CryptoVerif - the tool of crypto analysis

Ivo Seeba

The University of Tartu, The Department of Computer Science, EST

Abstract This paper gives a brief information about cryptographic protocol analyzing tool CryptoVerif. It gives information about basic terms and definitions, what is inside of CryptoVerif, and there are some examples of usage.

1 Introduction

The CryptoVerif can output proofs to files (in ordinary and in latex-format) about given protocols.

In the paper we try to describe the tool CryptoVerif that helps cryptographers to analyze many different cryptosystems. It outputs proofs to a command line, and to files (in LaTeX and in a common format). There are models (computational and formal i.e. Dolev-Yao) for security protocols, and CryptoVerif uses a computational model. The automatic proof has two approaches - indirect (uses Dolev-Yao proofs) and direct (designing automatic tools for proving protocols in a computational model) approach. The CryptoVerif uses computational model and CryptoVerif uses the direct model. The program proves secrecy and correspondence properties, provides a generic method for specifying properties of crypto primitives, works for unbounded number sessions with an active adversary, and gives a bound on the probability of an attack.

The basic idea of CryptoVerif is same as in papers [4] and [5]. It relies on a collection of game transformations, in order to transform the initial protocol into a game on which the desired security property is obvious. The proof is a sequence of games as follows. The first game is the real protocol, one goes from one game to the next by syntactic transformations or by applying the definition of security of a cryptographic primitive. The last game is an ideal one where the security property is obvious from the game’s shape. Games are formalized in a process calculus similar to π-calculus [1]. The calculus is purely probabilistic and runs in polynomial time.

In Sec. 2 there is information about other tools, in Sec. 3 about usage of CryptoVerif and Sec. 4 the internals of CryptoVerif. In Sec. 5 there are two examples where the first is taken from [8] and the second is about analyzing an ElGamal cryptosystem. The paper is written during a course [2] and so it is written as a beginner.
2 A little about other cryptoanalysing tools

CryptoTool [11] is a free e-learning graphical application for Windows. It enables to apply and analyze cryptographic mechanisms: numerous classic and modern cryptographic algorithms, visualization of several methods, analysis of certain algorithms, measuring and auxiliary methods, number theory’s tutorial etc. It contains exhaustive online help, and is available in English, German, Polish and Spanish. It’s a good tutorial program for beginner cryptographers.

Blanchet has made also another tool ProVerif [10]. It is similar to CryptoVerif, but the ProVerif analyses protocols in Dolev-Yao model and it is more natural and easier to use than CryptoVerif. It enables static analysis for cryptoprotocols under the perfect crypto assumptions, checks secrecy and correspondence properties, it says if protocols aren’t secure, translates protocols to sets of Horn clauses, etc. Similarities of the two tools are specification languages of protocols, being variants of the π-calculus.

Also a cryptographic analyzing tool, dependency graphs based Cryptoanalysyser has been made and its graphical and fast version is in the plan to do.

3 Using CryptoVerif

The syntax of the command line is “cryptoverif [options] ⟨filename⟩”, where options are -lib ⟨filename.(o)cvl⟩ (e.g. -lib myLib.cvl), -in ⟨frontend⟩ (oracles or channels) and -tex ⟨filename⟩ (e.g. -tex myProof.tex). Also text-based proof files can be produced e.g. “cryptoverif ddh.cv ⟩ ddh.proof”. The manual [3] consists of an introduction, command line instructions, channels- and oracle Front-end’s descriptions, a summary of between Front-ends and descriptions of predefined cryptographic primitives that can be found from file default.cvl and system’s output. The primitives are for example IND_CPA_sym_enc, PRP_cipher, IND_CPA_INT_CTX_sym_enc, etc. The file can be found from directory which is found from the archive [3].

4 Inside CryptoVerif

Games are communications between an adversary and environment. These depict pseudocodes for formalizing cryptosystems. Cryptographic proof are presented as a sequence of games where the first game (e.g. \(G_1\)) is a initial protocol that represent a real system and the last game (e.g. \(G_n\)) is an ideal protocol that security is obvious. Briefly the proof is shown \(G_1 \approx G_2 \approx \ldots \approx G_n\).

