Abstract

Security protocol is a special kind of protocols, which can be a real protocol or an abstract one. Currently, there are two approaches of analysing security protocol. The first one is the symbolic approach. Under this approach, all the cryptographic primitives in the protocol are considered as perfect black boxes and the adversary's actions are restricted. The other one is the computational approach. It is based on complexity theory. The security under this approach relies on the underlying algorithm assumptions and the adversary here is a probabilistic polynomial-time Turing machine. Currently, most of the automatic security provers work under the symbolic approach. Thus, their proofs are not computationally sound. CryptoVerif, which is designed by Bruno Blanchet, is a computationally sound automated security prover. It works directly in the computational model. In this thesis, we will use CryptoVerif to verify the security of the 1-out-of-2 oblivious transfer protocol. During the process of verification, the tool itself will be also evaluated.

Acknowledgement

I would like to thank Bogdan Warinschi, my supervisor, for supervising this project and guiding me in my study.

I would also like to thank Bruno Blanchet, the designer of CryptoVerif, for answering my questions about this tool.

Contents

1 Introduction

Security protocols are abstract or real protocols that consist of a sequence of processes for some security purposes. They are mainly used in protection of privacy, key management, entity authentication, integrity check, non-repudiation and data confidentiality. Security protocols are essential ingredients of modern cryptography. Historically, they first appeared in 1978 when Needham and Schroeder proposed a set of authentication protocols [13]. The Needham-Schroeder protocols only focus on the cryptographic operations at the abstraction level but not the details of cryptographic algorithms and their implementations. Anyway, it was a breakthrough of cryptography's development. Meanwhile, the need of techniques for verifying the correctness of the security protocols also appeared.

To be accepted, the security guarantees of security protocols should be given. Thus, the proofs of security protocols to check if they satisfy some required security properties are necessary. In the 1980s, cryptographers started to research techniques of analysing security protocols. Later, in the year 1989, Burrows, Abadi and Needham designed a logical language - the BAN logic [5] for modelling and analysing security protocols. It has been a success. The BAN logic formulates protocols as rules in a more systematic way. But actually, it was highly criticised because of two reasons: first, taking all the interactions among all parties into account is a difficult task; secondly, the BAN logic assumes that all the cryptographic primitives involved in the protocols are perfect. Anyway, it was a pioneering technique of proving security of protocols.

In order to improve the security of protocols, since the BAN logic was purposed, a number of security protocols analysing methods and tools have been developed. As is known to all, the proof of security protocol is difficult and susceptible to error. By-hand proofs of the protocols are really difficult tasks. The analysis of the computational details that are based on mathematical operations is hard and maybe undoable. Even in the abstraction level, the manual proofs of complex protocols are really time-consuming. In the last twenty years, with the development of cryptography, the automated verification tools of security protocols have become available. Currently, there are two main models of analysing security protocols: the Dolev-Yao model and the computational model. The former focuses on the abstraction level of the protocols and treats the involved cryptographic primitives as ideal black boxes which are unbreakable. The proofs in Dolev-Yao model are deterministic but not probabilistic. In contrast, the latter focuses on the bit-string level and treats the adversary as probabilistic polynomial algorithms. The security of protocol in computational model can be stated in terms of probabilistic. At present, most of the automated verification tools of security protocols work in Dolev-Yao model for easier implementation and verification. In fact, the security of a real protocol relies on the security of the involved cryptographic assumptions. Thus, the proofs in Dolev-Yao model are not computational sound.

CryptoVerif is an automated security prover of cryptographic protocols designed by Bruno Blanchet [3]. It uses a game-based proof technique and does not rely on the Dolev-Yao model but the computational model. Till now, a number of security protocols have been proved by CryptoVerif successfully. The rich enough predefined cryptographic primitives of CryptoVerif allow users to describe complex protocols and carry out the verification of simple protocols easily. It will be great helpful to the research of security protocols.

1.1 Aim and Objectives

The aim of this project is to carry out the security proof of 1-out-of-2 oblivious transfer protocol in the semi-honest model with using CryptoVerif.
For this aim, the following objectives are achieved:

1. Explore how CryptoVerif work
2. Investigate how to transfer the cryptographic primitives into the equivalences
3. Record and analyse the generated proofs
4. Draw conclusion from the records

1.2 Organisation of this dissertation

This thesis consists of six main parts.

In the first section, there is a general introduction of this project. The aim and objectives of this project will be also introduced in this part.

In the second section, we will first talk about the definition and functionalities of security protocols. Then we will introduce the two main approaches of analysing security protocols. After that, a brief introduction of automated security proof will be given.

In order to make readers understand the proofs which will be given in the fifth section, we will talk about the features and the basic semantics of CryptoVerif in the third section. At the end of this section, the verification procedures will be illustrated.

In the fourth section, we will first introduce the 1-out-of-2 oblivious transfer protocol which will be proved with using CryptoVerif and have a preliminary analysis of the protocol. Then we will show how to use CryptoVerif to prove the security of 1-out-of-2 oblivious transfer protocol. Firstly, the involved cryptographic primitives will be explained. Meanwhile, we will show how to define these cryptographic primitives in CryptoVerif. After that, the proof of 1-out-of-2 oblivious transfer protocol will be carried out by two steps: first, verify the security of the receiver, after that, verify the security of the sender. Finally, there will be a summary of the equivalences which is concluded from the experiments.

In the last section, the evaluation of the work and conclusion will be given.

2 Security protocol

Security protocols, which are also called cryptographic protocols, are widely used to protect information security on the internet. They can be real or abstract (academic) protocols. The main functionality of security protocols is to provide secure services for different parties in the open network which is insecure. For example, assuming there are two people playing chess though making telephone calls. It is possible because both of the players have not any private (secret) information: the positions of pieces are public. What they need to do is to take turns to tell the other player what he wants to do next. But if the two players are playing poker though the telephone calls. The situation becomes more complicated. They have private information (hand) and public information (deck). In this case, some questions should be thought about: (1) how to share the information of deck between the two players? (2) who deals the cards? (3) if the dealer claims that he got a royal flush, can the other player trust him? It seems that playing cards game though telephone calls is impossible, they need help of props or a trusted third party. But actually, it can be realised with using a security protocol. The protocol shuffles the deck fairly, deals cards randomly and guarantees the cards' information is undeniable and believable. In this way, the game is possible.

To achieve different security goals and to satisfy the required security standards, security protocols are integrated in various internet services. But at the present time, the analysis of security protocols is still a problem which is not yet solved. There are many insecure protocols had ever been long-term used as the secure protocols. The security risk will cause economic loss or even disclosure of state secrets. Therefore, the design and verification of security protocols are equally important issues.

2.1 Overview

As O.Goldreich's stated [8], the design of any cryptographic scheme can be seen as the design of security protocols for the desired functionalities. The research of security protocols is an important part of modern cryptography. A multi-party security protocol consists of a specified random process that maps m inputs to m outputs. The m inputs of the process can be seen as the local inputs from m parties, whilst the m outputs are the computation results which will be sent to the target parties. The random process implemented some desired functionalities. The purpose of the m parties is to send their local inputs to the corresponding target parties. If the m parties trust each other, they can send their inputs directly to the targets. Otherwise, some trusted parties are required. The local inputs should be sent to the trusted parties first (the trusted parties can be external or just some parties among them), and then the trusted parties will execute the computation of the process and send the results to the corresponding parties individually. In practice, the trusted parties may not exist. However, the security protocol can emulate a trusted party by the participants themselves, even though they do not trust the other parties.

2.2 Security services

The purpose of using security protocols is to provide security services in the open network. Generally, there are mainly five kinds of security services:

Access control provides protection against unauthorized access to the resource. The operations on the data are restricted. The operations on the data including: reading, writing, executing and their variations or extensions.

Data confidentiality provides protections of data transmission. The data are hidden by encipherment to avoid unauthorized disclosure. Cryptography and steganography are the two main techniques used to protect data confidentiality.

Data integrity protects the validity of the transmitted messages. It provides assurance of data that has not been modified by an adversary. The received data are exactly the original one sent by the sender. It may protect the whole message or just a part of the message.

Authentication includes two kinds of services: entity authentication and data authentication. Entity authentication provides authentication of the participants during the connection. Data authentication provides assurance that the received data is as it claimed to be.

Nonrepudiation protects against repudiation of the participants and the data involved in transactions. Both the sender and the receiver cannot deny a transmitted message. The sender cannot deny that the message was sent by him. Meanwhile, the receiver cannot deny he has received the message.

2.3 Two-party protocol

The two-party protocol is an important case of multi-party protocols. According to the definition of two-party protocol from O.Goldreich [8], a two-party protocol maps a pair of inputs from the two parties to a pair of outputs to the two parties. The desired process of the protocol is called the functionality of the protocol. It is denoted as:

$$f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$$

For each pair of inputs (x, y), the outputs of $f(x, y)$ are also in a pair. The party whose input is x, will obtain the first element of $f(x, y)$ and the other party who holds the input of y will obtain the second element.

2.4 Two approaches of analysing security protocols

Since 1980s, the analysis of security protocols has been carrying out under two main approaches: computational approach and symbolic approach. Computational approach adopts a more concrete model, whilst symbolic approach focuses on abstraction level of the protocol. There is a big gap between these two approaches.

2.4.1 Symbolic (Dolev-Yao) approach

Under the symbolic approach, the underlying cryptographic primitives are considered as ideal black boxes which are unbreakable and the messages are seen as terms. The adversary here is modeled as a Dolev-Yao attacker: the attacker has no computational limitation and takes control of the whole network. However, its actions are restricted - the attacker can only use the unbreakable cryptographic primitives which are involved in the protocol.

In a nutshell, the analysis under this approach focuses on the very high level: security protocols are modeled at the abstraction level and makes very strong assumptions of the adversary. The details of implementation are abstracted away. It is easy to use and captures most common mistakes in security protocols. This makes automatic verification possible. Currently, there are some automatic security protocols provers (e.g. ProVerif) are under symbolic approach. But there are two drawbacks: for the first thing, this approach depends on strong assumptions of the underlying primitives. It is actually not realistic; secondly, the proof does not guarantee the security of the protocol. Because even though a protocol is proved secure in the abstract level, an adversary in the lower level may be capable of attacking it. So the proof cannot provide any computational soundness.

