Mechanizing Game-Based Proofs of Security Protocols: The Tool CryptoVerif

Bruno Blanchet

INRIA Paris-Rocquencourt
Bruno.Blanchet@inria.fr

Year 2014-15
Outline

1. Using CryptoVerif
2. Proof technique: game transformations, proof strategy
3. Two examples:
   - Encrypt-then-MAC
   - FDH
4. Conclusion, future directions
CryptoVerif is an automatic prover that:

- generates proofs by sequences of games.
- proves secrecy and correspondence properties.
- provides a generic method for specifying properties of cryptographic primitives which handles MACs (message authentication codes), symmetric encryption, public-key encryption, signatures, hash functions, Diffie-Hellman key agreements, …
- works for $N$ sessions (polynomial in the security parameter), with an active adversary.
- gives a bound on the probability of an attack (exact security).
Input and output of the tool

1. Prepare the input file containing
   - the specification of the **protocol** to study (initial game),
   - the **security assumptions** on the cryptographic primitives,
   - the **security properties** to prove.

2. Run CryptoVerif

3. CryptoVerif outputs
   - the **sequence of games** that leads to the proof,
   - a **succinct explanation** of the transformations performed between games,
   - an upper bound of the **probability** of success of an attack.
Games are formalized in a process calculus:

- It is adapted from the pi calculus.
- The semantics is purely probabilistic (no non-determinism).
- The runtime of processes is bounded:
  - bounded number of copies of processes,
  - bounded length of messages on channels.
- Extension to arrays.
Process calculus for games: terms

Terms represent computations on messages (bitstrings).

\[ M ::= \]

- variable
- \[ f(M_1, \ldots, M_n) \] function application

Function symbols \( f \) correspond to functions computable by deterministic Turing machines that always terminate.
Process calculus for games: processes

\[
Q ::= \begin{align*}
0 & \quad \text{end} \\
Q \parallel Q' & \quad \text{parallel composition} \\
!i \leq N \ Q & \quad \text{replication } N \text{ times} \\
\text{newChannel} \ c; Q & \quad \text{restriction for channels} \\
in(c, (x_1 : T_1, \ldots, x_m : T_m)); P & \quad \text{input}
\end{align*}
\]

\[
P ::= \begin{align*}
yield & \quad \text{end} \\
\text{out}(c, (M_1, \ldots, M_m)); Q & \quad \text{output} \\
\text{event} \ e(M_1, \ldots, M_m); P & \quad \text{event} \\
\text{new} \ x : T; P & \quad \text{random number generation (uniform)} \\
\text{let} \ x : T = M \ \text{in} \ P & \quad \text{assignment} \\
\text{if} \ M \ \text{then} \ P \ \text{else} \ P' & \quad \text{conditional} \\
\text{find} \ j \leq N \ \text{such that defined} (x[j], \ldots) \wedge M \ \text{then} \ P \ \text{else} \ P' & \quad \text{array lookup}
\end{align*}
\]
Example: 1. symmetric encryption

We consider a probabilistic, length-revealing encryption scheme.

Definition (Symmetric encryption scheme SE)

- (Randomized) key generation function \( kgen \).
- (Randomized) encryption function \( enc(m, k, r') \) takes as input a message \( m \), a key \( k \), and random coins \( r' \).
- Decryption function \( dec(c, k) \) such that

\[
dec(enc(m, kgen(r), r'), kgen(r)) = i_\bot(m)
\]

The decryption returns a bitstring or \( \bot \):

- \( \bot \) when decryption fails,
- the cleartext when decryption succeeds.

The injection \( i_\bot \) maps a bitstring to the same bitstring in bitstring \( \cup \{ \bot \} \).
Example: 2. MAC

Definition (Message Authentication Code scheme MAC)

- (Randomized) key generation function \( mkgen \).
- MAC function \( mac(m, k) \) takes as input a message \( m \) and a key \( k \).
- Verification function \( verify(m, k, t) \) such that
  \[
  verify(m, k, mac(m, k)) = true.
  \]

A MAC is essentially a keyed hash function.