The process calculus (the syntax below) consists in countable number of channels, where all channel \(c\) the maximum length of message \(maxlen_k(c)\) sent along the channel \(c\) is polynomial in \(k\). The calculus assumes also others things: a set of parameters, types and functions ([5], Seq. 3).

The calculus has three main parts: terms, input- and output processes. All parts have specified grammars.
To make the presentation easy to understand, we start from a simple example and gradually introduce all basic concepts and techniques used in CryptoVerif. It is well known that double application of one-way function is also a one-way function. To prove this fact formally, we have to first formalize a game that measures security of a one-way function. The corresponding game in the CryptoVerif output (in latex format) notation is:

Game 1 is

\begin{verbatim}
start();
new x : D;
adv(g(x));
adv(x' : D);
if (g(x) = g(x')) then event bad
\end{verbatim}

In the games start(); is input channel start that executes the game, new x : D; is x \in \llbracket D \rrbracket (\llbracket D \rrbracket is interpretation of type D i.e. the set of all values that have the type), adv(g(x)); means the value of g(x) is output to the adversary, adv(x' : D); means the value from the adversary via channel adv() and the value is assigned to variable x' (with type D) from the adversary, if (g(x) = g(x')) then event bad means that check the equality between g(x) and g(x') holds. The function g is one-way function but the event bad has a negligible probability.

More formally, any security game must follow the following syntax

- Terms: M, N ::= i [x[M_1, M_2, \ldots, M_m]] f(M_1, M_2, \ldots, M_m), where
  - M and N are terms. The terms represent computations on bitstrings.
  - i is a replication index. It serves in distinguishing different copies of a replicated process \( i \leq m \).
  - [x[M_1, M_2, \ldots, M_m]] is a variable access. It returns the content of the cell of indexes \( M_1, M_2, \ldots, M_m \) of the \( m \)-dimensional array variable \( x \).
  - \( f(M_1, M_2, \ldots, M_m) \) is a application of function \( f \). It takes parameters \( M_1, M_2, \ldots, M_m \) (with correct types \( T_1, T_2, \ldots, T_m \) ) and results value of type \( T \).

- Input processes that uses grammar \( Q ::= 0 \mid Q|Q' \mid !^\leq Q \mid newChannel c; Q \mid c[M_1, M_2, \ldots, M_l]; x_1[i] : T_1, x_2[i] : T_2, \ldots, x_r[i] : T_r); P \) are ready to receive message on a channel. The input processes mean:
  - 0 is nil. The process 0 is discarded if parallel composed with some other process. Else if the current process is 0 without a parallel composed process it stops its executions.
  - Q\( |Q' \) - parallel composition of processes Q and Q'.
  - \( !^\leq Q \) replicating \( n \) times and represent \( n \) copies \( (Q_1 | Q_2 | \ldots | Q_n) \) with different values of Q in parallel.
  - newChannel c; Q channel restriction. The process creates a new private channel c and executes Q. It is reduced to Q, where each occurrence of c in Q is replaced by a fresh channel name \( c' \).
• If the input process \( c[M_1, M_2, \ldots, M_r][x_1] : T_1, x_2[i] : T_2, \ldots, x_r[i] : T_r; P \) and the first output process (i.e. \( c[M_1, M_2, \ldots, M_r](N_1, N_2, \ldots, N_k); Q \)) are composed in parallel, and the terms \( M_1, M_2, \ldots, M_r \) evaluate to values \( b_1[i] \in I(T_1), b_2[i] \in I(T_2), \ldots, b_r[i] \in I(T_r) \) then the input process reduced to \( P \) (and output process reduced to \( Q \)). Variables \( x_1[i], x_2[i], \ldots, x_r[i] \) in the input process are assigned to the values \( b_1[i], b_2[i], \ldots, b_r[i] \). If there are several process \( c[M_1, M_2, \ldots, M_r][x_1[i] : T_1, x_2[i] : T_2, \ldots, x_r[i] : T_r; P \) satisfying these conditions, one is chosen uniformly at random and the syntactical conditions on processes ensure that there is at most one processes of the form \( c[M_1, M_2, \ldots, M_r](N_1, N_2, \ldots, N_k); Q \). Executing the output process it looks an input on the same channel and with the same arity in the available input process. Finding no such input process, the output process blocks, finding such input processes the output processes chooses one of input processes uniformly. If the communication is executed, the messages \( N_1, N_2, \ldots, N_k \) are evaluated (results are truncated to maximum length of messages in channel \( c \)), the obtained bitstrings are stored \( x_1[i], x_2[i], \ldots, x_r[i] \) if these have suitable types otherwise the process blocks.

