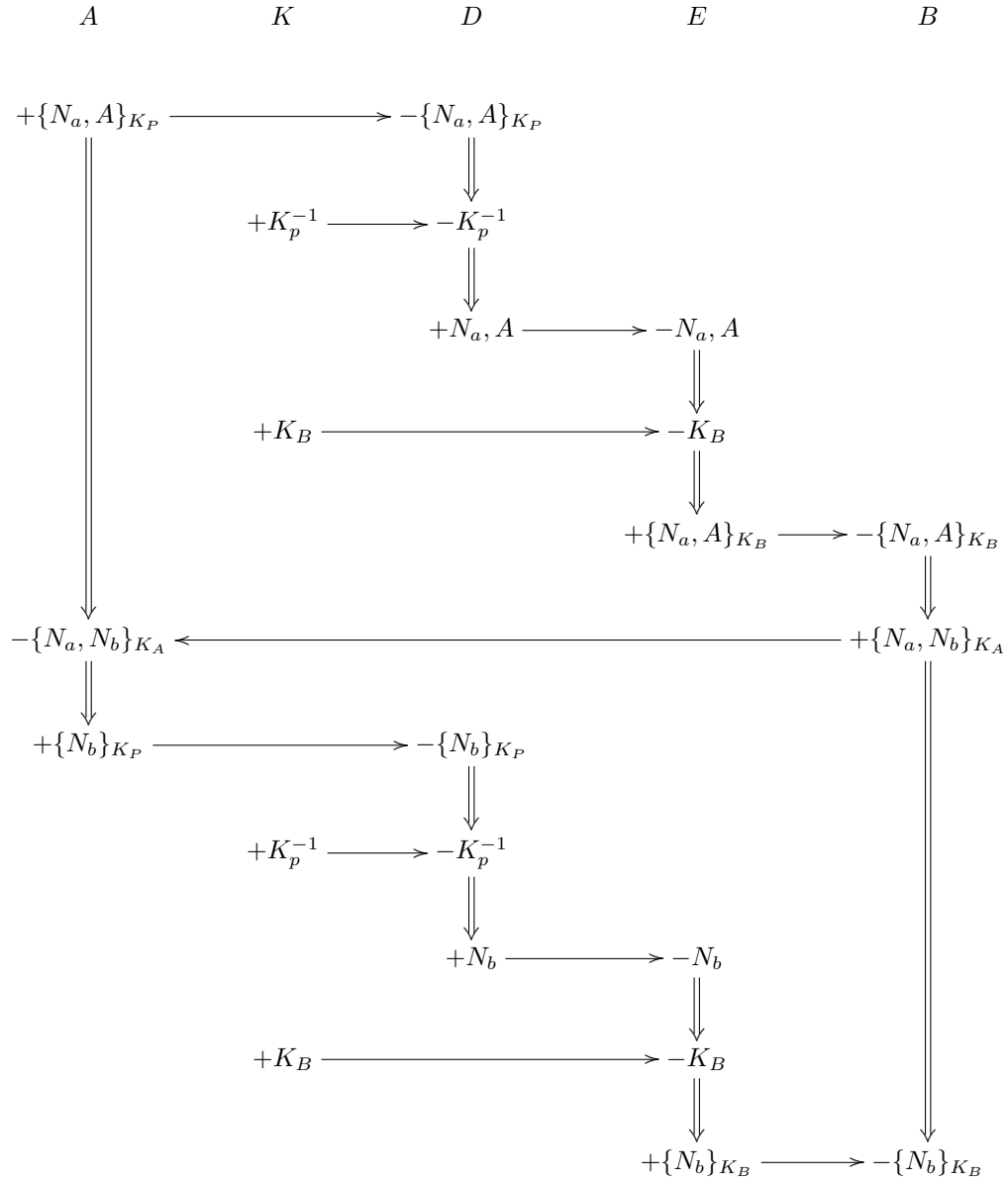
This protocol is due to Needham and Schroeder [NS78]. We will first explain the notation used in describing this protocol.  $A \rightarrow B$  means that  $A$  sends a message to  $B$ . After this, we list the message  $A$  sends. Parts of the message that are between braces are encrypted. The key used for this encryption is written as a subscript of the right brace. For example:  $\{A\}_{K_B}$  means that the message  $A$  is encrypted with key  $K_B$ . With  $K_B$ , we mean  $B$ 's public key, so that only  $B$  can decrypt the messages. Symmetric keys are written as  $K_{AB}$  (the key for  $A$  and  $B$ ). The letter  $N$  is reserved for nonces. These are random values, only known to the principal who created them, which are freshly created for each run of the protocol (so,  $N_a$  is only known to  $A$ ). Nonces are normally used to guarantee freshness of messages and to prevent replay attacks. Finally, the names of the principals ( $A$  and  $B$ ) can also be sent. Usually, this is implemented by sending the public key of the principal.

Now, using the Needham-Schroeder (NS) protocol, we are ready to discuss the strand space model. Instead of focusing on the messages that are being sent back and forth, we are now more interested in the principals of the protocol. Each legitimate principal in the protocol will be represented by one strand. A strand consists of a list of signed terms, where the sign is  $-$  if the term is received by the strand, and  $+$  if it is sent. A run of a protocol can now be described by a bundle, consisting of multiple connected strands. The bundle for a normal run of the NS protocol would look like this:



The arrows between the two strands indicate transmission of the message, while the arrows within the strands show the order in which messages are sent and/or received.

Attacks on protocols are modeled in strand spaces by including penetrator strands in the bundle. There are a number of different penetrator strands, which model the capabilities of our adversary. For example, there are penetrator strands for encryption, decryption, sending a plain text message, etc. The following is a bundle for the attack on the Needham-Schoeder protocol, due to Lowe [Low95]:



Here,  $K$  strands send out keys known by the penetrator,  $D$  strands decrypt, and  $E$  strands encrypt. All strands are penetrator strands, except of course the strands of  $A$  and  $B$ . Note that this is essentially a man-in-the-middle attack, allowing our attacker to impersonate  $A$  to  $B$ . He does this by simply re-encrypting the messages sent by  $A$  with  $B$ 's public key. At the end of this attack,  $A$  will think it has authenticated with our attacker, but  $B$  will believe he is talking to  $A$ , which is of course not what we want.