A MAC guarantees the integrity and authenticity of the message because only someone who knows the secret key can build the MAC.
Example: 3. encrypt-then-MAC

We define an authenticated encryption scheme by the encrypt-then-MAC construction:

\[
\text{enc}'(m, (k, mk), r'') = e, \text{mac}(e, mk) \text{ where } e = \text{enc}(m, k, r'').
\]

A basic example of protocol using encrypt-then-MAC:

- \(A\) and \(B\) initially share an encryption key \(k\) and a MAC key \(mk\).
- \(A\) sends to \(B\) a fresh key \(k'\) encrypted under authenticated encryption, implemented as encrypt-then-MAC.

\[
A \rightarrow B : e = \text{enc}(k', k, r''), \text{mac}(e, mk) \quad k' \text{ fresh}
\]

\(k'\) should remain secret.
Example: initialization

\[ A \to B : e = \text{enc}(k', k, r''), \text{mac}(e, mk) \quad k' \text{ fresh} \]

\[ Q_0 = \text{in}(\text{start}, ()); \text{new } r : \text{keyseed}; \text{let } k : \text{key} = \text{kgen}(r) \text{ in} \]
\[ \text{new } r' : \text{mkeyseed}; \text{let } mk : \text{mkey} = \text{mkgen}(r') \text{ in } \text{out}(c, ()); \]
\[ (Q_A \parallel Q_B) \]

Initialization of keys:

1. The process \( Q_0 \) waits for a message on channel \( \text{start} \) to start running.
   The adversary triggers this process.

2. \( Q_0 \) generates encryption and MAC keys, \( k \) and \( mk \) respectively, using the key generation algorithms \( \text{kgen} \) and \( \text{mkgen} \).

3. \( Q_0 \) returns control to the adversary by the output \( \text{out}(c, ()) \).
   \( Q_A \) and \( Q_B \) represent the actions of \( A \) and \( B \) (see next slides).
Example: role of $A$

$A \rightarrow B : e = \text{enc}(k', k, r'')$, $\text{mac}(e, mk)$ \quad k'$ fresh

$$Q_A = \not^i \leq^n \text{in}(c_A,()); \text{new } k' : \text{key}; \text{new } r'' : \text{coins};$$

$$\text{let } e : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \text{ in}$$

$$\text{out}(c_A, (e, \text{mac}(e, mk)))$$

Role of $A$:

1. $\not^i \leq^n$ represents $n$ copies, indexed by $i \in [1, n]$
   The protocol can be run $n$ times (polynomial in the security parameter).
2. The process is triggered when a message is sent on $c_A$ by the adversary.
3. The process chooses a fresh key $k'$ and sends the message on channel $c_A$. 
Example: role of $B$

$A \rightarrow B : e = \text{enc}(k', k, r'')$, $\text{mac}(e, mk)$ \quad k'$ fresh

$$Q_B = !^{i'' \leq n} \text{in}(c_B, (e' : \text{bitstring}, ma : \text{macstring}));$$

\quad \text{if } \text{verify}(e', mk, ma) \text{ then}

\quad \text{let } i_\bot(k2b(k'')) = \text{dec}(e', k) \text{ in out}(c_B, ())

Role of $B$:

1. $n$ copies, as for $Q_A$.
2. The process $Q_B$ waits for the message on channel $c_B$.
3. It verifies the MAC, decrypts, and stores the key in $k''$. 
Example: summary of the initial game

\[A \rightarrow B : e = enc(k', k, r''), \text{mac}(e, mk) \quad k' \text{ fresh}\]

\[Q_0 = \text{in}(\text{start}, ()); \text{new } r : \text{keyseed}; \text{let } k : \text{key} = \text{kgen}(r) \text{ in}\]
\[\text{new } r' : \text{mkeyseed}; \text{let } mk : \text{mkey} = \text{mkgen}(r') \text{ in } \text{out}(c, ());\]
\[(Q_A \parallel Q_B)\]

\[Q_A = !^{i \leq n} \text{in}(c_A, ()); \text{new } k' : \text{key}; \text{new } r'' : \text{coins};\]
\[\text{let } e : \text{bitstring} = enc(k2b(k'), k, r'') \text{ in}\]
\[\text{out}(c_A, (e, \text{mac}(e, mk)))\]

\[Q_B = !^{i'' \leq n} \text{in}(c_B, (e' : \text{bitstring}, ma : \text{macstring}));\]
\[\text{if verify}(e', mk, ma) \text{ then}\]
\[\text{let } i_{\perp}(k2b(k'')) = \text{dec}(e', k) \text{ in } \text{out}(c_B, ())\]
Security assumptions on primitives

The most frequent cryptographic primitives are already specified in a library. The user can use them without redefining them.

In the example:

- The MAC is UF-CMA (unforgeable under chosen message attacks). An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).
The most frequent cryptographic primitives are already specified in a library. The user can use them without redefining them.

In the example:

- The MAC is **UF-CMA** (unforgeable under chosen message attacks). An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).