Finally, the process \( P \) following the input is executed and the process \( Q \) following the output is stored in the available input processes for future execution.

- Output processes \( P \) using a grammar \( P ::= c[M_1, M_2, \ldots, M_r](N_1, N_2, \ldots, N_k); Q \mid \text{new } \langle x[i_1, i_2, \ldots, i_m] : T; P \rangle \mid \text{let } \langle x[i_1, i_2, \ldots, i_m] : T = M \text{ in } P \rangle \mid \text{if } M \text{ then } P \text{ else } P' \mid \text{find A suachthat defined } (M_{1j}, M_{2j}, \ldots, M_{rj}) \land M_j \text{ then } P \text{ else } P' \) output a message on a channel after executing some internal computations. The output processes mean:

  - The explanation of an output process \( c[M_1, M_2, \ldots, M_r](N_1, N_2, \ldots, N_k); Q \) is in the last input process. \( Q \) is an input process.
  - \( \text{new } \langle x[i_1, i_2, \ldots, i_m] : T; P \rangle \) means a random number. The random number is assigned to \( x \) and then the process \( P \) is executed.
  - \( \text{let } \langle x[i_1, i_2, \ldots, i_m] : T = M \text{ in } P \rangle \) means assignment. The value of \( M \) is assigned to \( x[i_1, i_2, \ldots, i_m] \) and then the process \( P \) is executed. The variable \( x[i_1, i_2, \ldots, i_m] \) must be exist in process \( P \) otherwise the statement is pointless.
  - If \( M \) then \( P \) else \( P' \) is a conditional statement, that means if the term \( M \) is true, the process \( P \) is executed and if the term is false, \( P' \) is executed.
  - \( \text{find } \oplus_{j=1}^{m} \langle u_j[i_1, i_2, \ldots, i_m] \rangle \leq n_{j_1}, u_2[i_1, i_2, \ldots, i_m] \leq n_{j_2}, \ldots, u_{jm}[i_1, i_2, \ldots, i_m] \leq n_{jm}, \text{ suchthat defined } (M_{j_1}, \ldots, M_{j_r}) \land M_j \text{ then } P_j \rangle \text{ else } P u_1, u_2, \ldots, u_m, \text{ where } u_j[i_1, i_2, \ldots, i_m] \) is a subarray of array \( u_j \) consisting of only element at indexes \( i_1, i_2, \ldots, i_m \).

The explanation begins in an simple example: \( \text{find } u \leq n \text{ suchthat defined}(x[u]) \land x[u] = a \text{ then } P' \text{ else } P \) tries to find an index \( u \) such that \( x[u] \) and \( x[u] = a \), finding such \( u \) it executes process \( P' \) otherwise it executes \( P \).
There is a little complex example: find $u_1[i_1, i_2, \ldots, i_{m_1}] \leq n_1, \ldots, u_m[i_1, i_2, \ldots, i_{m_1}] \leq n_m$ such that defined $(M_1, \ldots, M_t) \land M \rightarrow P')$ else $P$

$u_{j_1}, u_{j_2}, \ldots, u_{j_{m_1}}$ tries to find values of $u_1, \ldots, u_m$ for which $M_1, \ldots, M_t$ are defined and $M$ is true and it executes $P'$ in a success case and $P$ in failure case.