The attack by Lowe breaks an authentication property of the NS protocol. Now suppose we want to check if a protocol respects security, i.e. we want to know if our attacker can receive a certain secret. To model this in strand spaces, we simply add a strand to the bundle which receives this secret. Then, we see if the Prolog solver can complete the bundle by using the penetrator strands. Note that if the adversary can complete the bundle, this implies that he is able to send the secret (and thus, he must know it).

Strand spaces can also be used to prove correctness of protocols, which is at least as important as finding attacks. However, the Prolog analyzer we used does not (yet) have this ability. For more on proving correctness in the strand space model, see e.g. [FHG99].

## 2.2 Diffie-Hellman

When two principals want to communicate using an encrypted channel, they usually need a mutually known secret to begin with. However, in most cases, the principals do not have the opportunity to meet in person and agree on such a secret. A possible way to solve this problem is using Diffie-Hellman key agreement protocol [DH76]. The protocol works as follows:

Take a group  $G$  with generator  $g$ .  $A$  and  $B$  both choose a random element from the group, say  $a$  and  $b$ . Then,  $A$  sends  $g^a$  to  $B$ , and  $B$  sends  $g^b$  to  $A$ . Now,  $A$  computes  $(g^b)^a$ , and  $B$  computes  $(g^a)^b$ . Because  $g^{ab} = g^{ba} = k$ ,  $A$  and  $B$  now share the value  $k$ . However, we want  $k$  to be known only to  $A$  and  $B$ . The fact that this is true follows from the Computational Diffie-Hellman assumption, which states that knowing only  $g^a$  but not  $a$ , it is computationally infeasible to compute  $a$  if  $G$  is a suitably chosen group [BWJM97, BWM98]. So, our adversary has access to the values  $g^a$  and  $g^b$ , but since he knows neither  $a$  nor  $b$ , he cannot compute  $g^{ab}$ .

Note that the Diffie-Hellman only provides us with a secure channel between two principals, so there is no authentication of either of the participants. We will see the consequences of this later in the Prolog examples. Work has also been done on adding the Diffie-Hellman key agreement protocol to strand spaces. This makes proving protocol correctness considerably more difficult, because we basically need to allow our adversary to compute any function on messages [Her03]. However, in our case, we only need to add the Diffie-Hellman operations

to the operations our adversary can perform. We will discuss this point in further detail when we present our additions to the Prolog program.

### 2.3 Group Diffie-Hellman

The Diffie-Hellman key agreement protocol can be extended for use by more than two principals [STW96]. There are many ways of implementing Diffie-Hellman for groups, so we will present the most intuitive version. Suppose we have  $n$  principals. Each principal  $i$  chooses a random element  $r_i$  from the (suitably chosen) group  $G$ . In the first round, each principal  $i$  sends his  $g^{r_i}$  to principal  $i + 1$ , and so forth. In the second round, each principal  $i$  sends  $g^{r_i - 1 r_i}$  to principal  $i + 1$ . After  $n - 1$  rounds, each principal will then be able to compute  $g^{r_1 \cdots r_n} = k$ . The assumption that our adversary cannot compute  $k$  from what he has seen is the Computational Group Diffie-Hellman assumption [BCP01, BCP02b]. It has been shown that the Group Diffie-Hellman assumption can be reduced to the normal Computational Diffie-Hellman assumption [BCP03], so if our adversary is not part of the group (and thus knows none of the  $r_i$ ), he will not be able to compute  $k$ . We will now show an example run for GDH (Group Diffie-Hellman) with three principals:

1.  $A \rightarrow B : g^{r_a}, \quad B \rightarrow C : g^{r_b}, \quad C \rightarrow A : g^{r_c}$
2.  $A \rightarrow B : g^{r_a r_c}, \quad B \rightarrow C : g^{r_a r_b}, \quad C \rightarrow A : g^{r_b r_c}$
3.  $A, B$  and  $C$  can now compute  $g^{r_a r_b r_c} = k$ .

Note that the additions made to strand spaces to allow Diffie-Hellman key agreement also allow Group Diffie-Hellman to be done (because both are based on group exponentiation).

### 2.4 Constraint Solving in Prolog

Now, we will discuss the Prolog program we have been working with. The program was first created by Millen and Shmatikov [MS01], and later an improved version was introduced by Corin and Etalle [CE02]. Instead of normal strands, the program uses parametric strands, which are allowed to contain a variable [Son99, DLM<sup>+</sup>00]. Because of this, many different runs of the protocol can be represented by a single bundle. Since the ideas behind both programs are very similar, we will only discuss the original version here.

Instead of looking at connected strands in a bundle, we will now look at individual strands that are combined into a semibundle (a bundle which has not yet been completed). There are many different ways to interleave strands, and all of these are interleavings enumerated by the program (with some clever optimizations). Then, for each interleaving, a constraint set is generated. If the program can find a way to solve the constraint set given a certain set of penetrator reductions, it has found an attack. If it cannot solve the constraint set of any of the interleavings, it has not found an attack, and thus the protocol is secure (see [MS01] for the soundness and completeness proofs).

Again, we will have a look at our example protocol.:

1.  $A \rightarrow B : \{N_a, A\}_{K_B}$
2.  $B \rightarrow A : \{N_a, N_b\}_{K_A}$
3.  $A \rightarrow B : \{N_b\}_{K_B}$

Now, we want to check if our adversary can gain knowledge of an  $n_b$ . As interleaving, we take a normal execution of the protocol:

$$+\{n_a, a\}_{K_B}, -\{N_a, A\}_{K_b}, +\{N_a, n_b\}_{K_A}, -\{n_a, N_b\}_{K_a}, +\{N_b\}_{K_B}, -\{n_b\}_{K_b}, -\{n_b\}$$

Note that the capital letters are variables, for example,  $B$  can receive any nonce  $N_a$  in the second message, but since this is  $A$ 's nonce, it is not a variable in the first message. Also note we have added a node at the end which receives  $n_b$ . This means our adversary has to be able to send this nonce. This leads to the following constraint set to be solved by our adversary:

- (1)  $\{N_a, A\}_{K_b} : T_0 = \{a, b, e, pk(a), pk(b), \{n_a, a\}_{K_B}\}$
- (2)  $\{n_a, N_b\}_{K_a} : T_1 = T_0 \cup \{\{N_a, n_b\}_{K_a}\}$
- (3)  $n_b : T_2 = T_1 \cup \{\{N_b\}_{K_b}\}$

Here, we use the notation  $M : T$  to mean that the message  $M$  must be constructed by our adversary using his knowledge set  $T$ . The program can solve this particular set of constraints, showing the attack on this protocol shown in section 2.1. For more details, see [MS01].