- The encryption is **IND-CPA** (indistinguishable under chosen plaintext attacks). An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.
Security assumptions on primitives

The most frequent cryptographic primitives are already specified in a library. The user can use them without redefining them.

In the example:

- The MAC is **UF-CMA** (unforgeable under chosen message attacks). An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (for a message on which the MAC oracle has not been called).

- The encryption is **IND-CPA** (indistinguishable under chosen plaintext attacks). An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.

- All keys have the same length: \( \text{forall } y : \text{key}; Z(k2b(y)) = Z_k. \)
Security properties to prove

In the example:

- **One-session secrecy** of $k''$: each $k''$ is indistinguishable from a random number.
- **Secrecy** of $k''$: the $k''$ are indistinguishable from independent random numbers.
Demo

- CryptoVerif input file: enc-then-MAC.cv
- library of primitives
- run CryptoVerif
- output
Arrays

A variable defined under a replication is implicitly an array:

\[
Q_A = \forall i \leq n \text{ in } (c_A, ()); \text{ new } k'[i] : \text{ key}; \text{ new } r''[i] : \text{ coins}; \\
\text{ let } e[i] : \text{ bitstring } = \text{ enc} (k2b(k'[i]), k, r''[i]) \text{ in } \\
\text{ out} (c_A, (e[i], \text{ mac}(e[i], mk)))
\]

Requirements:

- Only variables with the current indices can be assigned.
- Variables may be defined at several places, but only one definition can be executed for the same indices.

(\text{if } \ldots \text{ then let } x = M \text{ in } P \text{ else let } x = M' \text{ in } P' \text{ is ok})

So each array cell can be assigned at most once.

**Arrays allow one to remember the values of all variables during the whole execution**
Arrays (continued)

**find** performs an array lookup:

\[
!^{i \leq N} \ldots \text{let } x = M \text{ in } P \\
|| !^{i' \leq N'} \text{ in } (c, y : T); \text{find } j \leq N \text{ such that defined}(x[j]) \land y = x[j] \text{ then } \ldots
\]

Note that **find** is here used outside the scope of \( x \).

This is the only way of getting access to values of variables in other sessions.

When several array elements satisfy the condition of the **find**, the returned index is chosen randomly, with uniform probability.
Arrays versus lists

Arrays replace lists often used in cryptographic proofs.

\[ !^{i \leq N} \ldots \text{let } x = M \text{ in let } y = M' \text{ in } P \]
\[ \parallel !^{i' \leq N'} \text{ in } (c, x' : T); \text{find } j \leq N \text{ such that defined}(x[j]) \land x' = x[j] \text{ then } P'(y[j]) \]

written by cryptographers

\[ !^{i \leq N} \ldots \text{let } x = M \text{ in let } y = M' \text{ in insert } (x, y) \text{ in } L; P \]
\[ \parallel !^{i'' \leq N'} \text{ in } (c, x' : T); \text{get } (x, y) \text{ in } L \text{ such that } x' = x; P'(y) \]

Arrays avoid the need for explicit list insertion instructions, which would be hard to guess for an automatic tool.
Indistinguishability

Two processes (games) $Q_1$, $Q_2$ are indistinguishable when the adversary has a negligible probability of distinguishing them: $Q_1 \approx Q_2$.

- The adversary is represented by an acceptable evaluation context $C ::= [\ ] \ C \parallel Q \ Q \parallel C \ \text{newChannel} \ c; C$.
- $C[Q]$ may execute events, collected in a sequence $E$.
- A distinguisher $D$ takes as input $E$ and returns true or false.
  - Example: $D_e(E) = \text{true}$ if and only if $e \in E$.
  - $D_e$ abbreviated $e: e(E)$ is true if and only if $e \in E$.
- $\Pr[C[Q] : D]$ is the probability that $C[Q]$ executes $E$ such that $D(E) = \text{true}$.

Definition (Indistinguishability)

We write $Q \approx_p^V Q'$ when, for all evaluation contexts $C$ acceptable for $Q$ and $Q'$ with public variables $V$ and all distinguishers $D$,

$$| \Pr[C[Q] : D] - \Pr[C[Q'] : D] | \leq p(C, D).$$
Indistinguishability

Lemma

1. **Reflexivity:** $Q \approx^V_0 Q$.
2. **Symmetry:** $\approx^V_p$ is symmetric.
3. **Transitivity:** if $Q \approx^V_p Q'$ and $Q' \approx^V_p Q''$, then $Q \approx^V_{p+p'} Q''$.
4. **Application of context:** if $Q \approx^V_p Q'$ and $C$ is an evaluation context acceptable for $Q$ and $Q'$ with public variables $V$, then $C[Q] \approx^V_{p'} C[Q']$, where $p'(C', D) = p(C'[C[]], D)$ and $V' \subseteq V \cup \text{var}(C)$. 
Proof technique

We transform a game $G_0$ into an indistinguishable one using:

- **indistinguishability properties** $L \approx_p R$ given as axioms and that come from security assumptions on primitives. These equivalences are used inside a context:

$$G_1 \approx_0 C[L] \approx_{p'} C[R] \approx_0 G_2$$

- **syntactic transformations**: simplification, expansion of assignments, ...