And more complex example - common output process above, generalized to $m$ branches tries to find a branch $j \in \{1, 2, \ldots, m\}$ such that there are values of $u_1[i_1, i_2, \ldots, i_{m_1}], \ldots, u_m[i_1, i_2, \ldots, i_{m_1}] \in \{(1, 2, \ldots, n_1) \times \ldots \times (1, 2, \ldots, n_{m_1})\}$ for $M_{j_1}, M_{j_2}, \ldots, M_{j_t}$ and executes $P$ if no conditions hold. Otherwise it executes $P_j$ choosing one of success found condition uniformly.

Observational equivalent means intuitively an adversary is unable to distinguish between processes $Q_L$ and $Q_R$ i.e. the two process are similar for the view of the adversary that is represented as a context.

Formally processes $Q_L$ and $Q_R$ are called to be observational equivalent (written by $Q_L \equiv_v Q_R$) with public variables $V$, when all evaluation contexts $C$ acceptable ([5], p.7) for $Q_L, Q_R, V$, for all channels $c$ and bitstrings $a, |Pr[c|Q_L, a \sim_k \exists a] - Pr[c|Q_R, a \sim_k \exists a]|$ is negligible in $k$. If $V = \emptyset$, then we can write $Q_L \equiv_v Q_R$ instead of $Q_L \equiv_v Q_R$.

The context is a term with a special symbol $\square$ (hole). Given a context $C$ and a term $M, C[M]$ denotes the result of replacing every occurrence of $\square$ in $C$ by $M$. The context uses a grammar $C ::= \square | \text{newChannel} c; C | (C)Q | (Q)C$. We also denote $C[Q]$ the process obtained by replacing $\square$ in the context $C$ with the process $Q$.

Transformations are environment changing in a way that the adversary doesn’t notice changing from certain game to a right neighbour game is performed is called transformation. There are two kind of transformations - syntactic and applying the definition of security of a cryptographic primitive. Describing transformations we need several processes while the transformation is too long that makes difficult to understand and these have to be divide small ones.

The crypto primitives’ security is defined with observational equivalences given as axioms. The primitives are specified using equivalences $(G_1, \ldots, G_m) \approx (G'_1, \ldots, G'_m)$ where $G_1, \ldots, G_m, G'_1, \ldots, G'_m$ has a grammar $G ::= \text{new} y_1 : T_1; \ldots, \text{new} y_m : T_m (G_1, \ldots, G_m) \{ (x_1 : T_1, \ldots, x_t : T_t) \rightarrow FP \}$ (the translation of the first statement inputs and outputs on channel $c$ so that the context can trigger the generation of random numbers $y_1, \ldots, y_m$, and the translation of the second statement inputs the arguments of the function on channel $c$ and translates $FP$ that outputs its result on channel $c$) and $FP$ follows grammar $FP ::= M \{ \text{new} x[i_1, \ldots, i_{m_1}] : T = M \text{in } FP | \text{if } M \text{ then } FP_1 \text{ else } FP_2 \}$ find $\oplus_{j=1}^m (u_1[i_1, i_2, \ldots, i_{m_1}] \leq n_{j_1}, u_2[i_1, i_2, \ldots, i_{m_1}] \leq n_{j_2}, \ldots, u_{j_m}[i_1, i_2, \ldots, i_{m_1}] \leq n_{j_{m_j}})$ such that defined $(M_{j_1}, \ldots, M_{j_{m_j}}) \land M_j \rightarrow FP_j)$ else $FP$.

We denote with $[FP]$ the translations of the functional process $FP$. The translation of $(x_1 : T_1, \ldots, x_t : T_t) \rightarrow FP$ inputs the arguments of the function on
channel $c$ and translates $FP$, which outputs the result of $FP$ on $c$. For example
let $y = E(x_1, \ldots, x_m)$ then $[E] = (c_E(x_1, \ldots, x_m), \overline{c}_E(y))$; the second example is a
one-way function (Sec. 4.1):
$Q_L = \text{setprove}(\text{neg}()) \ Q_R$
where
$Q_L = (\ell^n \text{ new } x : D; ( () \rightarrow f(x), (x' : D) \rightarrow (x = x')))$ and
$Q_R = (\ell^n \text{ new } x : D; ( () \rightarrow f(x), (x' : D) \rightarrow \text{ false}))$
where $(x' : D) \rightarrow x = x'$ is $(c(x') : b := (x = x'); \overline{c}(b));$, $(x_m : D) \rightarrow \text{ false}
means (c(x'); \overline{c}(\text{false});)$ and $(()) \rightarrow f(x)$ means $(c(); \overline{c}(f(x)));$. $!\ell^n \text{ new } x : D; (...)$ is
replication $\ell$ times performing $x \leftarrow l(D) : (...)$. 