## 2.5 Other programs

Of course, there are a lot more formal models than just the strand space model. Since it is usually hard to find attacks by hand, many protocol analyzers have been written. For example, Graham Steel et al. [SBM05] have written the CORAL program to find attacks on group protocols. There is no mention of Diffie-Hellman in their paper, but they do attack group protocols in a different way. Where the number of principals participating in a run is fixed in our program, their analyzer has no need for this. This could make it easier to find attacks on these kind of protocols.

Meadows et al. have also designed a protocol analyzer, this one called the NRL protocol analyzer. It uses equational rewriting to try and find attacks on protocols (or prove them secure). (Group) Diffie-Hellman operations have also been added to this tool [MN02, MNHE07]. An advantage of this method is that it will also find (some) computational attacks. Unfortunately, this analyzer has no public source code, so we have not been able to test it.

Also, theoretic work has been done on automatically analyzing Diffie-Hellman. Millen and Shmatikov for example describe a way to do symbolic analysis of protocols that use (Group) Diffie-Hellman [MS05]. For more information about group protocols and Diffie-Hellman, models and tool support, see [BCP<sup>+</sup>02a].

## 3 Adding Diffie-Hellman

### 3.1 Regular Diffie-Hellman

As mentioned above, the capabilities of our adversary are modeled by a set of penetrator reductions. So, to give our attacker the means to break Diffie-Hellman based protocols, we need to add the correct set of reduction rules. Please note that in the following, we will ignore computational properties of the algebraic constructs we are working with (due to implementation difficulties). For example, the fact that  $g^{|G|} = g$ , where  $|G|$  is the size of the underlying group, is ignored by our rule set. It is therefore possible that some real attacks are not found by our program.

We started by adding the non-group version of Diffie-Hellman. As we only deal with one or two exponents in this case, we quickly came up with the following three rules:

- (1) `reduct(dh(A), T, [[A, T]])`.
- (2) `reduct(dhk(A, B), T, [[A, T], [dh(B), T]])`.
- (3) `reduct(dhk(A, B), T, [[dh(A), T], [B, T]])`.

Rule one states the following: The attacker can create `dh(A)` (which is  $g^A$ ), if he can get `A` from the things he already knows (which is the set `T`). The second and third rules are similar, because they state: The attacker can create `dhk(A, B)` (which is  $g^{AB}$ ), if he can get either `A` and `dh(B)` from `T`, or `B` and `dh(A)` from `T`. This models the Diffie-Hellman assumption, because the attacker needs to know either `A` or `B` to compute  $g^{AB}$ , and can only compute  $g^A$  if he knows `A`.

We tried using the program with these rules on a simple non-authenticated Diffie-Hellman session, where a key is established and a secret is sent with this key. The program does indeed find the usual man-in-the-middle attack, in which the attacker sends each principal his own exponents instead of the ones received. Note that these rules are not enough to find certain attacks, because they do not allow for removal of exponents. Therefore, when looking at group Diffie-Hellman rules, we will take a different approach.

### 3.2 Group Diffie-Hellman

We will start by writing down the operations that can be performed by our attacker (for example, exponentiation). Then, we will write down the logic rules as they used in the program. We will show that these sets of rules are equivalent, and then explain how they are implemented in the Prolog program.

### 3.2.1 Logic rules

First, we take the ideal set of rules:

$$\frac{T \vdash_I q}{T \vdash_I gdh(\{q\})} GDH_I \quad \frac{T \vdash_I gdh(P) \quad T \vdash_I q}{T \vdash_I gdh(P \cup \{q\})} Exp_I$$

$$\frac{T \vdash_I gdh(P \cup \{q\}) \quad T \vdash_I q}{T \vdash_I gdh(P)} Inv_I \quad \frac{}{T \vdash_I P} P \in T \quad Axm_I$$

$GDH_I$ : If we can derive an exponent  $q$  from our knowledge set, then we can compute  $g^q$ .

$Exp_I$ : If we can derive both  $g^P$  and an exponent  $q$  from our knowledge set, we can compute  $g^{P \cup \{q\}}$ .

$Inv_I$ : If we can derive both  $g^{P \cup \{q\}}$  and an exponent  $q$  from our knowledge set, we can compute  $g^P$ .

$Axm_I$ : We can derive  $P$  from the knowledge set  $T$  if it is an element of  $T$ .

One of the problems we run into if we directly implement these rules is the fact that the Prolog program will not terminate. It can (and will) endlessly apply  $Exp_I$  and  $Inv_I$ . So, the set of logic rules we have implemented in the Prolog program looks like this:

$$\frac{T \vdash_P q}{T \vdash_P gdh(\{q\})} GDH_P \quad \frac{T \vdash_P gdh(P) \quad T \vdash_P q}{T \vdash_P gdh(P \cup \{q\})} Exp_P$$

$$\frac{T, gdh(P) \vdash_P m \quad T \vdash_P q}{T, gdh(P \cup \{q\}) \vdash_P m} Inv_P \quad \frac{}{T \vdash_P P} P \in T \quad Axm_P$$

Clearly, most practical rules are identical to the ideal rules, except the  $Inv$  rule. This is done because of the implementation problems described above. Now, we need to show that these two rulesets are equivalent, i.e.  $T \vdash_P m$  iff  $T \vdash_I m$ . First, we will need the following lemma:

**Lemma 3.2.1.** *If  $T \vdash_P gdh(P \cup \{q\})$ ,  $P \neq \emptyset$ , and  $T \vdash_P q$  then  $T \vdash_P gdh(P)$ .*

*Proof.* We will prove this by induction over the structure of the proof of  $T \vdash_P gdh(P \cup \{q\})$ .

**Base case:** If the proof of  $T \vdash_P gdh(P \cup \{q\})$  is done in one step, it must have been using the  $Axm_P$  rule. In this case  $T = T', gdh(P \cup \{q\})$ . We use the following proof:

$$\frac{\frac{}{T, gdh(P) \vdash_P gdh(P)} Axm_P \quad T \vdash_P q}{T, gdh(P \cup \{q\}) \vdash_P gdh(P)} Inv_P$$

Note that  $T, gdh(P \cup \{q\}) = T', gdh(P \cup \{q\}), gdh(P \cup \{q\}) = T$ , so we have  $T \vdash_P gdh(P)$ .