We obtain a sequence of games $G_0 \approx_{p_1} G_1 \approx \ldots \approx_{p_m} G_m$, which implies

$G_0 \approx_{p_1 + \ldots + p_m} G_m$.

If some trace property holds up to probability $p$ in $G_m$, then it holds up to probability $p + p_1 + \ldots + p_m$ in $G_0$. 
A MAC guarantees the integrity and authenticity of the message because only someone who knows the secret key can build the MAC. More formally, $\text{Succ}_{\text{MAC}}^{\text{uf-cma}}(t, q_m, q_v, l)$ is negligible if $t$ is polynomial in the security parameter:

\[
\text{Succ}_{\text{MAC}}^{\text{uf-cma}}(t, q_m, q_v, l) = \\
\max_{A} \Pr \left[ k \xleftarrow{\text{R}} \text{mkgen}; (m, s) \leftarrow A^{\text{mac}(.,k), \text{verify}(.,k,.)} : \text{verify}(m, k, s) \wedge m \text{ was never queried to the oracle } \text{mac}(., k) \right]
\]

where $A$ runs in time at most $t$, calls $\text{mac}(., k)$ at most $q_m$ times with messages of length at most $l$, calls $\text{verify}(., k, .)$ at most $q_v$ times with messages of length at most $l$. 

Bruno Blanchet (INRIA)
MAC: intuition behind the CryptoVerif definition

By the previous definition, up to negligible probability,

- the adversary cannot forge a correct MAC

- so, assuming $k \leftarrow^R mkgen$ is used only for generating and verifying MACs, the verification of a MAC with $\text{verify}(m, k, t)$ can succeed only if $m$ is in the list (array) of messages whose $\text{mac}(\cdot, k)$ has been computed by the protocol

- so we can replace a call to verify with an array lookup:
  if the call to $\text{mac}$ is $\text{mac}(x, k)$, we replace $\text{verify}(m, k, t)$ with

$$\text{find } j \leq N \text{ such that defined}(x[j]) \land (m = x[j]) \land \text{verify}(m, k, t) \text{ then true else false}$$
MAC: CryptoVerif definition

\[ \text{verify}(m, \text{mkgen}(r), \text{mac}(m, \text{mkgen}(r))) = \text{true} \]

\[ \!^N\!
\text{new } r : \text{mkeyseed}; (\]
\[ \!^N \text{Omac}(x : \text{bitstring}) := \text{mac}(x, \text{mkgen}(r)), \]
\[ \!^N' \text{Overify}(m : \text{bitstring}, t : \text{macstring}) := \text{verify}(m, \text{mkgen}(r), t) \]
\[ \approx \]

\[ \!^N''\!
\text{new } r : \text{mkeyseed}; (\]
\[ \!^N \text{Omac}(x : \text{bitstring}) := \text{mac}(x, \text{mkgen}(r)), \]
\[ \!^N' \text{Overify}(m : \text{bitstring}, t : \text{macstring}) := \]
\[ \text{find } j \leq N \text{ such that defined}(x[j]) \land (m = x[j]) \land \]
\[ \text{verify}(m, \text{mkgen}(r), t) \text{ then true else false} \]
MAC: CryptoVerif definition

\[ \text{verify}(m, \text{mkgen}(r), \text{mac}(m, \text{mkgen}(r))) = \text{true} \]

\[ !^N'' \text{new } r : \text{mkeyseed}; ( \]
\[ !^N \text{Omac}(x : \text{bitstring}) := \text{mac}(x, \text{mkgen}(r)), \]
\[ !^N' \text{Overify}(m : \text{bitstring}, t : \text{macstring}) := \text{verify}(m, \text{mkgen}(r), t) \]

\[ \approx \]
\[ !^N'' \times \text{Succ}_{\text{MAC}}^{\text{uf-cma}} (\text{time} + (N'' - 1)(\text{time}(\text{mkgen}) + N \times \text{time}(\text{mac}, \text{maxl}(x))) + N' \times \text{time}(\text{verify}, \text{maxl}(m)), N, N', \text{max}(\text{maxl}(x), \text{maxl}(m))) \]

\[ !^N'' \text{new } r : \text{mkeyseed}; ( \]
\[ !^N \text{Omac}(x : \text{bitstring}) := \text{mac'}(x, \text{mkgen'}(r)), \]
\[ !^N' \text{Overify}(m : \text{bitstring}, t : \text{macstring}) := \]
\[ \text{find } j \leq N \text{ suchthat defined}(x[j]) \land (m = x[j]) \land \]
\[ \text{verify'}(m, \text{mkgen'}(r), t) \text{ then true else false} \]