Formally, functions can be encoded as processes that input their arguments
and output their result on a channel. The semantics of the security games (pro-
cesses) is defined as follows [7]:

- $[(G_1, \ldots, G_m)] := [G_1] \ldots [G_m]$ -
- $[(x_1 : T_1, x_2 : T_2, \ldots, x_\ell : T_\ell) \rightarrow FP] := (x_1 : T_1, x_2 : T_2, \ldots, x_\ell : T_\ell); [FP] - process $FP$
getting values of $\ell$ variables can be represent as process using an
input channel $c$ for getting these.
- $!\ell^n \text{ new } y_1 : T_1, \text{ new } y_2 : T_2, \ldots, \text{ new } y_\ell : T_\ell; (G_1, G_2, \ldots, G_m) :=
!\ell^n (c(); \text{ new } y_1 : T_1, \text{ new } y_2 : T_2, \ldots, \text{ new } y_\ell : T_\ell; \overline{c}(); ([(G_1, G_2, \ldots, G_m)] -
- $[[M]] := \overline{c}(M)$ the term can be process outputing its value along channel $c$.
- $[\text{ new } x : T; FP] := \text{ new } x : T; [FP] - [FP]$ if variable $x \leftarrow l(T)$ in the game $FP$ that
the $TP$ can be a distinct process getting the random value along its channel.
- $[\text{ let } x : T = M \text{ in } FP] := \text{ let } x : T = M \text{ in } [FP] - [FP]$ the process $FP$ can be distinct
getting the value $M$ along an input channel and replacing all occurrences $x
with it.
- $[\text{ if } M \text{ then } FP_1 \text{ else } FP_2] := [\text{ if } M \text{ then } [FP_1] \text{ else } [FP_2] - processes $FP_1$ and
$FP_2$ are distinct. If term $M$ is true, that $FP_1$ executes, else $FP_2$ executes.

The validity control of transformations is performed CryptoVerif and user
work is to write suitable protocol specifications.

5 Examples

The specific programming language in which inputs are written, and about
which no programming tutorial could not be found, has some similarity with
language Ocaml. Comments are in form (* comments \ldots *). Types, channels,
events, functions, probabilities etc., must be declared.

There are two examples. The first taken from [8]. How can CryptoVerif proof
automatically the fact if $f$ is a one-way permutation, then $f \circ f$ is a one-way
function.

5.1 Example 1

In order to specify the security game for one-way function [8] we have to
specify the players. in our case, we have to define adversary and challenger.
The latter means that we have to specify two communication channels: \( \text{adv} \) for communicating with adversary, \( \text{start} \) for starting the game and \( \text{bad} \) for a bad-event that is intuitively false. The challenger does not a priori get any dedicated communication channels, since in most cases the challenger’s behaviour can be specified with a few lines of code in the process description. The corresponding declaration in the CryptoVerif’s input file is

\[
\text{channel start, adv.}
\]
\[
\text{event bad.}
\]
\[
\text{query event bad} \Rightarrow \text{false.}
\]

Next we have to define one-way function \( f \) and its double application \( g = f \circ f \). For that we need to define input domain \( D \). Input and output domains of the functions are defined as follows

\[
\text{type } D \; \text{[fixed]}. \\
\text{fun } f(D):D. \\
\text{fun } g(D):D. \\
\text{forall } x:D; \quad g(x) = f(f(x)). \\
\text{forall } x:D, \; x':D; \quad (f(x)=f(x')) = (x=x').
\]

where \( f(D) \) means that input domain of \( f \) is \( D \) and \( :D \) following \( f(D) \) means that output domain is \( D \). \([\text{fixed}]\) means that it is possible to uniformly pick random values of this type. \([\text{fixed}]\) statement means that a certain function can be applied for all values given a certain type.