2.4.2 Computational (cryptographic) approach

On the other hand, the computational model is based on complexity theory. The length of the random nonce is determined by the security parameter. Under the computational approach, messages are seen as bit strings, whist cryptographic primitives are seen as functions which map bit strings to bit strings. The primitives, which operate on messages, satisfy certain security properties. An adversary in computational model is a probabilistic polynomial-time Turing machine: it has computational limitations, but capable of breaking cryptographic operations. The proof in this model is to show if the adversary can break the scheme with significant probability in a reasonable time. It is usually carried out manually.

In computational model, security protocols are close to the real protocols. In contrast to the Dolev-Yao model, the attacker will not always perform the predefined actions of the protocols. Analysis under this approach is on the bit-string level and the security of protocols relies on the underlying algorithmic assumptions (e.g. RSA problem, Discrete Logarithm problem). Reductionist proof is the main tool here: if there is an adversary that can break the scheme, this adversary can also be used to solve the assumption of the scheme. However, using reduction on analysis of medium-sized protocols is really difficult.

2.4.3 Automatic security proofs

With the development of cryptography, the automated security provers have been surfacing in the recent decades, they provide a new way of security proofs. The goal of automated security proofs is to carry out proofs of security protocols automatically, in order to avoid human errors and to make the complex protocols verifiable. Until now, most of the available automated security provers work in the Dolev-Yao model. In this model, the protocols are easier to be fomalised and the verification can be easier to complete.

ProVerif, which is also designed by Bruno Blanche, is one of the available tools [18]. It has been successfully applied to many protocols. ProVerif also has been used by researchers who study security protocols. It can be used to prove:

Secrecy: to check if the protocol preserves the secrecy of variables.

Authentication: to verify the authentication in the form of “if an event is executed, and then the other relevant will also be executed”.

Strong secrecy: to check if the adversary can see the difference if the value of the secret changes.

Equivalences between processes that differ only by terms: to check if the two processes are testing equivalent - the adversary cannot distinguish the two processes.

ProVerif uses Horn clause to represent the processes (protocols), this approach makes fully automated proofs workable. But the verification result is only either "proof succeeded" or "proof failed", it cannot show how secure the protocol is.

To obtain the automated proofs with computational soundness, currently, there are two approaches: indirect approach and direct approach.

Indirect approach: make automated proof in Dolev-Yao model, and then use a sound result that can derive proofs in computational model from the proofs in the Dolev-Yao model.

Direct approach: make automated proof in the computational model directly.

The indirect approach was proposed by Abadi and Rogaway and has been applied to the proof of encryption scheme [1]. But it has limitation: the Dolev-Yao model and the computational model do not exactly coincide, hypotheses are necessary to be added.

Laud pioneered the direct approach [10, 11]. He designed an automatic analysis for a shared-key encryption protocol. Later, Laud designed a type system for checking the secrecy of message handled by protocols in the computational model [12]. The calculus of CryptoVerif is inspired Laud's work. It works directly in the computational model.

2.4.4 Automated proof of OAEP

Since the original version of Optimal Asymmetric Encryption Padding (OAEP) was purposed by Bellare and Rogaway [2], the research of it has never been stopped. The original version of OEAP was claimed to be secure against chosen ciphertext attack in the random oracle model. In 2001, OAEP is proved secure under the RSA assumption by Fujisaki, Okamoto, Pointcheval and Stern [7]. However, Shoup proved OAEP is insecure and suggested an improved scheme OAEP+ in the next year[16]. Later, some studies indicated that it is impossible to prove IND-CCA2 security of RSA-OAEP [15]. More recently, semantic security of OAEP scheme is claimed to be proved with using an automated security prover CertiCrypt by Bëguelin, B Grëgoire, S. Heraud, G. Barthe and F. Olmedo [9]. The results will be released in this year.

3 Introduction to CryptoVerif

3.1 Overview

CryptoVerif is an automatic security protocol prover designed by Bruno Blanchet implemented in Ocaml [3]. It works directly in the computational model without considering Dolev-Yao model. Thus, the proofs are computationally sound. CryptoVerif has the following functionalities:

1. Prove secrecy of variables are preserved by the protocol
2. Check correspondence properties
3. Provide a generic mechanism for specifying properties of cryptographic primitives
4. The generated proofs are proved by sequences of games that just like the manual proofs given by cryptographers
5. Evaluate the extra boundary of probability for an adversary successful in attacking against the protocol

CryptoVerif takes the script of the description of a security protocol as input, and checks if there are some secrets can be leaked in the processes. It operates in two modes: fully automatic mode and interactive mode. Fully automatic mode can be used for general purposes, whilst the interactive mode, which requires users to input commands to specify the order of transformations to perform, is suited for analysing the protocols with asymmetric cryptographic primitives.

3.2 The proof techniques

Reductionist proof is the main tool of analysis in the computational model. It is used in producing a reduction first, and then to prove that the view of an adversary provided by the reduction is indistinguishable from the view of the adversary in the real attack. It is difficult and error-prone. Victor Shoup proposed to prove it by small changes by using a sequence of games[17]. The first game starts from the real attack, and then modifies a few. The modification will involve either a statistical or a computational distance, but from the view of the adversary, the two games are indistinguishable. The modifications of the game can be seen as "rewriting rules" They may consist of renaming some variables, moving down the assignments and some other operations which will not change the distribution. Thus, the modification may not bring any difference and the distributions are statistically indistinguishable. The rewriting rule may be true under a computational assumption only: then appears the computational indistinguishability. The game-based proof is widely used in cryptographic proofs. It is easier to employ and less error-prone.

CryptoVerif produces proofs presented as sequences of games that like manual proofs carried out by cryptographers:

1. The initial game is the real protocol
2. The games are described as processes using formal methods techniques
3. The game goes from one game to be transforming to the next by syntactic transformations or by applying the definition of security of a cryptographic primitive. The transformations will bring negligible difference of probability of success of an attack
4. CryptoVerif keeps applying rewriting rules until the adversary's success probability equals to 0
5. The final game is ïdealänd the security property is obvious from the procedures of the game

The games are formalized in a process calculus. This calculus is inspired from the pi calculus[3]. The semantic is purely probabilistic. All processes run in polynomial time: there are polynomial number of copies of processes and the length of messages on channels bounded by polynomial. The main tool for specifying security properties is observational equivalence [4]. Two processes (games) Q_1 and Q_2 are observational equivalence when the adversary has a negligible probability of distinguishing Q_1 from Q_2, denotes $Q_1 \approx Q_2$. CryptoVerif transforms a game Q_0 into the next game with using observational equivalences or syntactic transformations. The methods of syntactic transformations are:

- Single assignment renaming: when a variable is assigned at several places, rename it with a distinct name for each assignment
- Expansion of assignments: replacing a variable with its value
- Move new: move restrictions downwards in the game as much as possible when there is no array reference to them.

Finally we can obtain a sequence of games $Q_0 \approx Q_1 \approx \ldots \approx Q_m$, which implies $Q_0 \approx Q_m$. If some equivalences or security properties holds with overwhelming probability in Q_m, then it also holds with overwhelming probability in Q_0.

CryptoVerif uses arrays to replace lists. The values of all variables during the execution of the process are stored in arrays. The arrays' length is fixed and each cell of the arrays can be only assigned at most once. In CryptoVerif, only variables with the current indexes can be assigned. Variables may be defined at several places, but only one definition can be executed for the same indexes.

The proof strategy of CryptoVerif is based on advice. It is also a key feature to make automatic proofs available. CryptoVerif tries to execute each transformation indicated in the definition of a cryptographic primitive. When it fails, CryptoVerif will try to analyse the reason of the failure occurred and to suggest other transformation to be applied before that transformation. If it works, CryptoVerif will continue to execute the syntactic transformation or else suggest other syntactic transformations and to execute it. If it doesn't work, CryptoVerif will continue to find the next one.

3.3 The advantages of using CryptoVerif

Here are the benefits of using CryptoVerif:

1. Works directly in the computational model, the proofs are computationally sound

2. Predefined a number of cryptographic primitives, the users do not need to redefine those primitives but just use them as a basis to build definitions of new primitives by copying and modifying them

3. Easy to describe the simple protocols with using the predefined primitives

4. Rich enough to describe complex protocols which based on computational assumptions

5. The proofs are presented as sequences of games; Even if the proof failed, CryptoVerif will still output a sequence of games which are generated during the execution of verification. The users can still analyse the results and understand why the proof failed

6. Efficient: generating the proof of FDH only requires 14ms on a Pentium M 1.8 GHz machine [3].

3.4 The experiments and results

CryptoVerif has been tested on the original and corrected versions of the following protocols (Provided by the designer [3]):

- Otway-Rees (shared-key)
- Yahalom (shared-key)
- Denning-Sacco (public-key)
- Woo-Lam shared-key and public-key
- Needham-Schroeder shared-key and public-key
- Full domain hash signature (with D. Pointcheval)
- Encryption schemes of Bellare-Rogaway'93 (with D. Pointcheval) Shared-key encryption are implemented as encrypt-then-MAC, using an IND-CPA encryption scheme. (For Otway-Rees, a SPRP encryption scheme, a IND-CPA + INT-CTXT encryption scheme and a IND-CCA2 + IND-PTXT encryption scheme are also considered)

In a few cases, CryptoVerif failed to prove the security properties (Provided by the designer [3]):

- Needham-Schroeder public-key when the exchanged key is the nonce N_A
- Needham-Schroeder shared-key: fails to prove that $N_B[i] \neq N_B[i] - 1$ with overwhelming probability, where N_B is a nonce
- Showing that the encryption scheme $\varepsilon(m, r) = f(r) \parallel H(r) \oplus m \parallel H(m, r)$ is IND-CCA2

In the rest of this section, we will introduce the semantics of CryptoVerif and describe the verification procedures.