CryptoVerif understands such specifications of primitives.
Exercise 1

The advantage of the adversary against strong unforgeability under chosen message attacks (SUF-CMA) of MACs is:

\[
\text{Succ}_{\text{MAC}}^{\text{suf-cma}}(t, q_m, q_v, l) = \\
\max_A \Pr[k \leftarrow \text{mkgen}; (m, s) \leftarrow A^{\text{mac}(., k), \text{verify}(., k, .)} : \text{verify}(m, k, s) \land s \text{ is not the result of calling the oracle } \text{mac}(., k) \text{ on } m]
\]

where \(A\) runs in time at most \(t\),
calls \(\text{mac}(., k)\) at most \(q_m\) times with messages of length at most \(l\),
calls \(\text{verify}(., k, .)\) at most \(q_v\) times with messages of length at most \(l\).

Represent SUF-CMA MACs in the CryptoVerif formalism.
Exercise 2

A signature scheme \( S \) consists of

- a key generation algorithm \( (pk, sk) \xleftarrow{\$} kgen \)
- a signature algorithm \( \text{sign}(m, sk) \)
- a verification algorithm \( \text{verify}(m, pk, s) \)

such that \( \text{verify}(m, pk, \text{sign}(m, sk)) = 1 \).

The advantage of the adversary against unforgeability under chosen message attacks (UF-CMA) of signatures is:

\[
\text{Succ}_{\text{uf-cma}}^S(t, q_s, l) = \max_A \Pr \left[ (pk, sk) \xleftarrow{\$} kgen; (m, s) \leftarrow A^{\text{sign}(\cdot, sk)}(pk) : \text{verify}(m, pk, s) \land m \text{ was never queried to the oracle } \text{sign}(\cdot, sk) \right]
\]

where \( A \) runs in time at most \( t \),
calls \( \text{sign}(\cdot, sk) \) at most \( q_s \) times with messages of length at most \( l \).

Represent UF-CMA signatures in the CryptoVerif formalism.
MAC: using the CryptoVerif definition

CryptoVerif applies the previous rule automatically in any context, perhaps containing several occurrences of $mac$ and of $verify$:

- Each occurrence of $mac$ is replaced with $mac'$.
- Each occurrence of $verify$ is replaced with a **find** that looks in all arrays of computed MACs (one array for each occurrence of function $mac$).
Symmetric encryption: definition of security (IND-CPA)

An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.

**Definition (INDistinguishability under Chosen Plaintext Attacks, IND-CPA)**

\[
\text{Succ}_{\text{SE}}^{\text{ind-cpa}}(t, q_e, l) = \max_{A} 2 \Pr \left[ b \leftarrow \{0, 1\}; k \leftarrow k_{\text{gen}}; b' \leftarrow A^{\text{enc}}(LR(.,.,b),k) : b' = b \right] - 1
\]

where \( A \) runs in time at most \( t \),
calls \( \text{enc}(LR(.,.,b),k) \) at most \( q_e \) times on messages of length at most \( l \),
\( LR(x, y, 0) = x, LR(x, y, 1) = y \), and \( LR(x, y, b) \) is defined only when \( x \) and \( y \) have the same length.
Symmetric encryption: CryptoVerif definition

\[ \text{dec}(\text{enc}(m, \text{kgen}(r), r'), \text{kgen}(r)) = \bot(m) \]

\[ \!^{N'} \text{new } r : \text{keyseed}; \!^{N} \text{Oenc}(x : \text{bitstring}) := \]
\[ \text{new } r' : \text{coins}; \text{enc}(x, \text{kgen}(r), r') \]
\[ \approx \]

\[ \!^{N'} \text{new } r : \text{keyseed}; \!^{N} \text{Oenc}(x : \text{bitstring}) := \]
\[ \text{new } r' : \text{coins}; \text{enc}(Z(x), \text{kgen}(r), r') \]