\[
\text{forall } x:D, \; x':D; \quad (f(x)=f(x')) = (x=x')
\]

means that the Injectivity of the function \( f \) is needed for proving the event \( \text{bad} \Rightarrow \text{false} \). For one-wayness of the function \( f \) we need also probability \( \text{negl} \) and a parameter \( n \) for replication \( !n \):

\[
\text{proba negl.} \\
\text{param } n.
\]

We need \( !n \text{new } x: D; ((\rightarrow f(x), x': D \rightarrow x = x') \approx_{\text{set proba negl}} !n \text{new } x: D; ((\rightarrow f(x), x': D \rightarrow \text{false}) \) for denoting

\[
|\Pr[x, x' \overset{\leftarrow}{\leftarrow} D : x = x'] - \Pr[x, x' \overset{\leftarrow}{\leftarrow} D : \text{false}]| \leq \text{negl}:
\]

\[
\text{equiv} \\
!n \text{new } x: D; ((\rightarrow f(x), (x':D) \rightarrow x=x') \\
\leq (\text{negl})=\} \\
!n \text{new } x: D; ((\rightarrow f(x), (x':D) \rightarrow \text{false}).
\]

And finally we can define the game that is represented as a process:
where the description of the game is given in Sec 3.

Here is the proof by CryptoVerif that represents a sequence of games (the games is modified a little with goal to make better readable):

<table>
<thead>
<tr>
<th>$G_1$</th>
<th>$G_2$</th>
<th>$G_3$</th>
<th>$G_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>start();</td>
<td>start();</td>
<td>start();</td>
<td>start();</td>
</tr>
<tr>
<td><strong>new</strong> $x : D$;</td>
<td><strong>new</strong> $x : D$;</td>
<td><strong>new</strong> $x : D$;</td>
<td><strong>new</strong> $x : D$;</td>
</tr>
<tr>
<td>$ADV(g(x))$;</td>
<td>$ADV(f(f(x)))$;</td>
<td>$ADV(f(f(x)))$;</td>
<td>$ADV(f(f(x)))$;</td>
</tr>
<tr>
<td>$ADV(x : D)$;</td>
<td>$ADV(x : D)$;</td>
<td>$ADV(x : D)$;</td>
<td>$ADV(x : D)$;</td>
</tr>
<tr>
<td>if ($g(x) = g(x')$) then event bad</td>
<td>if ($x = x'$) then event bad</td>
<td>if false then event bad</td>
<td></td>
</tr>
</tbody>
</table>

There is description of transformations, where $G_1 \implies G_2$ and $G_3 \implies G_4$ are syntactical transformations and $G_2 \implies G_3$ applies the equivalence:

- $G_1 \implies G_2$: Applying simplify yields. The result of the transformations is using definition inside of function $g$ and because of injectivity of $f$ it uses $x = x'$ instead of $g(x) = g(x')$. In the game $G_1$ we can apply the simplification that replace function $g$ with $f \circ f$ and the equality of $g(x)$ and $g(x')$ with an equality of $x$ and $x'$ because in assumption of injectivity of $f$ ($x = x' \implies f(x) = f(x') \implies f(f(x)) = f(f(x'))$). The transformations can by specified in a input file e.g. the injectivity can by specified as $\text{forall } x : D, x' : D; (f(x) = f(x')) = (x = x')$. The description of the code of $G_1$ is given in Sec 3.

- $G_2 \implies G_3$: Applying equivalence:

\[
\forall^* \text{new } x : D; (\_ \rightarrow f(x), (x' : D) \rightarrow (x = x')) \approx_{\text{set(proba}} \leq \text{negl)}
\]

[Excluding set(dist2 > 3, proba \leq \text{negl})] yields the observational equivalence $G_2 \approx G_3$ is applied that $|\Pr[G_2 \Rightarrow 1] - \Pr[G_3 \Rightarrow 1]| \leq \text{negl}$ (where returning one means bad-setting) because $\Pr[G_2 \Rightarrow 1] \leq \text{negl}$ and $\Pr[G_3 \Rightarrow 1] = 0$. The result of transformation is usage of $\text{false}$ instead of $x = x'$.