**Step case:** The last step of the proof is either  $Inv_P$  or  $Exp_P$ . If we used  $Exp_P$ , the end of the proof is as follows:

$$\frac{\begin{array}{c} \vdots \\ T \vdash_P gdh(P') \end{array} \quad \begin{array}{c} \vdots \\ T \vdash_P q' \end{array}}{T \vdash_P gdh(P' \cup \{q'\})} Exp_P$$

with  $P' \cup \{q'\} = P \cup \{q\}$ . If we have  $P = P'$ , we immediately get  $T \vdash_P gdh(P)$ . If  $P \neq P'$ , we have two cases. If  $|P| = 1$ , then  $P = q'$ , and we use  $GDH_P$  to get  $T \vdash_P gdh(\{q'\})$  from  $T \vdash_P q'$ . If  $|P| > 1$ , we write  $P'' \cup \{q\} = P'$ . Now, we have  $T \vdash_P gdh(P'' \cup \{q\})$ ,  $P'' \neq \emptyset$  and  $T \vdash_P q$ . Using the induction hypothesis, we get  $T \vdash_P gdh(P'')$ . Now, we can use this and  $T \vdash_P q'$  with  $Exp_P$  to get  $T \vdash_P gdh(P'' \cup \{q'\}) = gdh(P)$ :

$$\frac{T \vdash_P gdh(P'') \quad T \vdash_P q'}{T \vdash_P gdh(P'' \cup \{q'\}) = gdh(P)} Exp_P$$

If we used  $Inv_P$ , the end of the proof is as follows:

$$\frac{\begin{array}{c} \vdots \\ T', gdh(P') \vdash_P gdh(P \cup \{q\}) \end{array} \quad \begin{array}{c} \vdots \\ T' \vdash_P q' \end{array}}{T', gdh(P' \cup \{q'\}) \vdash_P gdh(P \cup \{q\})} Inv_P$$

Here,  $T = T', gdh(P' \cup \{q'\})$ . Using the induction hypothesis on  $T', gdh(P') \vdash_P gdh(P \cup \{q\})$  gives us  $T', gdh(P') \vdash_P gdh(P)$ . Then, we use  $T' \vdash_P q'$  and  $Inv_P$  to show  $T', gdh(P' \cup \{q'\}) \vdash_P gdh(P)$ :

$$\frac{T', gdh(P') \vdash_P gdh(P) \quad T' \vdash_P q'}{T = T', gdh(P' \cup \{q'\}) \vdash_P gdh(P)} Inv_P$$

□

We will also need the following logic rule, commonly called the Cut Elimination rule:

**Proposition 3.2.2** (Cut Elimination). *If we can prove  $m$  from  $T, s$ , and we can prove  $s$  from  $T$ , then we need only  $T$  to prove  $m$ :*

$$\frac{T, s \vdash_P m \quad T \vdash_P s}{T \vdash_P m} CutElim$$

*Proof.* If the proof of  $T, s \vdash_P m$  does not use  $s$ , then we can use the same proof to show that  $T \vdash_P m$ . If the proof does use  $s$ , we must construct a new proof tree. We already have a proof tree for  $T, s \vdash_P m$  and  $T \vdash_P s$ . Now, we will show how to construct a proof tree of  $T \vdash_P m$ . We will do this by removing all uses of  $s$  from the tree for  $T, s \vdash_P m$ .

If  $m$  is not of the form  $gdh(P)$  then  $s$  can only have been used in the  $Axm_P$  rule:

$$\frac{}{T, s \vdash_P s} Axm_P$$

We can then replace the uses of this rule with the proof tree of  $T \vdash_P s$ .

If  $s$  is of the form  $gdh(P)$ , we show there is a proof without using  $s$  by induction on the size of  $P$ :

**Base case:** If  $P$  is a singleton, then it can only have been used by the  $Axm_P$  rule and we can remove it from the proof using the method described above.

**Step case:** Assume that  $s$  is of the form  $gdh(P \cup \{q\})$ . If this term was used with the  $Axm_P$  rule, then we can remove it from the proof as above. If, on the other hand, it was used with the  $Inv_P$  rule then the end of the proof of  $T, gdh(P \cup \{q\}) \vdash_P m$  looks like:

$$\frac{\begin{array}{c} \vdots \\ T, gdh(P) \vdash_P m \end{array} \quad \begin{array}{c} \vdots \\ T \vdash_P q \end{array}}{T, gdh(P \cup \{q\}) \vdash_P m} Inv_P$$

So now, we need to prove  $T \vdash_P m$  from the above and from  $T \vdash_P gdh(P \cup \{q\})$ . The induction hypothesis we want to use here is

$$\frac{T, gdh(P) \vdash_P m \quad T \vdash_P gdh(P)}{T \vdash_P m} Ind$$

We already have the first statement, so we need to prove  $T \vdash_P gdh(P)$ . But, since we already have  $T \vdash_P gdh(P \cup \{q\})$  and  $T \vdash_P q$ , we can use Lemma 3.2.1 to do this. So, we have proven  $T \vdash_P m$ .  $\square$

Now, we are ready to state our main theorem and its proof:

**Theorem 3.2.3.** *The ideal and practical rulesets listed above are equivalent, i.e.  $T \vdash_P m \iff T \vdash_I m$ .*

*Proof.* We will prove this by induction on the structure of the proofs of  $T \vdash_I m$  and  $T \vdash_P m$  respectively.

**Base case:** Since  $Axm_P$  is the same as  $Axm_I$ , the base case is covered for both directions.

**Step cases:** for  $T \vdash_I m$  implies  $T \vdash_P m$ :

- $GDI$ : if the last step of the proof used this rule, then it must be the case that  $m = gdh(\{q\})$  and  $T \vdash_I q$ . By the induction hypothesis, we have that  $T \vdash_P q$ . Therefore  $T \vdash_P gdh(\{q\})$ , by applying the  $GDP$  rule.
- $Exp_I$ : if the last step of the proof used this rule, then it must be the case that  $m = gdh(P \cup \{q\})$ ,  $T \vdash_I gdh(P)$  and  $T \vdash_I q$ . By the induction hypothesis, we have that  $T \vdash_P gdh(P)$  and  $T \vdash_P q$ . Therefore  $T \vdash_P gdh(P \cup \{q\})$ , by applying the  $Exp_P$  rule.