3.5 The two front-ends of CryptoVerif

There are two kinds of front-ends in CryptoVerif: *channels* and *oracles.*

To execute CryptoVerif, users need to indicate in what front-end style their scripts are written:

```
./cryptoverif -in <frontend> <filename>
```

The <*frontend*> can be *channels* or *oracles.*
The main differences between these two front-ends are as follows:

- The *channels* front-end uses channels to represent security protocols while the *oracles* front-end uses oracles
- The main process of *channels* front-end consists of input processes and output processes whilst *oracles* front-end includes the body of oracles and the definition of oracles
- The calculus used in *channels* front-end is inspired by the pi calculus, while the *oracles* front-end uses a calculus which is closer to cryptographic games
- Semantics

The proof in section 5 on page 29 is done in the *channels* front-end, so we will introduce the semantics of the *channels* front-end in this section.

3.6 Input and output processes

In channels front-end, there are two kinds of processes: input processes and output processes. The input process <*iprocess*> is to receive a message from a channel; the output process is to send a message to a channel.

To receive a message on a specified channel, the users need to use an input statement: *in*(<*channel*>*,* <*pattern*>)*;* <*oprocess*>, the parameter <*channel*> specifies the target channel and the <*pattern*> indicates the pattern of the message. The specific data types should be given in the <*pattern*>. After received a message from the specified channel, the output process <*oprocess*> will be executed. The form of a output statement is similar to the input statement: *out*(<*channel*>*,* <*terms*>)*;* <*iprocess*>. After a message <*terms*> was sent to the channel <*channel*>, the input process <*iprocess*> will be executed. The input statement is always followed by the output process, while the output statement is followed by the input process. A protocol should have at least a single output process and more than one input processes. When the output process executes *out*(<*channel*>*,* <*terms*>)*;* <*iprocess*>, then it will look for an input on the same channel in the available input processes. If there is no such a process, it will choose an input process randomly with uniform probability.
An output process contains:

- The other ouput processes

- Events
- Definitions
- Assignments
- If statements
- Find statements
- Output statement

The output statement must be at the end of the output process.
An input process contains:

- The other input processes, which may have N copies in parallel where N is defined as the parameter before use
- The 0 process, which means to do nothing
- Input statement

The input statement must be also at the end of an input process.

Example 1 :

```
in(c1, a:T);
new b:T;
event E;
if (a = b) then
let c:T = a in
out(c2, b);
```

In this example, *in(c1, a:T);* is an input process that only contains an input statement. It is followed by an output process: *new b:T; event E; if (a = b) then let c:T = a in out(c2, b);*. The output process contains a random variable definition, an event execution, an if statement, an assignment and an output statement.

3.7 The semantics of CryptoVerif

In this section, we will talk about the semantics of CryptoVerif.

3.7.1 The basic semantics

Declaration of variables: *new x:T;M* selects a random variable of type P with uniform probability, then stores it in variable x and return the result of M.

Example 2

```
new a:T;
new b:T;
```

In this example, a random variable a of type T is declared. And then *new b:T* will be executed, but its return value will not be used in *new a:T;*.

Assignment: *let* p = M *in* P *else* Q tries to decompose the term M according to the pattern p. If M can be decomposed, then the result of P will be returned, otherwise Q will be returned instead. The pattern p can be:

- Variable with its type: *x:T*, the value of M will be assigned to x and other else branch will be omitted
- Function which is declared [compose]: $f(p_1, p_2, \ldots, p_n)$, if there is a set of $M_1, M_2, \ldots, M_n$ which satisfy $f(M_1, M_2, \ldots, M_n) = M$, P will be returned, otherwise Q will be returned
- Tuple: $p_1, p_2, \ldots, p_n$, to check if M can be decomposed to the given tuple and its type
- A certain bitstring: $= M'$, to check if M equals to M'

Example 3 :

```
let a:T = b in
let f(T) = b in
out(c1, b);
```

In this example, a random variable a of type T is assigned with the value of b, if the assignment is successful, the second assignment will be executed. The second assignment tries to find a variable t of type T that $f(t)$ equals to b. If the variable is found, the following output statement will be executed.

If statement: *if condition then P else Q* returns the result of P if *condition* is true or else returns Q.

Example 4 :

```
if (a = b) then
(let c:T = b)
else
  out(c1, b).
```

In this example, if the condition *(a = b)* is true, c will be assigned with the value of b, otherwise b will be output on channel $c1$.

The find statement: *find [unique]* FB_1 *suchthat condition*$_1$ *then* P_1 *else* Q_1 *orfind . . . orfind* FB_m *suchthat condition*$_m$ *then* P_m *else* Q_m each find branch FB_j can be of the form $u_j \leq n_j, \ldots, u_{j_m} \leq n_{j_m}$. It carries out a sequence of arrays look-up to find m cells that satisfy each condition of every find branch. If the condition in a find branch is true, the value of P_j will be returned, or else the value of Q_j will be returned. In CryptoVerif, the variables are usually defined as arrays; the find statement is useful to find variables that satisfy some conditions. When the option [unique] presents, it means there is at most one branch and a single value of the indices makes the corresponding condition to be true.

Example 5 :

```
find [unique] i <= N suchthat defined{b[i]} && (b[i] = m)
then event E else yield.
```

This example tries to find the only one cell which is defined and equals to m in the array b. If the cell can be found, event E will be executed, or else output nothing.

Event declation: *event e(*$M_1, \ldots, M_n$*);* P executes an event e with parameters $M_1, \ldots, M_n$, then return the result of P. The event e must be declared by the event declaration *event e(*$M_1, \ldots, M_n$*).* before use, the arguments are optional. The queries of events can be used to prove different security properties by querying "If an event is executed, the relevant event will also be executed".

Example 6 :

```
event e(A,B).
```

It declares an event e which has two arguments of type A and B respectively.

Parameter declaration: *param* $n_1, \ldots, n_m$. declares m parameters $n_1, \ldots, n_m$ that will be used in the script. The parameters are usually used to represent the replications of the processes and variables. They also measures maximum number of the queries which can be used in the probability formulas to calculate the probability of an attack.

Example 7 :

```
param N1, N2.
```

It declares two parameters $N1$ and $N2$.

Probability declaration: *proba p* declares a probability p which will be used in the script.

Example 8 :

```
param POW1,POW2.
```

It declares two probabilities $POW1$ and $POW2$.

Constants declaration: *const* $c_1, \ldots, c_n$*:T.* declares n constants $c_1, \ldots, c_n$ of type T.

Example 9 :

```
const hostA, hostB:bitstring;
```

It declares two constants *hostA* and *hostB* of type bitstring.

Channel declaration: *channel* $c_1, \ldots, c_n$. declares n channels which will be used in the script. Different channel names for each input and output is recommended for more precisely capturing from which copy of inputs the adversary receives a message and to which copy of outputs he sent a message.

Example 10 :

```
channel c0, c1, c2.
```

It declares three channels $c0$, $c1$ and $c2$.

Forall statement: *forall* $x_1 : T_1, \dots, x_n : T_n; M.$ specifies a simplification rewriting rule: for all the value of $x_1, \dots, x_n$ in types $T_1, \dots, T_n$, the term M is always true. The term M must be a simple term which has not any array access. It can be an equality $M_1 = M_2$ or an inequality $M_1 <> M_2$. If it is an equality, the term M_1 will be rewritten into M_2; if it is an inequality, CryptoVerif will rewrite $M_1 = M_2$ into false and $M1 <> M2$ will be rewritten into true. In the simplification stage, CryptoVerif will perform the simplification by rewriting the terms according to the rewriting rule specified by the forall statements.

Example 11 :

```
forall x:D, x':D;(f(x)=f(x'))=(x=x').
```

This forall statement indicates the function f is injective. $f(x) = f(x')$ will be rewritten into $x = x'$.

Type definition: *type T [type options].* defines a type T. The types of CryptoVerif consist of sets of bitstrings or/and a special symbol $\perp$ which indicates fail decryptions, etcs. The type options are optional, they can be:

- bounded indicates the type is a finite set of bitstrings and $\perp$
- fixed means the type is a set of bitstrings with a certain length
- large means the type is large enough, the collisions between two elements of the type T is negligible
- password means the probability of collisions between two elements of the type T is not negligible and the probability of guessing the right password is also not negligible

Example 12 :

```
type A [large, fixed].
```

It declares a type A which is large and fixed.

3.7.2 Declation of functions

A function can be declared by *fun* $f(T_1, \ldots, T_n)$: T *[function options]*. It means that the function f takes n argument of type $T_1, \ldots, T_n$ respectively and the return value of type T. The function options can be:

- compos: f is poly-injective which means f is injective and its inverse can be computed in polynomial time
- decompos: f is an inverse of a poly-injective function and it is unary
- uniform: f is unary and it maps a uniform distribution to a uniform distribution
- commut: f is commutative and binary. The two arguments of f should be of the same type.

There is no definition of a function's body in CryptoVerif, but its security properties can be formalized in the equivalences. Thus, the definition of a function's body is unnecessary.

Example 13 :

```
fun xor(bitstring, bitstring):bitstring [commut].
```

In this example, a commutative function *xor* is declared. It takes two arguments of type *bitstring* and return a value of type *bitstring*.

3.7.3 Definition of equivalences

In CryptoVerif, security properties of cryptographic primitives are defined as equivalences. Actually, common cryptographic primitives are predefined in CryptoVerif. Users can still define their own equivalences by *equiv L <= (p) => R*. It means that a probabilistic Turing machine can distinguish L from R with probability at most p in time *time*. In another word, CryptoVerif can transform L into R with probability lose p during the game transformations. Both L and R are function groups (in the oracles front-end, L and R are oracles groups). Both L and R are sets of functions. L should be in the form of *<function group> [function mode]* while R should be be *[manual] <function group>*. The *<function group>* can be:

1. $(x_1 : T_1, \dots, x_n : T_n)[n][required]$ -> M means a function takes n arguments $x_1, \dots, x_n$ of types $T_1, \dots, T_n$ and return the result of M. The number in the square brackets is the priority of the function. Smaller the number n, the higher priority the set of functions has. In the case $n = 0$, the function has the highest priority. When there are several functions can be used during the game transformation, the function with highest priority is most likely to be chosen. If this option does not exist, CryptoVerif assumes $n = 0$. If the option [required] exists, it means CryptoVerif will apply the equivalence during the game transformation only when the function is used at least once in the game.