If the game $G_2 = Q \parallel [C]$ and $G_3 = Q \parallel [C']$, then

\[
Q = (\_; c_1(); c_1(y); \overline{ADV}(y)); ADV(x : D));
\]

$[C] = (\forall^* \text{new } x : D; (c_1(); c_1(f(x))); c_2(x' : D); c_2(x = x'))$ and

$[C'] = (\forall^* \text{new } x : D; (c_1(); c_1(f(x))); c_2(x' : D); c_2(false));$

In observational equivalences $G_2 \approx C_2[L_2] \approx C_3[R_2] \approx G_3$ where $C_2 = (\text{newchannel} c(), \overline{c}(); (\_; \_; \_; \_; \_; \_); (\_; \_; \_; \_; \_; \_); (\_; \_; \_; \_; \_; \_); (\_; \_; \_; \_; \_; \_))$. 
\( \mathbb{T}(b); \text{if } b \text{ then event bad }[\square]) \),
\[ L_2 = (c(z, z'); r := (g(z) = g(z')); \mathbb{T}(r); ) \]
\[ R_2 = (c(z, z'); r := (z = z'); \mathbb{T}(r)); ) \]

- \( G_3 \Rightarrow G_4 \): Applying simplify yields, where the useless code is removed, because \( \Pr[\text{false}] = 0 \) and bad-setting is impossible. The result of the transformation is previous game without if-statement.

If the game \( G_3 = Q [C] \) and \( G_4 = Q [C'] \), then

The game \( G_4 \) is a final game. It is secure because the adversary cannot get more information excluding the value of \( f(f(x)), x \xleftarrow{} I(D) \). And CryptoVerif summarizes the simple proof:

- Proved event bad \( \Rightarrow \text{false} \) excluding set(dist2 > 3, proba \( \leq \text{negl} \))
- Proved event bad \( \Rightarrow \text{false} \) with probability negl

All queries proved.

Here set(dist2 > 3, proba \( \leq \text{negl} \)) designates the set of traces that allow the adversary to distinguish \( G_2 \) from \( G_3 \), with the additional information that the probability of these traces is at most \( \text{negl} \).

### 5.2 Example 2, the ElGamal encryption system

There is an another example. Let \( G \) be a group (it is also a message space) of prime order \( q \), and let \( \gamma \in G \) be a generator (i.e. \( G = \langle \gamma \rangle \)). There is a sketch of the cryptosystem, where Gen is a key generation, Enc is a encryption and Dec is a decryption from [6] Sec 3.2:

\[
\begin{align*}
\text{Gen} & \quad | \quad x \xleftarrow{} \mathbb{Z}_q \\
& \quad | \quad \alpha \xleftarrow{} \gamma^x \\
& \quad | \quad pk \leftarrow \alpha \\
& \quad | \quad sk \leftarrow x \\
& \quad | \quad \text{return } (pk, sk)
\end{align*}
\]

\[
\begin{align*}
\text{Enc}_{pk}(m) & \quad | \quad \alpha \xleftarrow{} pk \\
& \quad | \quad y \xleftarrow{} \mathbb{Z}_q \\
& \quad | \quad \beta \xleftarrow{} \gamma^y \\
& \quad | \quad \delta \xleftarrow{} \alpha^y \\
& \quad | \quad \zeta \xleftarrow{} \delta \cdot m \\
& \quad | \quad \text{return } c
\end{align*}
\]

\[
\begin{align*}
\text{Dec}_{sk}(c) & \quad | \quad x \leftarrow sk \\
& \quad | \quad m \leftarrow \zeta / \beta^x \\
& \quad | \quad \text{return } m \\
& \quad | \quad c \xleftarrow{} (\beta, \zeta) \\
& \quad | \quad \text{return } m
\end{align*}
\]