- $Inv_I$ : if the last step of the proof used this rule, then it must be the case that  $m = gdh(P)$ ,  $T \vdash_I gdh(P \cup \{q\})$  and  $T \vdash_I q$ . By the induction hypothesis, we have that  $T \vdash_P gdh(P \cup \{q\})$  and  $T \vdash_P q$ . Now, we prove  $T \vdash_P gdh(P)$  (where the rule  $Ind$  stands for use of the induction hypothesis):

$$\frac{\frac{\overline{T, gdh(P) \vdash_P gdh(P)} \text{ Axm} \quad \overline{T \vdash_P q} \text{ Ind}}{\overline{T, gdh(P \cup \{q\}) \vdash_P gdh(P)} \text{ Inv}_P \quad \overline{T \vdash_P gdh(P \cup \{q\})} \text{ Ind}}{\overline{T \vdash_P gdh(P)} \text{ CutElim}}$$

**Step cases:** for  $T \vdash_P m$  implies  $T \vdash_I m$ :

- $GDH_P$ : if the last step of the proof used this rule, then it must be the case that  $m = gdh(\{q\})$  and  $T \vdash_P q$ . By the induction hypothesis, we have that  $T \vdash_I q$ . Therefore  $T \vdash_I gdh(\{q\})$ , by applying the  $GDH_I$  rule.
- $Exp_P$ : if the last step of the proof used this rule, then it must be the case that  $m = gdh(P \cup \{q\})$ ,  $T \vdash_P gdh(P)$  and  $T \vdash_P q$ . By the induction hypothesis, we have that  $T \vdash_I gdh(P)$  and  $T \vdash_I q$ . Therefore  $T \vdash_I gdh(P \cup \{q\})$ , by applying the  $Exp_I$  rule.
- $Inv_P$ : if the last step of the proof used this rule, then it must be the case that  $T, gdh(P) \vdash_P m$  and  $T \vdash_P q$ . By the induction hypothesis, we have that  $T, gdh(P) \vdash_I m$  and  $T \vdash_I q$ . Now, we prove  $T, gdh(P \cup \{q\}) \vdash_I m$  (writing  $T'$  for  $T, gdh(P \cup \{q\})$ ):

$$\frac{\frac{\overline{T, gdh(P) \vdash_I m} \text{ Ind} \quad \overline{T' \vdash_I gdh(P \cup \{q\})} \text{ Axm} \quad \frac{\overline{T \vdash_I q} \text{ Ind}}{\overline{T' \vdash_I q} \text{ Add}} \text{ Inv}_I}{\overline{T', gdh(P) \vdash_I m} \text{ Add} \quad \overline{T' \vdash_I gdh(P)} \text{ CutElim}}{\overline{T' \vdash_I m}}$$

where  $Add$  means we add extra information to the left-hand side, which never hurts.

□

### 3.2.2 Implementation

We will start out by proving that our program will terminate:

**Theorem 3.2.4.** *The practical ruleset can produce only finite proof trees.*

*Proof.* Suppose the attacker is asked to produce  $gdh(X)$  from his knowledge set  $T$ , where  $X$  is a set of exponents. We will look at the rules he can use to do this. The program can, for example, apply the  $Inv_P$  rule to the knowledge set. If  $T = \{gdh(a, b), b\}$ , applying  $Inv_P$  will result in  $T = \{gdh(a), b\}$ . As the knowledge set is finite, the  $Inv_P$  rule can only remove a finite amount

of exponents. The program can also use the  $Exp_P$  rule. As the number of exponents we are looking for is finite, and  $Exp_P$  adds exactly one exponent at a time, this rule can only be used a finite number of times. Furthermore,  $GDH_P$  is used at most once. Since the other rules are used only a finite amount of times, there will also be a finite amount of uses of the  $Axm_P$  rule. So, there is only a finite amount of possible ways to try and construct  $gdh(X)$ .  $\square$

So, if the program uses the rules in the way described above, it will terminate.

Now, we will have a look at the logic rules as they are implemented in the Prolog program. First, the new reduction rules:

- ```
(1): reduct(gdh(A),T,[[A,T]]) :-
      var(A), !.

(2): reduct(gdh([A]),T,[[A,T]]).

(3): reduct(gdh(X),T,[[gdh(Y),T],[A,T]]) :-
      select(A,X,Y),
      permutation(Y,Z).

(4): reduct(gdh(X),T,[]) :-
      member(Y,Z),
      member(gdh([Y]),T),
      subset(X,[Y]).
```

Rule 1 states that if our adversary needs to find  $g^A$ , where  $A$  is a variable, he can skip over it (just like a normal variable is skipped over in the program). Rule 2 states that we can construct  $g^A$  if  $A$  is a single constant, and if we can reduce that constant from  $T$  (this rule corresponds to  $GDH_P$ ). Rule 3 states that we can add an exponent  $A$  we already know to  $g^Y$ , which we also know (this rule corresponds to  $Exp_P$ ). Rule 4 is not listed in our logical rules as described above, but is an attempt to make our adversary more powerful. This rule states that if we know  $g^Y$ , where  $Y$  is a single variable, we can construct  $g^{YY}$ ,  $g^{YYY}$  etc, by exponentiating with 2,3,etc. respectively. Note that the  $Axm_P$  rule is not listed here, because it is already implemented in the program by other rules.

Now that we have seen the construction part, let us have a look at the remove rules. In the original program, the knowledge set was simplified in a single pass over the contents of this set. Unfortunately, this will not suffice in our case. For example, consider a knowledge set that looks like this:  $T = \{a + gdh([b]), gdh([b, c]), c\}$ . If we go through the list once, we will not be able to decrypt  $a$ , because we do not know the key. However, once we have removed the exponent  $c$  from the second element of the list, we do have the key we need. So, we changed the original remove rule to one that repeatedly goes through  $T$ , repeating until nothing can be broken down anymore. This is done using the following rules:

```

remv(X,Z) :-
    breakrules(X,Z).

breakrules(X,Z) :-
    breakrule(X,X,Y),
    not(permutation(X,Y)), !,
    breakrules(Y,Z).

breakrules(X,X).