2. $!i \leq N\ new\ x_1 : T_1; \dots;\ new x_m : T_m; (FG_1, \dots, FG_n)$ means there are N replications of a process that selects random numbers $x_1, \dots, x_n$ of type $T_1, \dots, T_n$. These random numbers make function groups $(FG_1, \dots, FG_n)$ available. The variables are described as array and must be declared under the replication $!i \leq N$. N should be declared in the parameter declation.

The function mode is optional; it can be one of the following:

1. [exist] means at least one function group in this equivalence will be transformed in the games. If there is no function option in the left-hand side, it will be [exist] by default

2. [all] means all of the occurrences of the functions in the games will be transformed

When the option [manual] exists in R, it means this equivalence will not be transformed by CryptoVerif automatically. The equivalence can be used only in the interactive mode by giving the manual instruction.

The equivalences can be defined in both *channels* and *oracles* front-end style. The equivalences defined in *oracles* front-end style will still work well in the *channels* front-end.

Example 14 :

```
equiv !nx new a:bitstring; (b:bitstring) -> xor(a,b)
      <=(0)=>
      !nx new a:bitstring; (b:bitstring) -> a.
```

This equivalence defines the property of *xor* function for bitstring. Given a bitstring b, an adversary can distinguish the result of $xor(a, b)$ from a with probability 0, where a is a random bitstring.

3.7.4 Definition of queries

A query can be used to indicate which security property the users desire to prove. It is defined by *query <queries>*. The queries can be:

1. *secret1 x*; it proves the one-section secrecy of a variable x. This query proves that the adversary cannot distinguish x from a random number by a single test query. x must be a variable defined in the script.More exactly, x is defined in the script and there is a set of variables S. Only the variables in the set S depends on x. The output messages and the control flows do not depend on x.

2. *secret x*; it proves the secrecy of a variable x. This query proves that the adversary cannot distinguish x from an array of random numbers by several test queries. It should satisfy the one-section secrecy and $x[i]$ and $x[i']$ should be defined by different restriction when $i \neq i'$.

3. $x_1 : T_1, \ldots, x_n : T_n$*; event* P ==> Q; it proves that for all variables that occurred in P, if P is true, there exist variables in Q makes Q to be true. Those variables do not occur in P. Both P and Q can be a single event or several events. When [inj:] exists before an event, the system proves an injective correspondence. If [inj:] occurred before ==>, it should also occur after ==>. Q can also be two boolean values: *true* or *false*. In this case, the system proves if the event(s) P will happen.

Example 15 :

```
query secret x.
query event inj:eventA(a,b) ==> inj:eventB(a,b).
query event forge ==> false.
```

In this example, the first query is to prove the secrecy of a variable x. The second query is to prove the injective correspondence of *eventA* and *eventB*. The third one is to prove the event *forge* will not happen in the protocol.

3.7.5 Using interactive mode

To use interactive mode, a declaration *set interactiveMode = true.* should be declared in the script. In this case, CryptoVerif will not carry out the verification of the protocol automatically. Instead, manual instructions are required. The commands will be displayed by typing *help* or ? in the interactive mode. Users can also specify the commands in the script by using *proof* $\{command_1; \ldots; command_n\}$ where the $command_1, \ldots, commands_n$ are the commands of interactive mode.

3.7.6 Using the predefined cryptographic primitives

Several common cryptographic primitives are predefined as macros in CryptoVerif. The macros may contain: type declation, function declation, parameter declation, probability declation and forall statement. The users can add these predefined cryptographic primitives to the script by expending the macros: *expand <macro name> (<macro arguments>)*. The available predefined cryptographic primitives are as follows: *probabilistic symmetric encryption scheme, IND-CPA and INT-CTXT probabilistic symmetric encryption scheme, IND-CCA2 and INT-PTXT probabilistic symmetric encryption scheme, deterministic symmetric encryption scheme, pseudo-random permutation, block cipher in the ideal cipher model, UF-CMA MAC, SUF-CMA MAC, IND-CCA2 probabilistic public-key encryption scheme, UF-CMA probabilistic signature scheme, SUF-CMA probabilistic signature scheme, hash function in the random oracle model, collision resistant hash function, one-way trapdoor permutation, one-way trapdoor permutation with random self-reducibility* and *pseudo-random function*. The macro arguments can be some of a type, a function, a replication parameter or a probability. Once the macro is expended, the users can use them in the script.

3.7.7 The verification procedures of CryptoVerif

The verification of a security protocol will be carried out as follows:

Step 1: Initial simplification it includes:

1. Expand *if, let and find* statements
2. Use simplification rules to simplify the game
3. Move all binders
4. Remove useless assignments

Step 2: Queries check check if there are some queries remain unproved. If no, output the result and the proof which represents as a sequence of games. If yes, move to Step 3

Step 3: Equivalences application try to apply equivalences. If there is no equivalence can be applied output the result: queries cannot be proved. Otherwise, try to apply one of the available equivalences. Once an equivalence is applied successfully, move to Step 4, or else try to find another equivalence until one equivalence can be applied or there is no equivalence left (in this case, output the result: queries cannot be proved).

Step 4: Simplificaition it concludes:

1. Use simplification rules to simplify the game
2. Move all binders
3. Remove useless assignments

Then move back to Step 2.

4 Oblivious transfer protocol

Michael Rabin introduced oblivious transfer protocol in 1981[14]. The purpose of oblivious transfer protocol is to send some secret information to the target, but the sender does not know what the receiver got. Oblivious transfer is a useful tool of security protocols.

4.1 Rabin Oblivious Transfer

Michael Rabin invented a protocol with some curious properties [14]. This protocol is a kind of formalization of ”noisy wire” communication. In this scheme, the sender will send a message to the receiver with probability of $\frac{1}{2}$. Thus, Rabin's scheme was named $\frac{1}{2}$-OT. This scheme is based on RSA cryptosystem.

The scheme works as follow:

1. The sender finds two large prime numbers p, q and calculate $N = p \cdot q$ then reveals n to the receiver

2. The receiver picks a random element x and sends $t = x^2 modN$ to the sender

3. The sender computes $s = \sqrt{t}$. There will be four possible roots: $x, -x, y$, and $-y$. The sender selects one of these roots randomly and sends s to receiver

4. If $s = \pm x$ then the receiver can learn nothing from the sender, but if $s = \pm y$, the receiver can calculate p and q by calculating $gcd((x-s), N)$ and $gcd((x+s), N)$

The probabilities of $s = \pm x$ and $s = \pm y$ are likely the same because x is chosen randomly. In another word, the receiver can learn p and q with probability of $\frac{1}{2}$. But anyway, the sender will never know if the receiver learned the secret or nothing.

4.2 1-out-of-k Oblivious Transfer

The oblivious transfer protocol was developed by Oded Goldreich later[8]. He introduced a more useful form of the oblivious transfer protocol: the 1-out-of-k Oblivious Transfer protocol. The protocol is constructed based on trapdoor permutation. There are two participant in the protocol: the one who holds a sequence of secrets $(\sigma_0, \sigma_1, \ldots, \sigma_{k-1})$ is called the sender, the other participant who hold a query number $i \in \{0, 1, \ldots, k-1\}$ is called the

receiver. The purpose of this scheme is to transfer the chosen secret to the receiver. During the processes, the sender cannot get to know about which of the secrets was required by the receiver, whist the receiver cannot obtain any information about the other secrets of the sender.
The 1-out-of-k Oblivious Transfer protocol works as follow:

The sender's input: $(\sigma_0, \sigma_1, \ldots, \sigma_{k-1}) \in 0, 1^k$
The receiver's input: $i \in 0, 1, \ldots, k-1$
The sender's output: nothing
The receiver's output: σ_i

1. The sender chooses a one-way trapdoor permutation f and the corresponding trapdoor t, and then sends f to the receiver

2. The receiver uniformly and independently selects k elements $x_0, x_1, \ldots, x_{k-1}$ in the domain of the one-way trapdoor permutation f. Then set $y_i = f(x_i), y_j = x_j$ for $j = 0, 1, \ldots, k-1$ and $j \neq i$. Send $(y_0, y_1, \ldots, y_{k-1})$ to the sender

3. The sender computes $z_j = f^{-1}(y_j)$ for $j = 0, 1, \ldots, k-1$ with the trapdoor t. Then computes the hard-core predicate $B(z_j)$ of the trapdoor permutation f and sends $(\sigma_0 \oplus B(z_0), \sigma_0 \oplus B(z_1), \ldots, \sigma_{k-1} \oplus B(z_{k-1}))$ to the receiver

4. The receiver obtains σ_i via computation:

$$\begin{aligned}
\sigma_i \oplus B(z_i) \oplus B(x_i) &= (\sigma_i \oplus B(f^{-1}(y_i)) \oplus B(x_i) \\
&= (\sigma_i \oplus B(f^{-1}(f(x_i)) \oplus B(x_i) \\
&= (\sigma_i \oplus B(x_i) \oplus B(x_i) \\
&= \sigma_i
\end{aligned}$$

The 1-out-of-k oblivious transfer protocol relies on the collection of enhanced trapdoor permutation and the hard-core predicate of it. It assumes that calculating of $f^{-1}(x_j)$ is not feasible and finding $B(f^{-1}(x_i))$ will not be better than guessing randomly. These assumptions guaranteed the secrecy of both the sender and the receiver will not be leaked.

4.3 1-out-of-2 Oblivious Transfer

In the case of k = 2, it will be a 1-out-of-2 Oblivious Transfer. 1-out-of-2 Oblivious Transfer was suggested by Even, Goldreich and Lempel [6].
The 1-out-of-2 Oblivious Transfer protocol works as follow:

The sender's input: (σ_0, σ_1)
The receiver's input: $i \in 0, 1$
The sender's output: nothing
The receiver's output: σ_i

1. The sender chooses a one-way trapdoor permutation f and the corresponding trapdoor t, and then sends f to the receiver
2. The receiver uniformly and independently selects two elements x_0, x_1 in the domain of the one-way trapdoor permutation f. Then set $y_i = f(x_i), y_{1-i} = x_{1-i}$. Send (y_0, y_1) to the sender
3. The sender computes $z_j = f^{-1}(y_j)$ for every y with the trapdoor t. Then computes the hard-core predicate $B(z_j)$ of the trapdoor permutation f and sends $(\sigma_0 \oplus B(z_0), \sigma_1 \oplus B(z_1))$ to the receiver
4. The receiver computes $\sigma_i = \sigma_i \oplus B(z_i) \oplus B(x_i)$

4.4 Preliminary analysis

There are two kinds of cheating behaviors of two-party protocols: semi-honest and malicious. In the semi-honest model, a semi-honest party will completely follow the protocol but keep a record of all his intermediate computations. He tries to cheat with the information he got.