\(Z(x)\) is the bitstring of the same length as \(x\) containing only zeroes (for all \(x : \text{nonce}, Z(x) = Z_{\text{nonce}}, \ldots\)).
Symmetric encryption: CryptoVerif definition

\[
dec(enc(m, kgen(r), r'), kgen(r)) = i\bot(m)
\]

\[
!^{N'} \textbf{new } r : \text{keyseed}; !^N \textit{Oenc}(x : \text{bitstring}) := \\
\textbf{new } r' : \textit{coins}; \text{enc}(x, kgen(r), r')
\]

\[
\approx N' \times \text{Succ}_{\text{SE}}^{\text{ind-cpa}}(\text{time} + (N' - 1)(\text{time}(kgen) + N \times \text{time}(\text{enc}, \text{maxl}(x)) + N \times \text{time}(Z, \text{maxl}(x))), N', \text{maxl}(x))
\]

\[
!^{N'} \textbf{new } r : \text{keyseed}; !^N \textit{Oenc}(x : \text{bitstring}) := \\
\textbf{new } r' : \textit{coins}; \textit{enc}'(Z(x), kgen'(r), r')
\]

\(Z(x)\) is the bitstring of the same length as \(x\) containing only zeroes (for all \(x : \text{nonce}, Z(x) = Z\text{nonce}, \ldots\)).
Exercise 3

The advantage of the adversary against ciphertext integrity (INT-CTXT) of a symmetric encryption scheme SE is:

\[
\text{Succ}_{\text{SE}}^{\text{int-ctxt}}(t, q_e, q_d, l_e, l_d) = \max_{\mathcal{A}} \Pr \left[ k \leftarrow \text{kgen}; c \leftarrow \mathcal{A}^{\text{enc}(\cdot, k), \text{dec}(\cdot, k) \neq \bot} : \text{dec}(c, k) \neq \bot \wedge c \text{ is not the result of a call to the enc(\cdot, k) oracle} \right]
\]

where \( \mathcal{A} \) runs in time at most \( t \),
calls \( \text{enc}(\cdot, k) \) at most \( q_e \) times with messages of length at most \( l_e \),
calls \( \text{dec}(\cdot, k) \neq \bot \) at most \( q_d \) times with messages of length at most \( l_d \).

Represent INT-CTXT encryption in the CryptoVerif formalism.
Exercise 4

A public-key encryption scheme $AE$ consists of

- a key generation algorithm $(pk, sk) \leftarrow kgen$
- a probabilistic encryption algorithm $enc(m, pk)$
- a decryption algorithm $dec(m, sk)$

such that $dec(enc(m, pk), sk) = m$.

The advantage of the adversary against indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA2) is

$$Succ_{AE}^{ind-cca2}(t, q_d) = \max_A 2 \Pr \left[ b \leftarrow \{0, 1\}; (pk, sk) \leftarrow kgen; (m_0, m_1, s) \leftarrow A_1^{dec(., sk)}(pk); y \leftarrow enc(m_b, pk); b' \leftarrow A_2^{dec(., sk)}(m_0, m_1, s, y) : b' = b \wedge A_2 \text{ has not called } dec(., sk) \text{ on } y \right] - 1$$

where $A = (A_1, A_2)$ runs in time at most $t$ and calls $dec(., sk)$ at most $q_d$ times. Represent IND-CCA2 encryption in the CryptoVerif formalism.
Syntactic transformations (1)

Expansion of assignments: replacing a variable with its value. (Not completely trivial because of array references.)

Example

If \( mk \) is defined by

\[
\text{let } mk = mk\text{gen}(r')
\]

and there are no array references to \( mk \), then \( mk \) is replaced with \( mk\text{gen}(r') \) in the game and the definition of \( mk \) is removed.
Syntactic transformations (2)

**Single assignment renaming**: when a variable is assigned at several places, rename it with a distinct name for each assignment. (Not completely trivial because of array references.)

**Example**

```plaintext
in(start, ()); new r_A : T_r; let k_A = kgen(r_A) in
  new r_B : T_r; let k_B = kgen(r_B) in out(c, ()); (Q_K || Q_S)

Q_K = !^{i \leq n} in(c, (h : T_h, k : T_k))
  if h = A then let k' = k_A else
  if h = B then let k' = k_B else let k' = k

Q_S = !^{i' \leq n'} in(c', h' : T_h);
  find j \leq n such that defined(h[j], k'[j]) \land h' = h[j] then P_1(k'[j])
  else P_2
```
Syntactic transformations (2)

Single assignment renaming: when a variable is assigned at several places, rename it with a distinct name for each assignment.
(Not completely trivial because of array references.)