```

In the breakrule predicate, all old remv rules are listed, as well as the new ones used to break down GDH values. The new rules are as follows:

```

(1): breakrule(Z,[gdh(X)|T],W) :-
    member(B,X),
    member(inv(B),X),
    select(B,X,E),
    select(inv(B),E,Y),
    breakrule(Z,[gdh(Y)|T],W), !.

(2): breakrule(Z,[gdh(X)|T],W) :-
    member(A,X),
    var(A),
    removevars(X,Y),
    breakrule(Z,[gdh(Y)|T],W), !.

(3): breakrule(Z,[gdh(X)|T],W) :-
    member(A,X),
    select(gdh(X),Z,Z2),
    solve([[A,Z2]],[]),
    select(A,X,Y),
    breakrule(Z,[gdh(Y)|T],W), !.

```

The first rule is needed for certain protocols. It states that if an exponent and its inverse are in the list of exponents, they are both removed (note that this is in fact what is done computationally when an exponent is removed). The second rule removes all variables from the list of exponents, because those variables are constructed by our adversary (so he always knows those). The third rule removes the exponents our adversary is able to deduce from the rest of the knowledge set, via the solve predicate. Note that if we write `member(A,Z)` instead of `solve` and `select`, the program becomes a lot faster but is unable to find certain kinds of attacks. There is a set of these rules for messages like  $A + gdh(X)$  as well, allowing our adversary to decrypt messages when he can construct the key.

### 3.3 Adding XOR

After adding (G)DH to the Prolog program, we decided to add predicates for XOR as well. We will use these predicates to find attacks on group protocols which use the XOR operation in the wrong way (see [MS07]). First, we added support for the XOR operation on two elements:

(1): `reduct(A,T,[[B,T1]]) :-  
       member(mxor(A,B),T),  
       select(mxor(A,B),T,T1).`

(2): `reduct(A,T,[[B,T1]]) :-  
       member(mxor(B,A),T),  
       select(mxor(B,A),T,T1).`

(3): `reduct(mxor(A,B),T,[[A,T],[B,T]]).`

The first and second rule describe how we can get information from XOR-ed data. For example, we can get  $A$  from  $A \text{ XOR } B$  if we know (can construct)  $B$ . The third rule simply describes that our adversary can compute the XOR of two messages he knows.

Although not necessary for most protocols which use the XOR operation, we can also add support for XOR-ing of a list of elements. To do this, we could implement the following set of rules:

$$\begin{array}{c}
 \frac{T \vdash q}{T \vdash \text{XOR}(\{q\})} \text{ XOR} \qquad \frac{T \vdash \text{XOR}(P \cup \{q\}) \quad T \vdash P}{T \vdash q} \text{ Inv} \\
 \\
 \frac{T \vdash \text{XOR}(P \cup \{q\}) \quad T \vdash q}{T \vdash \text{XOR}(P)} \text{ Rem} \qquad \frac{T \vdash \text{XOR}(P) \quad T \vdash q \quad q \notin P}{T \vdash \text{XOR}(P \cup \{q\})} \text{ Add} \\
 \\
 \frac{T \vdash \text{XOR}(P) \quad T \vdash \text{XOR}(Q)}{T \vdash \text{XOR}(P \Delta Q)} \text{ Comb} \qquad \frac{}{T \vdash P} P \in T \text{ Axi}
 \end{array}$$

Here,  $P \Delta Q$  is the notation for the symmetric difference of  $P$  and  $Q$ , i.e. all  $x$  which are in either  $P$  or  $Q$ , but not in  $P \cap Q$ . To implement these rules in the Prolog program, we would need to solve the loop problem with *Add* and *Rem* just like in the case of the ideal GDH rules. A similar proof as before would be needed to show equivalence between the two rulesets. Due to time constraints, this option was not explored further.

## 4 Example protocols and attacks

The Prolog program with the new rules added to it was of course tested on several protocols. We will list our results below.

The first test we did was on a normal Diffie-Hellman exchange. Of course, this exchange is not safe against active adversaries, so the program should find an attack. Indeed, it does find the usual man-in-the-middle attack, which intercepts all DH values the principals are trying to send. The output of the program looks as follows:

```
Trace:
send(dh(na))
recv(dh(_G422))
send([s]+dhk(na, _G422))
recv(dh(_G443))
send(dh(nb))
recv([_G460]+dhk(_G443, nb))
recv(s)
```

```
Bundle:
[send(dh(na)), recv(dh(_G422)), send([s]+dhk(na, _G422))]
[recv(dh(_G443)), send(dh(nb)), recv([_G460]+dhk(_G443, nb))]
[recv(s)]
```

Note that the third strand in the bundle is a test strand, which receives the secret  $s$ . The program has to complete the bundle with this test strand in it in order to actually find an attack. Now, we will have a look at an attack on the normal GDH protocol. The output looks as follows:

```
Trace:
send(gdh([r1]))
recv(gdh([r1]))
send(gdh([r1, r1]))
recv(gdh([r1, r1]))
send(s+gdh([r1, r1, r1]))
recv(s)
```

```
Bundle:
[recv(s), send(stop)]
[send(gdh([r1]), recv(gdh([r1])), send(gdh([r1, r1])),
  recv(gdh([r1, r1])), send(s+gdh([r1, r1, r1]))]
[recv(gdh([_G489]), send(gdh([r2])), recv(gdh([_G489, _G512])),
  send(gdh([_G489, r2]))]
[recv(gdh([_G535]), send(gdh([r3])), recv(gdh([_G555, _G535])),
  send(gdh([_G535, r3])), recv(_G579+gdh([_G555, _G535, r3]))]
```

Note that the attack is found without using the other two principals, by just reflecting whatever principal 1 sends back at him. Of course, a man-in-the-middle attack would work here as well, but this attack is simpler.

An attempt was made by Ateniese et al. [AST98, AST00] to add authentication to the GDH protocol to try and secure it against active adversaries. Unfortunately, the resulting protocols are not really secure [PQ01], and later on, it was shown that building a secure group protocol using only GDH is in fact impossible [PQ06]. The protocol we will look at now is called A-GDH.2.

The protocol works as follows: In the first phase,  $i$  will receive  $p_{i-1}^{r_1^{-1}}, \dots, p_{i-1}^{r_{i-1}^{-1}}$ , where  $p_{i-1} = g^{r_1 r_2 r_3 \dots r_{i-1}}$ . He will then pick a random value  $r_i$ , and send out  $p_i, p_i^{r_1^{-1}}, \dots, p_i^{r_i^{-1}}$ , where  $p_i = g^{r_1 r_2 r_3 \dots r_i}$ . The last principal  $m$  in this chain will then broadcast  $p_m^{r_1^{-1} K_1}, \dots, p_m^{r_{m-1}^{-1} K_{m-1}}$ . Each principal  $i$  can then calculate the key  $p_m$  by taking the appropriate message and exponentiating it with  $r_i K_i^{-1}$ .