In the malicious model, the cheating party may not follow the protocol and tries to cheat. In the 1-out-of-2 oblivious transfer protocol, assuming the sender is honest, a malicious receiver can cheat easily. After received the one-way trapdoor permutation from the sender, the receiver selects two elements and computes $y_0 = f(x_0), y_1 = f(x_1)$, then sends (y_0, y_1) to the sender. The sender will still compute $z_i = f^{-1}(y_i)$ and sends $(\sigma_0 \oplus B(z_0), (\sigma_1 \oplus B(z_1)$ back to the receiver. Finally, the receiver obtain the two secrets of the sender by computing $\sigma_i = \sigma_i \oplus B(z_i) \oplus B(x_i)$ which equal to σ_0 and σ_1.

A malicious sender can also cheat. In the first step, the sender may send another function f which is not a permutation. For example, the function f maps a larger domain D to a smaller domain D'. Then the honest receiver selects x_0 and x_1 randomly, computes $y_i = f(x_i), y_{i-1} = f(x_{i-1})$ and send (y_0, y_1) to the sender. The sender will know which secret the receiver intends to obtain because y_0 and y_1 are not in the same domain.

The analysis of security of both parties in the semi-honest model will be given by using CryptoVerif in section 5 on page 29.

4.5 Related theories

Here are the advanced cryptography techniques used in 1-out-of-2 Oblivious transfer protocol.

4.5.1 One-way trapdoor permutation

A collection of one-way trapdoor permutations is a set of one-way permutation. Given an index i, the one-way permutation f_i is easy to compute but hard to invert f_i^{-1} without knowing the trapdoor t_i. Currently, the best-known trapdoor families are the RSA and the Rabin families of function. Both of them are related to the problem of factorization.

4.5.2 Hard-code predicate of one-way permutations

The hard-code predicate of any one-way permutation is denoted as B. It is a function output just only 1 bit. Given the value of x, one can calculate $B(x)$ efficiently but there is no efficient algorithm to guess $B(x)$ from $f(x)$ successfully with probability significantly higher than one half. The hard-core predicate of one-way trapdoor permutations can be used to construct public-key encryption scheme.

5 Automated proof of 1-out-of-2 OT protocol

In this section, we will show how to use CryptoVerif to prove the security of 1-out-of-2 oblivious transfer protocol in the semi-honest model. The problems that encountered in the experiments will also be analysed. The 1-out-of-2 oblivious transfer protocol is a two-party protocol, the users' securities can be divided into two parts: security of the sender and security of the receiver.

5.1 Definition of cryptographic primitives

5.1.1 One-way trapdoor permutation

A family of one-way trapdoor permutations on domain D is defined by the following algorithms:

- *The key generation algorithms*: it contains two parts: the public key generation algorithm pkgen and secret key generation algorithm skgen. Given a key seed r as argument, pkgen produces a public key pk by computing $pk = pkgen(r)$ whist skgen generates the corresponding secret key sk by computing $sk = skgen(r)$

- *The evaluation algorithm* f: f output $y = f(pk, x)$ by given a public key pk and an element x where $x \in D$
- *The inversion algorithm* invf: given an element y and the trapdoor sk where $y \in D$, invf outputs the pre-image x of y by computing $x = invf(sk, y)$

The above algorithms should also hold two conditions:

1. Easy to compute
2. Hard to invert

The algorithms are assumed to be efficient which can be computed easily. The second condition "hard to invert" means the one-wayness property. More precisely, it can be defined by: given the public key pk and an element y, the probability of an adversary $\mathcal{A}$ can successfully calculate x' equals to x which is the pre-image of y is negligible in a reasonable time. The one-way trapdoor permutation is predefined in CryptoVerif. To use this cryptographic primitive, we need to expand the macro OW_trapdoor_perm by the macro expanding statement:

```
expand OW_trapdoor_perm(seed, pkey, skey, D, pkgen, skgen, f,
invf, POW).
```

Hence we will have:

- *seed*: the type of key seeds. It must be declared as fixed.
- *pkey*: the type of public key. It must be decladed as bounded.
- *D*: the type of the domain of the permutation. It must be decladed as fixed.
- *pkgen(seed)*: the public key generation function.
- *skgen(seed)*: the secret key generation function.
- *f(pkey, D):D*: the permutation function. It takes *pkey* and *D* as arguments and return the value of type *D*.
- *invf(skey,D):D*: the inverse permutation function. It takes *skey* and *D* as arguments and return the value of type *D*.
- *POW*: the probability of breaking the one-wayness property in time t.

The types listed above and the probability *POW* should be declared before expanding this macro. The functions are defined within this macro, so they can be used directly in the script without declarations. Users can change the identifiers of the types, functions and probability if they want.

The corresponding simplification rules, the equivalences and the involved parameters will be added to the script. Therefore, we will have two equivalences which are defined in this macro.

Equivalence 1: one-wayness

```
equiv !nK new r: seed; (
     Opk() := pkgen(r),
     !nf new x: D; (
       Oy() := f(pkgen(r), x),
       !n2 Oeq(x': D) := (x' = x),
         Ox() := x))
<=(set(proba <= nK * nf * POW(time + (nK - 1.) *
time(pkgen) + (Oy - 1.) * time(f))))=>
      !nK new r: seed; (
     Opk() := pkgen'(r),
     !nf new x: D; (
       Oy() := f'(pkgen'(r), x),
       !n2 Oeq(x': D) := if defined(k) then (x' = x) else false,
         Ox() := let k: bitstring = mark in x))
```

This equivalence is written in the *oracles* front-end style. It defines the one-wayness property of one-way trapdoor permutations. The left-hand side of the equivalence is used to describe the original process. In the first group of oracles, a random variable r of type seed is picked. A public key oracle *Opk* publishes the public key by returning the value of $pkgen(r)$. In the second group of oracles, a random variable x of type D is picked. For every r and x, the following three oracles are available: the oracle *Oy* returns the value of $f(pkgen(r), x)$; the oracle *Oeq* takes x' of type D as argument and returns the boolean value of if $x' = x$; the last oracle *Ox* returns the value of x. The original process can be transformed in to the process described in the right-hand side with probability lost of $n_K \times n_f \times POW(time + (n_K - 1.) \times time(pkgen) + (Oy - 1.) \times time(f))$. The proof of this probabilistic formula can be found in the appendix of [4]. In the right-hand side, the function f and $pkgen$ are replaced by f' and $pkgen'$ respectively to prevent the infinite loop of game transformation because CryptoVerif may keep applying this equivalence with r. The equality oracle *Oeq* is replaced by $Oeq(x' : D) :=$ *if defined(k) then* $(x' = x)$ *else false* while oracle Ox is replaced by $Ox() := let\ k : bitstring = mark\ in\ x$. It means that if an adversary has never called the oracle Ox to obtain the value of x, he cannot judge if x' equals to a given variable x. Once the oracle Ox is called, a variable k of type bitstring will be defined and then the condition *if defined(k)* of oracle Oeq which is under the same replication will become true.

Equivalence 2: the property of permutation

```
equiv !nK new r: seed; (
    Opk() := pkgen(r),
    !nf new x: D; (
      Oant() := invf(skgen(r), x),
      Oim() := x))
<=(0)=>
  !nK new r: seed; (
    Opk() := pkgen(r),
    !nf new x: D; (
      Oant() := x,
      Oim() := f(pkgen(r), x)))
```

This equivalence describes the property of permutation. In both side of the equivalence, for every random variable r of type *seed*, the oracle Opk returns the public key $pkgen(r)$. To every random variable x of type D, the return value of the oracle $Oant$ is replaced by x while the return value of oracle Oim is replaced by $f(pkgen(r), x)$. Because in a permutation, x is uniformly distributed random number, both f and $invf$ are injective. When $y = f(pkgen(r), x)$ is given, its pre-image x is determined. If x is given, $y = f(pkgen(r), x)$ is also determined. Thus, in this equivalence, $invf(skgen(r), x)$ can be replaced by x and x can be replaced by $f(pkgen(r), x)$.

5.1.2 The hard-core predicate

The hard-core predicate based on any one-way permutation isn't predefined in CryptoVerif. Thus, we need to define it by ourselves. The function of hard-core predicate can be defined as follow:

```
fun B(x:D):bool.
```

The function B takes x of type D as argument and return the value of type boolean and we interpret the result as 0 and 1. Next, we need to define the property of hard-core predicate: given a variable $y = f(pk, x)$ of type D, an adversary cannot guess $B(x)$ correctly with probability significant than $\frac{1}{2}$.

For another expression, given y, an adversary cannot distinguish $B(x)$ from a random boolean value. Thus, this equivalence is defined as follow:

Equivalence 3: the property of hard-core predicate

```
equiv !nk new r:seed; (
      () -> pkgen(r),
        !nb new b:bool; (x:D) -> B(invf(skgen(r), x)))
      <=(0)=>
      !nk new r:seed; (
      () -> pkgen(r),
        !nb new b:bool; (x:D) -> b).
```

Because the 1-out-of-2 protocol will be proved in the channels front-end, the rest equivalences will be defined in channels front-end style.