Example

```plaintext
 utilizando start,(); new r_A : T_r; let k_A = kgen(r_A) in
     new r_B : T_r; let k_B = kgen(r_B) in out(c,()); (Q_K ∥ Q_S)

Q_K = !i≤n in(c, (h : T_h, k : T_k))
    if h = A then let k'_1 = k_A else
    if h = B then let k'_2 = k_B else let k'_3 = k

Q_S = !i'≤n' in(c', h' : T_h);
    find j ≤ n suchthat defined(h[j], k'_1[j]) ∧ h' = h[j] then P_1(k'_1[j])
    orfind j ≤ n suchthat defined(h[j], k'_2[j]) ∧ h' = h[j] then P_1(k'_2[j])
    orfind j ≤ n suchthat defined(h[j], k'_3[j]) ∧ h' = h[j] then P_1(k'_3[j])
    else P_2
```
Move new: move restrictions downwards in the game as much as possible, when there is no array reference to them.
(Moving \texttt{new } \texttt{x : T} under a \texttt{if} or a \texttt{find} duplicates it.
A subsequent single assignment renaming will distinguish cases.)

\begin{example}
\texttt{new x : nonce; if c then P_1 else P_2}
becomes
\texttt{if c then new x : nonce; P_1 else new x : nonce; P_2}
\end{example}
Syntactic transformations (4)

- **Merge arrays**: merge several variables $x_1, \ldots, x_n$ into a single variable $x_1$ when they are used for different indices (defined in different branches of a test `if` or `find`).

- **Merge branches of `if` or `find** when they execute the same code, up to renaming of variables with array accesses.
Simplification and elimination of collisions

- CryptoVerif collects equalities that come from:
  - **Assignments**: let $x = M$ in $P$ implies that $x = M$ in $P$
  - **Tests**: if $M = N$ then $P$ implies that $M = N$ in $P$
  - **Definitions of cryptographic primitives**
    - When a find guarantees that $x[j]$ is defined, equalities that hold at definition of $x$ also hold under the find (after substituting $j$ for the array indices at the definition of $x$)
  - **Elimination of collisions**: if $x$ is created by new $x : T$, $x[i] = x[j]$ implies $i = j$, up to negligible probability (when $T$ is large)

- These equalities are combined to simplify terms.
- When terms can be simplified, processes are simplified accordingly. For instance:
  - If $M$ simplifies to **true**, then if $M$ then $P_1$ else $P_2$ simplifies $P_1$.
  - If a condition of find simplifies to **false**, then the corresponding branch is removed.
Proof of security properties: one-session secrecy

**One-session secrecy**: the adversary cannot distinguish any of the secrets from a random number with one test query.

**Definition (One-session secrecy)**

Assume that the variable $x$ of type $T$ is defined in $G$ under a single $i \leq n$. $G$ preserves the one-session secrecy of $x$ up to probability $p$ when, for all evaluation contexts $C$ acceptable for $G \parallel Q_x$ with no public variables that do not contain $S$, $2 \Pr[C[G \parallel Q_x] : S] - 1 \leq p(C)$, where

$$Q_x = \text{in}(c_0, ()); \text{new } b : \text{bool}; \text{out}(c_0, ()) ; \begin{cases} \text{in}(c, j : [1, n]); \text{if defined}(x[j]) \text{ then} \\ \quad \text{if } b \text{ then } \text{out}(c, x[j]) \text{ else } \text{new } y : T; \text{out}(c, y) \\ \parallel \text{in}(c', b' : \text{bool}); \text{if } b = b' \text{ then event } S \end{cases}$$

$c_0, c, c', b, b', j, y, \text{ and } S$ do not occur in $G$. 
**Proof of security properties: one-session secrecy**

**One-session secrecy:** the adversary cannot distinguish any of the secrets from a random number with one test query.

**Criterion for proving one-session secrecy of** $x$:

$x$ is defined by `new $x[i] : T` and there is a set of variables $S$ such that only variables in $S$ depend on $x$.

The output messages and the control-flow do not depend on $x$. 
**Secrecy:** the adversary cannot distinguish the secrets from independent random numbers with several test queries.

**Criterion for proving secrecy of** $x$: same as one-session secrecy, plus $x[i]$ and $x[i']$ do not come from the same copy of the same restriction when $i \neq i'$. 