For example, with 4 principals, an exchange would be:

$$\begin{aligned} A &\rightarrow B : g^{r_1} \\ B &\rightarrow C : g^{r_1 r_2}, g^{r_1}, g^{r_2} \\ C &\rightarrow D : g^{r_1 r_2 r_3}, g^{r_1 r_2}, g^{r_2 r_3}, g^{r_1 r_3} \\ D &\rightarrow \text{all} : g^{r_2 r_3 r_4 K_1}, g^{r_1 r_3 r_4 K_2}, g^{r_1 r_2 r_4 K_3} \end{aligned}$$

This protocol can be attacked in the following way, due to Pereira and Quisquater [PQ01]:

1. We assume our adversary takes part in the first run of the protocol, as principal  $C$ . He acts like a normal participant of the protocol, except that he sends  $g^{r_1 r_2 r_3}, g^{r_1 r_2}, g^{r_2 r_3}, g^{r_1 r_2}$  instead of  $g^{r_1 r_2 r_3}, g^{r_1 r_2}, g^{r_2 r_3}, g^{r_1 r_3}$ . So,  $g^{r_1 r_2 r_4 K_2}$  will be broadcast, and our adversary can compute  $g^{r_1 r_2 r_4}$  from  $g^{r_1 r_2 r_4 K_3}$  because he knows  $K_3$ .

2. Now, our adversary leaves the group. The other three principals decide to do another run of the protocol. This time, the message the first principal sends is replaced by our adversary by  $g^{r_1 r_2 r_4}$ . The protocol will then proceed as follows:

$$\begin{aligned} A &\rightarrow B : g^{r_1 r_2 r_4} \\ B &\rightarrow C : g^{r_1 r_2 r_4 r'_2}, g^{r_1 r_2 r_4}, g^{r'_2} \end{aligned}$$

Then, in the final broadcast, our adversary sends  $g^{r_1 r_2 r_4 K_2}$  to principal  $B$  who will compute the key to be  $g^{r_1 r_2 r_4 r'_2}$ . However, this key was sent by principal  $B$  in message 2, so the adversary will be able to read the messages sent by  $B$  while not being part of the protocol run.

Our program should be able to find this attack by letting it analyze the two protocol runs, but unfortunately it is not fast enough at this point. So, instead, we let it find the attack we do in phase 2, by providing our adversary with the values recorded in run 1. The results then are as follows:

```

Trace:
send(gdh([r1]))
recv(gdh([r1]))
recv(gdh([r11, r21, r41]))
send([gdh([r2, r11, r21, r41]), gdh([r11, r21, r41]), gdh([r2])])
recv(gdh([r11, r21, r41, msk(b, d)]))
send(s+gdh([inv(msk(b, d)), r2, r11, r21, r41, msk(b, d)]))
recv([gdh([r2, r11, r21, r41]), gdh([r2, r11, r21, r41]),
      gdh([r2, r11, r21, r41])])
send(gdh([msk(b, d), r4, r2, r11, r21, r41]))
send(gdh([msk(a, d), r4, r2, r11, r21, r41]))
recv(s)

```

```

Bundle:
[send(gdh([r1]), recv(gdh([r1])))]
[recv(gdh([r11, r21, r41]),
  send([gdh([r2, r11, r21, r41]), gdh([r11, r21, r41]), gdh([r2])]),
  recv(gdh([r11, r21, r41, msk(b, d)])),
  send(s+gdh([inv(msk(b, d)), r2, r11, r21, r41, msk(b, d)])))]
[recv([gdh([r2, r11, r21, r41]), gdh([r2, r11, r21, r41]),
      gdh([r2, r11, r21, r41])]),
  send(gdh([msk(b, d), r4, r2, r11, r21, r41])),
  send(gdh([msk(a, d), r4, r2, r11, r21, r41]))]
[recv(s), send(stop)]

```

Indeed, this is exactly the attack we are supposed to find.

Now, we will present an attack found using the newly added rules for the XOR operation. The attack used here is on the Bull Otway recursive authentication protocol [BO97], found by Ryan and Schneider [RS98]. An example run of this protocol for 3 principals and a server would be:

```

A → B : {A, B, NA}Ka
B → C : {B, C, NB, {A, B, NA}Ka}KB
C → S : {C, S, NC, {B, C, NB, {A, B, NA}Ka}KC}KC
S → C : KAB ⊕ sha({NA}KA), {A, B, NA}KAB, KAB ⊕ sha({NB}KB), {B, A, NB}KAB,
KBC ⊕ sha({NB}KB), {B, C, NB}KBC, KBC ⊕ sha({NC}KC), {C, B, NC}KBC
C → B : KAB ⊕ sha({NA}KA), {A, B, NA}KAB, KAB ⊕ sha({NB}KB), {B, A, NB}KAB,
KBC ⊕ sha({NB}KB), {B, C, NB}KBC
B → A : KAB ⊕ sha({NA}KA), {A, B, NA}KAB

```

After this exchange, each principal can then compute the key for communication with his “right” and “left” neighbor. However, as we will see, if our attacker acts as  $C$  in this exchange, he cannot only get  $K_{BC}$ , but also  $K_{AB}$ . The output of the program for this attack is:

Trace:

```
send([a, b, na]+ka)
recv([a, b, na]+ka)
send([b, c, nb, [a, b, na]+ka]+kb)
recv([c, s, nc, [b, c, nb, [a, b, na]+ka]+kb]+kc)
send([mxor(kab, sha(na+ka)), mxor(kab, sha(nb+kb)),
      mxor(kbc, sha(nb+kb)), mxor(kbc, sha(nc+kc))])
recv(kab)
```

Bundle:

```
[recv(kab), send(stop)]
[send([a, b, na]+ka),
 recv([mxor(_G621, sha(na+ka)), [a, b, na]+_G621])]
[recv([a, b, na]+ka),
 send([b, c, nb, [a, b, na]+ka]+kb),
 recv([_G680, _G683, mxor(_G689, sha(nb+kb)),
      [b, a, nb]+_G689, mxor(_G715, sha(nb+kb)), [b, c, nb]+_G715]),
 send([_G680, _G683])]
[recv([c, s, nc, [b, c, nb, [a, b, na]+ka]+kb]+kc),
 send([mxor(kab, sha(na+ka)), mxor(kab, sha(nb+kb)),
      mxor(kbc, sha(nb+kb)), mxor(kbc, sha(nc+kc))])] ]
```

Note that the encrypted messages are left out in the final parts of the protocol, in order to speed up the program. We can do this because the encrypted parts are not needed to find the key.