We also need to define the *xor* function for bits by ourselves. The function declaration is:

```
fun xor(bool,bool):bool [commut].
```

The *xor* function takes two arguments of type *boolean*, and the type of the return value is also *boolean*. It is communicative, so it should be declared as [commut]. The *xor* function should also meet the following two conditions:

- $\forall a : bool, b : bool, xor(xor(a, b), b) = a$
- $\forall a : bool, b : bool, c : bool, xor(a, b) = xor(c, b) \to a = c$

Thus, two simplification rules should be added to the script:

```
forall a:bool,b:bool; xor(xor(a, b),b)=a.
forall a:bool,b:bool,c:bool;(xor(a, b)=xor(c,b))=(a=c).
```

Given a single bit b, an oracle take b as input and return the result of $xor(a, b)$ where a is a random bit. From the view of an adversary, the return value of the oracle cannot be distinguished from a random bit a. This property is defined as follow:

Equivalence 4: the property of the *xor* function for bits

```
equiv !nx new a:bool; (b:bool) -> xor(a, b)
      <=(0)=>
      !nx new a:bool; (b:bool) -> a.
```

5.2 Security of the receiver

The only one secret the receiver has is the query bit i. We will prove that there is no information leak in the processes of the protocol. More exactly, the secrecy of the bit i will be proved.

As we know, in the 1-out-of-2 oblivious transfer protocol, the sender will send the one-way trapdoor permutation f and the public key pk to the receiver. Then the receiver will select two random elements x_0 and x_1 from the domain D and send (y_i, y_{1-i}) where $y_i = f(pk, x_i), y_{1-i} = x_{1-i}$ to the sender. After that, no matter what messages will the sender send to the receiver, the receiver will not leak any information about the query bit i to the sender even though the message from the sender to the receiver is not sent. Since clearly, this message does not affect the secrecy of i. Therefore, we need to prove the secrecy of i will not leak by sending (y_i, y_{1-i}) to the sender. In this case, we can use a "simplified" 1-out-of-2 oblivious transfer protocol, it works as follows:

1. The sender selects a one-way trapdoor permutation function f and then send f to the receiver
2. The receiver chooses two elements x_0, x_1 from D, computes $y_i = f(pk, x_i), y_{1-i} = x_{1-i}$ and send (y_i, y_{1-i}) to the sender

Actually, the function f cannot be output on channels. So we will put the selection of one-way trapdoor permutation and the key generation in the main process, so that both the sender (in the case of simplified 1-out-of-2 oblivious transfer protocol, the sender is omitted) and the receiver will know a function f is selected. To prove the security of this protocol, only Equivalence 1 and Equivalence 2 are needed. Because of the criterion for proving secrecy of a variable: the variable should be defined in the script, the query bit i cannot be taken as local input. As an alternative, it is chosen randomly in the process of the receiver.

Then we run CryptoVerif to generate the proof fully automatically. Unfortunately, the secrecy of i cannot be proved. To understand why the script doesn't work, we need to analyse the generated proof.

Problem 1 : *Need to rewrite the equivalence*

There are four games generated by CryptoVerif: the initial game is the real protocol, the games 2-3 are obtained by initial simplification, and then CryptoVerif successfully applied the Equivalence 1 and generated game 4.

The last game we obtained:

```
Game 4 is
in(start, ());
new r: seed;
let pk: pkey = @1_pkgen'(r) in
out(cstart, pk);
in(c1, ());
new i: bool;
if i then
(
  new x0_21: D;
  new x1_22: D;
  let y1_16: D = f(pk, x1_22) in
  out(c2, (x0_21, y1_16))
)
else
  new x0_19: D;
  let y0_17: D = f(pk, x0_19) in
  new x1_20: D;
  out(c2, (y0_17, x1_20))
```

As we can see from the operations record of CryptoVerif, it tried to apply the Equivalence 2 but failed. We should consider how to make Equivalence 2 can be applied in the game transformation.

```
equiv !nK new r: seed; (
    Opk() := pkgen(r),
    !nf new x: D; (
      Oant() := invf(skgen(r), x),
      Oim() := x))
<=(0)=>
  !nK new r: seed; (
    Opk() := pkgen(r),
    !nf new x: D; (
      Oant() := x,
      Oim() := f(pkgen(r), x)))
```

In this equivalence, for every key seed r and a random variable x of type D, the oracle which returns the term of $invf(skgen(r), x)$ will be replaced by x. The variable x will be replaced by $f(pkgen(r), x)$. In the simplified 1-out-of-2 oblivious transfer protocol, there is no term in the form of $invf(skgen(r), x)$, so the equivalence cannot be applied. In other to prove this protocol, we need to define the following equivalence:

Equivalence 5: the property of permutation 2

```
equiv !nk new r:seed; (
      ()  -> pkgen(r),
        !nf new x:D; (
        () -> f(pkgen(r), x), () -> x))
      <=(0)=>
      !nk new r:seed; (
      () -> pkgen(r),
        !nf new x:D; (
        () -> x, () -> invf(skgen(r), x))).
```

The Equivalence 5 is derived from Equivalence 2 by exchanging the left-hand side and the right-hand side. For every r and x, the term of $f(pkgen(r), x)$ will be replaced by x and x will be replaced by $inv(skgen(r), x)$.

We continue to run CryptoVerif to prove the simplified protocol fully automatically, but the result is still the same. Even though the script has been executed several times, CryptoVerif still cannot prove the secrecy of i, the generated proof is the same.

Problem 2 : *Priority of equivalences*

The initial game is:

```
Initial state
Game 1 is
in(start, ());
new r: seed;
let pk: pkey = pkgen(r) in
let sk: skey = skgen(r) in
out(cstart, pk);
in(c1, ());
new i: bool;
new x0: D;
new x1: D;
if i then
(
  let y0_15: D = x0 in
  let y1_16: D = f(pk, x1) in
  out(c2, (y0_15, y1_16))
)
else
  let y0_17: D = f(pk, x0) in
  let y1_18: D = x1 in
  out(c2, (y0_17, y1_18))
```

The last game we obtained is:

```
Game 4 is
in(start, ());
new r: seed;
let pk: pkey = @1_pkgen'(r) in
out(cstart, pk);
in(c1, ());
new i: bool;
if i then
(
  new x0_21: D;
  new x1_22: D;
  let y1_16: D = f(pk, x1_22) in
  out(c2, (x0_21, y1_16))
)
else
  new x0_19: D;
  let y0_17: D = f(pk, x0_19) in
  new x1_20: D;
  out(c2, (y0_17, x1_20))
```

From the operations record, we can conclude that CryptoVerif always applies the Equivalence 1 first. After applied Equivalence 1, all the $pkgen(r)$ and $f(pkgen(r), x)$ are replaced by $pkgen'(r)$ and $f(pkgen(r), x)$ respectively. It is the reason why Equivalence 5 cannot be applied.

Manual instructions are needed to indicate in what order the equivalences should be applied. Thus, we use the interactive mode and then control the game transformation by typing commands. The equivalences are applied in the following order:

1. Equivalence 5
2. Equivalence 4

After applied the equivalence 5, we obtained the game 5:

```
Game 5 is
in(start, ());
new r: seed;
out(cstart, pkgen(r));
in(c1, ());
new i: bool;
if i then
(
  new x0_21: D;
  new x1_22: D;
  let y1_16: D = x1_22 in
  out(c2, (x0_21, y1_16))
)
else
  new x0_19: D;
  let y0_17: D = x0_19 in
  new x1_20: D;
  out(c2, (y0_17, x1_20))
```

Actually, the two branches of the if statement are the same: they both select two random variable x_0, x_1 and output them. Thus the if statement can be removed. After the application of Equivalence 1, we can simplify the protocol and obtain the last game:

```
Game 8 is
in(start, ());
new r: seed;
out(cstart, @1_pkgen'(r));
in(c1, ());
new i: bool;
new x0_19: D;
new x1_20: D;
out(c2, (x0_19, x1_20))
```

As we can see from the last game, the output $(x0\_19, x1\_20)$ does not depend on the query bit i. The only one variable depends on i is i itself. Thus, the secrecy of i is proved. The result is:

```
RESULT Proved secrecy of i excluding set(dist 6->7, proba <= 0)
RESULT Proved secrecy of i with probability 0
All queries proved.
```

To generate the proof fully automatically, the following codes could be added to the script:

```
set interactiveMode = true.

proof{
crypto 1 *;
crypto 2 *;
success
}
```

The command *crypto* 1 $*$ means to apply Equivalence 5 to all binders. 1 is the number which is given by CryptoVerif corresponds to Equivalence 5. *crypto* 2 $*$ means to apply Equivalence 1 to all binders.

In fact, the secrecy of i is proved after applied Equivalence 5 and simplified the game. But in this case, the probability of an adversary can successfully get the secret of the receiver can not be estimated. To obtain the probability, there should be at least one probability formula exists in the applied equivalences.

5.3 Security of the sender

To prove security of the sender, we need to prove the secrecy of s_0 and s_1. In this case, we need to describe the whole protocol and use the five equivalences defined before. We also added *query secret i* in the script in order to prove security of both the sender and the receiver in a single script. Both s_0 and s_1 are not represented as local inputs, they are chosen randomly in the script.

First, we try to prove the 1-out-of-2 oblivious transfer protocol fully automatically. The initial game is:

```
Game 1 is
in(start, ());
new r: seed;
let pk: pkey = pkgen(r) in
let sk: skey = skgen(r) in
out(cstart, pk);
(
  in(c0, ());
  new i: bool;
  new x0: D;
  new x1: D;
  if i then
  (
    let y0_21: D = x0 in
    let y1_22: D = f(pk, x1) in
    out(c1, (y0_21, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool));
    let m_25: bool = xor(z1'_23, B(x1))
  )
  else
    let y0_26: D = f(pk, x0) in
    let y1_27: D = x1 in
    out(c1, (y0_26, y1_27));
    in(c2, (z0'_29: bool, z1'_28: bool));
    let m_30: bool = xor(z0'_29, B(x0))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  let z0: bool = xor(B(invf(sk, y0')), s0) in
  let z1: bool = xor(B(invf(sk, y1')), s1) in
  out(c4, (z0, z1))
)
```

The last game we obtained is:

```
in(start, ());
new r: seed;
let pk: pkey = @1_pkgen'(r) in
out(cstart, pk);
(
  in(c0, ());
  new i: bool;
  if i then
  (
    new x0_33: D;
    new x1_34: D;
    let y1_22: D = f(pk, x1_34) in
    out(c1, (x0_33, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool))
  )
  else
    new x0_31: D;
    let y0_26: D = f(pk, x0_31) in
    new x1_32: D;
    out(c1, (y0_26, x1_32));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  new b_35: bool;
  out(c4, (b_36, b_35))
)
```

The result is:

```
RESULT Proved secrecy of s1
RESULT Proved secrecy of s0
RESULT Could not prove secrecy of i.
```

The secrecy of i cannot be proved.