The specification file \( (\text{Avik-lgamal.CV}) \) is got from B. Blanchet along e-mail and it cannot be found from Internet. In the file the encryption system is shown to be SS under the DDH assumption ([6], Sec 3). The proof by CryptoVerif consists of seven game where the first is the initial game and 7th is the final game. The proof is modified to be hopefully more readable.
The lines in the first games accordingly game title \((G_1)\), replication, \(x \leftarrow 7 \mathbb{Z}_q\), \(\alpha \rightarrow \delta^y\), \(\delta^y\) a sending via channel \(cPK\), \(m_0, m_1\) receiving via channel \(cE, m \leftarrow \{m_0, m_1\}\), \(\beta \rightarrow \delta^y, \delta \rightarrow \alpha^y, \zeta \rightarrow \delta \cdot m\) and sending \(\beta, \zeta\) via output channel \(cE.\) \(\gamma\) means a replication index and the code runs \(q\) times. Transformations in the sequence of games:

- \(G_1 \implies G_2\): Applying simplification. It is syntactic transformation displacing \(\gamma\) with \(y\) and \(\gamma_{\text{mult}}(x, y)\). Result is \(\delta^y\) represented as a power of \(\gamma\). It simplifies \(\delta = \alpha^y \Rightarrow \delta = \gamma^y\) using the definitions written into the input file: \(\text{equiv}(y) \rightarrow \text{newx: Zq; exp(gamma, x)[all] <= (0) \rightarrow (n-> newy: G; y. \text{unless}[\forall x : G, x : Zq, y : Zq, \text{exp}(a, x, y) = \text{exp}(a, \text{mult}(x, y))].

- \(G_2 \implies G_3\): Applying equivalence in DDH-assumption

\[
!\text{new x: Zq; new y: Zq; } (\gamma) \rightarrow \text{exp}(y, x), (\gamma) \rightarrow \text{exp}(y, x), (\gamma) \rightarrow \text{exp}(y, \text{mult}(x, y)) \]

\[
!\text{new x: Zq; new y: Zq; new z: Zq; } (\gamma) \rightarrow \text{exp}(y, x), (\gamma) \rightarrow \text{exp}(y, x), (\gamma) \rightarrow \text{exp}(y, z)\]

[Excluding \(\text{set}(\text{dist2} \rightarrow 3, \text{proba} \leq p\text{DDH}(\text{time + time(context for game 2)} \times q))\) yields. I.e. the transformation uses the observational equivalence. The adversary can distinguish \(G_3\) from \(G_3\), with the additional information that the probability of these traces is at most \(p\text{DDH}(\text{time + time(context for game 2)) \times q})\) where \(\text{time}\) is the runtime of the adversary attacking the protocol, \(\text{time(context for game 2)}\) is the difference between the runtime of the adversary attacking the protocol and the adversary attacking DDH and \(q = |\mathbb{Z}_q|\).

- \(G_3 \implies G_4\): Applying equivalence

\[
!\text{new x: Zq; } (\gamma) \rightarrow \text{exp}(y, x)[al\ll] \approx_0 !\text{new y: G; } (\gamma) \rightarrow y \] yields. This is based on cryptographic transformation. The result is \(y_{12}\) instead on \(\gamma^y\), \(y_{14}\) instead on \(\gamma^y\), and \(y_{16}\) instead of \(\gamma^y\).
In the conclusion the CryptoVerif is a tool that can be used for proofs. For drawback no good tutorials for proving protocols weren’t found, e.g. reading
several slides from [9] about theories, but no info about writing corresponding input files (except the item with files enc-then-MAC.cv and fdh.cv). It made some difficulties. The CryptoVerif helps to prove many protocols like FDH, signed Diffie Hellman, etc.

7 Acknowledgements

The author of this work would like to thank Peeter Laud, Sven Laur and Bruno Blanchet for their comments and for helpful discussions on the topic without whose help the paper would have been totally unreadable.

References

10. B. Blanchet ProVerif http://www.proverif.ens.fr/