## 5 Conclusion

To conclude, the additions we have made to the Prolog program seem to be successful. All known attacks we have tried to find with our improved program are indeed found. Unfortunately, the program executes very slowly. So, while it would have been nice to test other protocols with it, this has not been done yet. Once we manage to speed it up, it would be interesting to see if it can find known attacks on other protocols, and maybe even new ones.

### 5.1 Further Work

There is a lot of potential further work on and with this analyzer. For example:

- Add operations used in other (group) protocols.
- Add support for multiple runs of one protocol (after each other instead of at the same time).
- Improve performance.
- Analyze existing protocols.

- Add support for computational attacks on Diffie-Hellman-based protocols.
- Allow the program to prove security against passive (or even active) adversaries.

Finally, I'd like to thank my supervisor, Tom Chothia, very much for all the support and inspirational meetings.

## References

- [AST98] G. Ateniese, M. Steiner, and G. Tsudik. Authenticated group key agreement and friends. In *5th ACM Conference on Computer and Communications Security*, pages 17–26. ACM Press, 1998.
- [AST00] G. Ateniese, M. Steiner, and G. Tsudik. New multiparty authentication services and key agreement protocols. *IEEE Journal on Selected Areas in Communications*, 4:628–639, 2000.
- [BCP01] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Provably authenticated group diffie-hellman key exchange - the dynamic case. In *ASIACRYPT '01: Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security*, pages 290–309, London, UK, 2001. Springer-Verlag.
- [BCP<sup>+</sup>02a] E. Bresson, O. Chevassut, O. Pereira, D. Pointcheval, and J. Quisquater. Two formal views of authenticated group diffie-hellman key exchange. DIMACS Workshop on Cryptographic Protocols in Complex Environments, Piscataway, New Jersey, USA, May 2002.
- [BCP02b] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Group diffie-hellman key exchange secure against dictionary attacks. In *ASIACRYPT '02: Proceedings of the 8th International Conference on the Theory and Application of Cryptology and Information Security*, pages 497–514, London, UK, 2002. Springer-Verlag.
- [BCP03] E. Bresson, O. Chevassut, and D. Pointcheval. The group diffie-hellman problems. In *SAC '02: Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*, pages 325–338, London, UK, 2003. Springer-Verlag.
- [BO97] J. Bull and D. Otway. The authentication protocol. Technical report, DRA/CIS3/PROJ/CORBA/SC/1-/CSM/436-04/0.5b, 1997.
- [BWJM97] S. Blake-Wilson, D. Johnson, and A. Menezes. Key agreement protocols and their security analysis. In *Proceedings of the 6th IMA*

- International Conference on Cryptography and Coding*, pages 30–45, London, UK, 1997. Springer-Verlag.
- [BWM98] S. Blake-Wilson and A. Menezes. Authenticated diffie-hellman key agreement protocols. In *SAC '98: Proceedings of the Selected Areas in Cryptography*, pages 339–361, London, UK, 1998. Springer-Verlag.
- [CE02] R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In *9th Int. Static Analysis Symp. (SAS)*, volume LNCS 2477, pages 326–341. Springer-Verlag, 2002.
- [DH76] W. Diffie and M.E. Hellmann. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.
- [DLM<sup>+</sup>00] N. Durgin, P. Lincoln, J. Mitchell, A. Scedrov, and I. Cervesato. Relating strands and multiset rewriting for security protocol analysis. In *CSFW '00: Proceedings of the 13th IEEE workshop on Computer Security Foundations*, page 35, Washington, DC, USA, 2000. IEEE Computer Society.
- [FHG99] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.
- [Her03] J.C. Herzog. The diffie-hellman key-agreement scheme in the strand-space model. In *Proceedings, 16th IEEE Computer Security Foundations Workshop*. IEEE CS Press, June 2003.
- [Low95] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.
- [MN02] C. Meadows and P. Narendran. A unification algorithm for the group diffie-hellman protocol. In *In Proc. of WITS 2002*, pages 14–15, 2002.
- [MNHE07] C. Meadows, P. Narendran, J. Hendrix, and S. Escobar. Diffie-hellman cryptographic reasoning in the maude-nrl protocol analyzer. In *2nd International Workshop on Security and Rewriting Techniques (SecReT 2007)*, 2007.
- [MS01] J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *8th ACM Conference on Computer and Communication Security*, pages 166–175. ACM SIGSAC, November 2001.

- [MS05] J. Millen and V. Shmatikov. Symbolic protocol analysis with an abelian group operator or diffie-hellman exponentiation. *Journal of Computer Security*, 13(3):515–564, May 2005.
- [MS07] R. Monroy and G. Steel. Faulty group protocols. 2007.
- [NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
- [PQ01] O. Pereira and J. Quisquater. A security analysis of the cliques protocol suites. In *14th IEEE Computer Security Foundations Workshop*, pages 73–81. IEEE Computer Society Press, June 2001.
- [PQ06] O. Pereira and J. Quisquater. On the impossibility of building secure cliques-type authenticated group key agreement protocols. *Journal of Computer Security*, 14(2):197–246, 2006.
- [RS98] P. Ryan and S. Schneider. An attack on a recursive authentication protocol. a cautionary tale. *Inf. Process. Lett.*, 65(1):7–10, 1998.
- [SBM05] G. Steel, A. Bundy, and M. Maidl. Attacking group protocols by refuting incorrect inductive conjectures. *Journal of Automated Reasoning*, pages 1–28, 2005.
- [Son99] D. X. Song. Athena: a new efficient automatic checker for security protocol analysis. In *CSFW '99: Proceedings of the 12th IEEE workshop on Computer Security Foundations*, page 192, Washington, DC, USA, 1999. IEEE Computer Society.
- [STW96] M. Steiner, G. Tsudik, and M. Waidner. Diffie-hellman key distribution extended to group communication. In *CCS '96: Proceedings of the 3rd ACM conference on Computer and communications security*, pages 31–37, New York, NY, USA, 1996. ACM.