Problem 3 : *the proof strategy*

From the operations record we can know, all the equivalences can be applied successfully and automatically except Equivalence 5. Thus, manual instructions are still needed. We try to apply the equivalences in the following order:

1. Equivalence 5
2. Equivalence 3
3. Equivalence 4
4. Equivalence 1

In this case, Equivalence 5 failed to be applied. Because $sk = skgen(r)$, which is another binder of r, prevents the application of Equivalence 5. In the simplified 1-out-of-2 oblivious transfer protocol, sk has never been used in the processes. Thus, it is removed by simplification. To apply Equivalence 5, there are two available options:

- Eliminate sk before the application of Equivalence 5
- Add sk to left-hand side function groups of Equivalence 5

sk is the secret key of sender, it should be preserved. It must not appear in the process of the receiver. Thus, the optinon 2 is impossible. We should eliminate sk before the application of Equivalence 5. We can apply Equivalence 3 first. Both $xor(B(invf(skgen(r), y0')), s0)$ and $xor(B(invf(skgen(r), y1')), s1)$ will be replaced by $xor(b, s0)$ and $xor(b, s1)$ where b is a random variable of boolean and it will be renamed by CryptoVerif. After applied Equivalence 3, Equivalence 4 and Equivalence 5 can be applied. The order of equivalences application should be:

1. Equivalence 3
2. Equivalence 4
3. Equivalence 5
4. Equivalence 1

The proof is done by applying equivalences in this order. The last game we obtained is:

```
Game 15 is
in(start, ());
new r: seed;
out(cstart, @1_pkgen'(r));
(
  in(c0, ());
  new i: bool;
  new x0_31: D;
  new x1_32: D;
  out(c1, (x0_31, x1_32));
  in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  new b_35: bool;
  out(c4, (b_36, b_35))
)
```

The result is (the proof can be found in Appendix A):

```
RESULT Proved secrecy of i excluding set(dist 14->15, proba <= 0)
RESULT Proved secrecy of i with probability 0
RESULT Proved secrecy of s1 excluding set(dist 14->15, proba <= 0)
RESULT Proved secrecy of s1 with probability 0
RESULT Proved secrecy of s0 excluding set(dist 14->15, proba <= 0)
RESULT Proved secrecy of s0 with probability 0
All queries proved.
```

The output of the sender does not depend on both $s0$ and $s1$, whist the output of the receiver does not depend on i. There is no variable depends on them except themselves. Therefore, the secrecies of $s0$, $s1$ and i are proved.

For fully automatically proof, the following codes should be added to the script:

```
set interactiveMode = true.

proof
{
crypto B *;
crypto 3 *;
crypto xor *;
crypto 4 *;
success
}
```

The command *crypto* $B*$ means to apply the equivalence 3 which is determined by the symbol B to all the binders. The symbol can be a function symbol that only occurs in the left-hand side of an equivalence or a probability function occurs in the probability formula of an equivalence. *crypto* $3*$ and *crypto* $4*$ represent to apply the Equivalence 5 and Equivalence 1 to all binders respectively.

5.4 Summary of equivalences application

Through a series of experiments, some points of equivalences application can be concluded as follows:

- Every random number generation in the left-hand side of an equivalence should be corresponding to the random variable definition in the game
- The terms in the game can be seen as the return values of function groups (oracle groups) in the left-hand side and they will replaced by the corresponding results in the right-hand side

- The function groups (oracle groups) should be well defined. For instance, *!N new a:T;* (FG_1) is the left-hand side of an equivalence, where the function group FG_1 is *new b:T;* (FG_2). The random variable a makes the functions defined in FG_1 available, whist the random variable b makes functions in FG_2 available. In this case, a should appear in FG_1. Both a and b should appear in FG_2. There should be at least a function whose return value contains both a and b defined in FG_2. It provides "clue" to CryptoVerif to locate the random variables a and b

- If there is a random variable defined in the left-hand side, all the binders of that random variable should appear in the equivalence except the binders which will not be used in the game. Otherwise, eliminate those binders which do not appear in the equivalence before applying that equivalence

- CryptoVerif will try to apply the equivalences which are predefined in the macros first. There is no way to modify prioriies of the predefined equivalences. If you cannot obtain the proof because of this reason, try to use interactive mode

6 Conclusion and future work

At present, with the security protocols become more complex, the verification of protocols also becomes more difficult and even undoable. With the development of cryptography, the automated security protocols provers have become available. They provide a new way for the research of security protocols. The two greatest features of automated provers are:

1. The correctness of the proofs only depends on the correctness of the provers themselves

2. If users model the protocols correctly, what is proved secure is indeed secure

CryptoVerif is a computationally sound automated security prover designed by Bruno Blanchet. The aim of this project is to use CryptoVerif to carry out security proof of the 1-out-of-2 oblivious transfer protocol in the semi-honest model. The goal was achieved by the end of the project. It is the first automated proof of the 1-out-of-2 oblivious transfer protocol.

All of the objectives related to the aim of the project are also achieved. In Section 3, we introduced the proof techiniques and the verification procedures of CryptoVerif. In Section 5, we used a "simplified" 1-out-of-2 protocol to verify the security of the receiver first, and then implemented the proof of the whole protocol based on the proof of the "simplified" one. The problems occurred in the experiments are also recorded and analysed.

During the implementation phase, the most challenge lies in the design of equivalences. In CryptoVerif, equivalences are used for specifying properties of cryptographic primitives. More precisely, the application of equivalences determines game transformations. In addition, the calculation of boundary of probability for an adversary can successfully attack a protocol is also based on the applied equivalences. Thus, for using CryptoVerif, understanding of how to design equivalences is important. Some problems occurred in the experiments. By completing a series of experiments, we obtained a lot of generated proofs. From the analysis of these records, some points about equivalences are concluded in Section 5.4.

CryptoVerif is helpful to the research of security protocols. In the future, we may use it to carry out proofs of the other security protocols.

References

[1] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *Journal of Cryptology,* 15(2):103-127, 2002.

[2] M. Bellare and P. Rogaway. Optimal asymmetric encryption - How to encrypt with RSA. In *Eurocrypt '94,* LNCS 950, pages 92-111. Springer -Verlag, Berlin, 1995.

[3] B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy.* pages 140-154, 2006.

[4] B. Blanchet and D. Pointcheval. Automated security proofs with sequences of games. In *Advances in Cryptology - CRYPTO'06, volume 4117 of Lecture Notes in Computer Science,* pages 537-554. Springer-Verlag, 2006.

[5] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society*, 426:233-271, 1989.

[6] S. Even, O. Goldreich and A. Lempel, Ä Randomized protocol for Signing Contracts; *Communications of the ACM 28*, pages 637-647, 1985.

[7] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. In *Advances in Cryptol ogy-Crypto 2001*, 2001.

[8] O. Goldreich. Foundations of Cryptography Volume II: Basic Applications. *Cambridge University Press,* pages 599-643, 2004.

[9] S. Beguëlin, B Grëgoire, S. Heraud, G. Barthe and F. Olmedo. Automating Language-Based Cryptographic Proofs. Available at www-sop.inria.fr/oasis/SAFA/abstracts09/safa09-BGH.pdf.

[10] P. Laud. Handling encryption in an analysis for secure information flow. In *P. Degano, editor, Programming Languages and Systems, 12th European Symposium on Programming, ESOP'03, volume 2618 of Lecture Notes on Computer Science*, pages 159-173, 2003.

[11] P. Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *IEEE Symposium on Security and Privacy,* pages 71-85, Oakland, California, May 2004.

[12] P. Laud. Secrecy types for a simulatable cryptographic library. In *12th ACM Conference on Computer and Communications Security (CCS'05),* pages 26-35, Alexandria, VA, Nov. 2005. ACM.

[13] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM, 21,* 12:993-999, Dec 1978.

[14] M. O. Rabin, How to exchange secrets by oblivious transfer, *Technical Memo TR-81,* Aiken Computation Laboratory, 1981.

[15] P. Paillier and J. Villar. Trading one-wayness against chosen-ciphertext security in factoring-based encryption. In *Xuejia Lai and Kefei Chen, editors, ASIACRYPT 2006, volume 4284 of LNCS*, pages 252-266, 2006.

[16] V. Shoup. OAEP reconsidered. *J. Cryptology*, 15(4):223-249, 2002.

[17] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. *Cryptology ePrint Archive* 2004/332, 2004.

[18] ProVerif. Available at http://www.proverif.ens.fr/.

A Proof of 1-out-of-2 OT protocol

```
==================== Proof starts ======================
Initial state
Game 1 is
in(start, ());
new r: seed;
let pk: pkey = pkgen(r) in
let sk: skey = skgen(r) in
out(cstart, pk);
(
  in(c0, ());
  new i: bool;
  new x0: D;
  new x1: D;
  if i then
  (
    let y0_21: D = x0 in
    let y1_22: D = f(pk, x1) in
    out(c1, (y0_21, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool));
    let m_25: bool = xor(z1'_23, B(x1))
  )
  else
    let y0_26: D = f(pk, x0) in
    let y1_27: D = x1 in
    out(c1, (y0_26, y1_27));
    in(c2, (z0'_29: bool, z1'_28: bool));
    let m_30: bool = xor(z0'_29, B(x0))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  let z0: bool = xor(B(invf(sk, y0')), s0) in
  let z1: bool = xor(B(invf(sk, y1')), s1) in
  out(c4, (z0, z1))
)


Applying simplify yields

Game 2 is
in(start, ());
new r: seed;
let pk: pkey = pkgen(r) in
let sk: skey = skgen(r) in
out(cstart, pk);
(
  in(c0, ());
  new i: bool;
  new x0: D;
  new x1: D;
  if i then
  (
    let y0_21: D = x0 in
    let y1_22: D = f(pk, x1) in
    out(c1, (y0_21, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool))
  )
  else
    let y0_26: D = f(pk, x0) in
    let y1_27: D = x1 in
    out(c1, (y0_26, y1_27));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  let z0: bool = xor(B(invf(sk, y0')), s0) in
  let z1: bool = xor(B(invf(sk, y1')), s1) in
  out(c4, (z0, z1))
)


Applying move all binders yields

Game 3 is
in(start, ());
new r: seed;
let pk: pkey = pkgen(r) in
let sk: skey = skgen(r) in
out(cstart, pk);
(
  in(c0, ());
  new i: bool;
  if i then
  (
    new x0: D;
    let y0_21: D = x0 in
    new x1: D;
    let y1_22: D = f(pk, x1) in
    out(c1, (y0_21, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool))
  )
  else
    new x0: D;
    let y0_26: D = f(pk, x0) in
    new x1: D;
    let y1_27: D = x1 in
    out(c1, (y0_26, y1_27));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  let z0: bool = xor(B(invf(sk, y0')), s0) in
  let z1: bool = xor(B(invf(sk, y1')), s1) in
  out(c4, (z0, z1))
)


Applying remove assignments of useless yields

Game 4 is
in(start, ());
new r: seed;
let pk: pkey = pkgen(r) in
let sk: skey = skgen(r) in
out(cstart, pk);
(
  in(c0, ());
  new i: bool;
  if i then
  (
    new x0_33: D;
    new x1_34: D;
    let y1_22: D = f(pk, x1_34) in
    out(c1, (x0_33, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool))
  )
  else
    new x0_31: D;
    let y0_26: D = f(pk, x0_31) in
    new x1_32: D;
    out(c1, (y0_26, x1_32));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  let z0: bool = xor(B(invf(sk, y0')), s0) in
  let z1: bool = xor(B(invf(sk, y1')), s1) in
  out(c4, (z0, z1))
)


Applying remove assignments of binder sk yields

Game 5 is
in(start, ());
new r: seed;
let pk: pkey = pkgen(r) in
out(cstart, pk);
(
  in(c0, ());
  new i: bool;
  if i then
  (
    new x0_33: D;
    new x1_34: D;
    let y1_22: D = f(pk, x1_34) in
    out(c1, (x0_33, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool))
  )
  else
    new x0_31: D;
    let y0_26: D = f(pk, x0_31) in
    new x1_32: D;
    out(c1, (y0_26, x1_32));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  let z0: bool = xor(B(invf(skgen(r), y0')), s0) in
  let z1: bool = xor(B(invf(skgen(r), y1')), s1) in
  out(c4, (z0, z1))
)


Applying equivalence
equiv ! !_17 <= nk new r: seed; (
  () -> pkgen(r),
  ! !_18 <= nb (x: D) -> B(invf(skgen(r), x)))
<=(0)=>
      ! !_19 <= nk new r: seed; (
  () -> pkgen(r),
  ! !_20 <= nb new b: bool; (x: D) -> b)
with r yields

Game 6 is
in(start, ());
new r: seed;
let pk: pkey = pkgen(r) in
out(cstart, pk);
(
  in(c0, ());
  new i: bool;
  if i then
  (
    new x0_33: D;
    new x1_34: D;
    let y1_22: D = f(pk, x1_34) in
    out(c1, (x0_33, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool))
  )
  else
    new x0_31: D;
    let y0_26: D = f(pk, x0_31) in
    new x1_32: D;
    out(c1, (y0_26, x1_32));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  let z0: bool = xor(b_36, s0) in
  new b_35: bool;
  let z1: bool = xor(b_35, s1) in
  out(c4, (z0, z1))
)


Applying remove assignments of binder pk yields

Game 7 is
in(start, ());
new r: seed;
out(cstart, pkgen(r));
(
  in(c0, ());
  new i: bool;
  if i then
  (
    new x0_33: D;
    new x1_34: D;
    let y1_22: D = f(pkgen(r), x1_34) in
    out(c1, (x0_33, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool))
  )
  else
    new x0_31: D;
    let y0_26: D = f(pkgen(r), x0_31) in
    new x1_32: D;
    out(c1, (y0_26, x1_32));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  let z0: bool = xor(b_36, s0) in
  new b_35: bool;
  let z1: bool = xor(b_35, s1) in
  out(c4, (z0, z1))
)


Applying equivalence
```

```
equiv ! !_11 <= nk new r: seed; (
  () -> pkgen(r),
  ! !_12 <= nf new x: D; (
    () -> f(pkgen(r), x),
    () -> x))
<=(0)=>
      ! !_13 <= nk new r: seed; (
  () -> pkgen(r),
  ! !_14 <= nf new x: D; (
    () -> x,
    () -> invf(skgen(r), x)))
with x0_31 yields

Game 8 is
in(start, ());
new r: seed;
out(cstart, pkgen(r));
(
  in(c0, ());
  new i: bool;
  if i then
  (
    new x0_33: D;
    new x1_34: D;
    let y1_22: D = x1_34 in
    out(c1, (x0_33, y1_22));
    in(c2, (z0'_24: bool, z1'_23: bool))
  )
  else
    new x0_31: D;
    let y0_26: D = x0_31 in
    new x1_32: D;
    out(c1, (y0_26, x1_32));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  let z0: bool = xor(b_36, s0) in
  new b_35: bool;
  let z1: bool = xor(b_35, s1) in
  out(c4, (z0, z1))
)


Applying remove assignments of useless yields

Game 9 is
in(start, ());
new r: seed;
out(cstart, pkgen(r));
```

```
(
  in(c0, ());
  new i: bool;
  if i then
    new x0_33: D;
    new x1_34: D;
    out(c1, (x0_33, x1_34));
    in(c2, (z0'_24: bool, z1'_23: bool))
  else
    new x0_31: D;
    new x1_32: D;
    out(c1, (x0_31, x1_32));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  let z0: bool = xor(b_36, s0) in
  new b_35: bool;
  let z1: bool = xor(b_35, s1) in
  out(c4, (z0, z1))
)


Applying equivalence
equiv ! !_15 <= nx new a: bool; (b: bool) -> xor(a, b)
<=(0)=>
      ! !_16 <= nx new a: bool; (b: bool) -> a
 yields

Game 10 is
in(start, ());
new r: seed;
out(cstart, pkgen(r));
(
  in(c0, ());
  new i: bool;
  if i then
    new x0_33: D;
    new x1_34: D;
    out(c1, (x0_33, x1_34));
    in(c2, (z0'_24: bool, z1'_23: bool))
  else
    new x0_31: D;
    new x1_32: D;
    out(c1, (x0_31, x1_32));
    in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  let z0: bool = xor(b_36, s0) in
  new b_35: bool;
  let z1: bool = b_35 in
  out(c4, (z0, z1))
)


Applying simplify yields

Game 11 is
in(start, ());
new r: seed;
out(cstart, pkgen(r));
(
  in(c0, ());
  new i: bool;
  new x0_31: D;
  new x1_32: D;
  out(c1, (x0_31, x1_32));
  in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  let z0: bool = xor(b_36, s0) in
  new b_35: bool;
  let z1: bool = b_35 in
  out(c4, (z0, z1))
)


Applying remove assignments of useless yields

Game 12 is
in(start, ());
new r: seed;
out(cstart, pkgen(r));
(
  in(c0, ());
  new i: bool;
  new x0_31: D;
  new x1_32: D;
  out(c1, (x0_31, x1_32));
  in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  let z0: bool = xor(b_36, s0) in
  new b_35: bool;
  out(c4, (z0, b_35))
)


Applying equivalence
equiv ! !_15 <= nx new a: bool; (b: bool) -> xor(a, b)
<=(0)=>
      ! !_16 <= nx new a: bool; (b: bool) -> a
 yields

Game 13 is
in(start, ());
new r: seed;
out(cstart, pkgen(r));
(
  in(c0, ());
  new i: bool;
  new x0_31: D;
  new x1_32: D;
  out(c1, (x0_31, x1_32));
  in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  let z0: bool = b_36 in
  new b_35: bool;
  out(c4, (z0, b_35))
)


Applying remove assignments of useless yields

Game 14 is
in(start, ());
new r: seed;
out(cstart, pkgen(r));
(
  in(c0, ());
  new i: bool;
  new x0_31: D;
  new x1_32: D;
  out(c1, (x0_31, x1_32));
  in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  new b_35: bool;
  out(c4, (b_36, b_35))
)


Applying equivalence
equiv ! !_5 <= @1_nK new @1_r: seed; (
  @1_Opk() := pkgen(@1_r),
  ! !_6 <= @1_nf new @1_x: D; (
    @1_Oy() := f(pkgen(@1_r), @1_x),
    ! !_7 <= @1_n2 @1_Oeq(@1_x': D) := (@1_x' = @1_x),
    @1_Ox() := @1_x))
<=(set(proba <= @1_nK * @1_nf * POW(time + (@1_nK - 1.)
   * time(pkgen) + (#@1_Oy - 1.) * time(f))))=>
      ! !_8 <= @1_nK new @1_r: seed; (
  @1_Opk() := @1_pkgen'(@1_r),
  ! !_9 <= @1_nf new @1_x: D; (
    @1_Oy() := @1_f'(@1_pkgen'(@1_r), @1_x),
    ! !_10 <= @1_n2 @1_Oeq(@1_x': D) := if defined(@1_k)
    then (@1_x' = @1_x) else false,
    @1_Ox() := let @1_k: bitstring = @1_mark in @1_x))
 [Excluding set(dist 14->15, proba <= 0)] yields

Game 15 is
in(start, ());
new r: seed;
out(cstart, @1_pkgen'(r));
(
  in(c0, ());
  new i: bool;
  new x0_31: D;
  new x1_32: D;
  out(c1, (x0_31, x1_32));
  in(c2, (z0'_29: bool, z1'_28: bool))
) | (
  in(c3, (y0': D, y1': D));
  new s0: bool;
  new s1: bool;
  new b_36: bool;
  new b_35: bool;
  out(c4, (b_36, b_35))
)


RESULT Proved secrecy of i excluding set(dist 14->15, proba <= 0)
RESULT Proved secrecy of i with probability 0
RESULT Proved secrecy of s1 excluding set(dist 14->15, proba <= 0)
RESULT Proved secrecy of s1 with probability 0
RESULT Proved secrecy of s0 excluding set(dist 14->15, proba <= 0)
RESULT Proved secrecy of s0 with probability 0
All queries proved.
```