---

*Bruno Blanchet (INRIA)*

*CryptoVerif*
Proof strategy: advice

1. One tries to execute each transformation given by the definition of a cryptographic primitive.
2. When it fails, it tries to analyze why the transformation failed, and suggests syntactic transformations that could make it work.
3. One tries to execute these syntactic transformations. (If they fail, they may also suggest other syntactic transformations, which are then executed.)
4. We retry the cryptographic transformation, and so on.
Proof of the example: initial game

\[ Q_0 = \text{in}(\text{start},()); \text{new } r : \text{keyseed}; \text{let } k : \text{key} = kgen(r) \text{ in} \]
\[ \text{new } r' : \text{mkeyseed}; \text{let } mk : \text{mkey} = mkgen(r') \text{ in out}(c,()); \]
\[ (Q_A \parallel Q_B) \]

\[ Q_A = \!i^{\leq n}\text{in}(c_A,()); \text{new } k' : \text{key}; \text{new } r'' : \text{coins}; \]
\[ \text{let } m : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \text{ in} \]
\[ \text{out}(c_A, (m, \text{mac}(m, mk))) \]

\[ Q_B = \!i''^{\leq n}\text{in}(c_B, (m' : \text{bitstring}, ma : \text{macstring})); \]
\[ \text{if } \text{verify}(m', mk, ma) \text{ then} \]
\[ \text{let } i_{\bot}(k2b(k'')) = \text{dec}(m', k) \text{ in out}(c_B,()) \]

Bruno Blanchet (INRIA)
Proof of the example: remove assignments \( mk \)

\[
\begin{align*}
Q_0 & = \text{in}(\text{start}, ()); \text{new } r : \text{keyseed}; \text{let } k : \text{key} = \text{kgen}(r) \text{ in } \\
& \quad \text{new } r' : m\text{keyseed}; \text{out}(c, ()); (Q_A \parallel Q_B) \\
Q_A & = !i^{\leq n} \text{in}(c_A, ()); \text{new } k' : \text{key}; \text{new } r'' : \text{coins}; \\
& \quad \text{let } m : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \text{ in } \\
& \quad \text{out}(c_A, (m, \text{mac}(m, \text{mkgen}(r')))) \\
Q_B & = !i''^{\leq n} \text{in}(c_B, (m' : \text{bitstring}, ma : \text{macstring})); \\
& \quad \text{if } \text{verify}(m', \text{mkgen}(r'), ma) \text{ then } \\
& \quad \text{let } i\perp(k2b(k'')) = \text{dec}(m', k) \text{ in out}(c_B, ())
\end{align*}
\]
Proof of the example: security of the MAC

\[ Q_0 = \textbf{in}(\textit{start}, ()); \textbf{new} \ r : \textit{keyseed}; \textbf{let} \ k : \textit{key} = \textit{kgen}(r) \textbf{ in} \]
\[ \textbf{new} \ r' : \textit{mkeyseed}; \textbf{out}(c, ()); (Q_A \parallel Q_B) \]

\[ Q_A = !i^\leq n \textbf{in}(c_A, ()); \textbf{new} \ k' : \textit{key}; \textbf{new} \ r'' : \textit{coins}; \]
\[ \textbf{let} \ m : \textit{bitstring} = \textit{enc}(k2b(k'), k, r'') \textbf{ in} \]
\[ \textbf{out}(c_A, (m, \textit{mac'}(m, \textit{mkgen'}(r')))) \]

\[ Q_B = !i'^\leq n \textbf{in}(c_B, (m' : \textit{bitstring}, ma : \textit{macstring})); \]
\[ \textbf{find} \ j \leq n \textbf{ suchthat defined}(m[j]) \land m' = m[j] \land \]
\[ \textit{verify'}(m', \textit{mkgen'}(r'), ma) \textbf{ then} \]
\[ \textbf{let} \ i_\bot(k2b(k'')) = \textit{dec}(m', k) \textbf{ in} \textbf{out}(c_B, ()) \]

Probability: \[ \text{Succ}^\text{uf-cma}_{\text{MAC}}(\text{time} + \text{time}(\textit{kgen}) + n \text{ time}(\textit{enc}, \text{length}(\textit{key})) + \]
\[ n \text{ time}(\textit{dec}, \text{maxl}(m'))) \land n, n, \text{max}(\text{maxl}(m'), \text{maxl}(m))). \]
Proof of the example: simplify

\[
Q_0 = \text{in}(\text{start}, ()); \text{new } r : \text{keyseed}; \text{let } k : \text{key} = \text{kgen}(r) \text{ in }
\text{new } r' : \text{mkeyseed}; \text{out}(c, ()); (Q_A \parallel Q_B)
\]

\[
Q_A = i \leq n \text{ in}(c_A, ()); \text{new } k' : \text{key}; \text{new } r'' : \text{coins};
\text{let } m : \text{bitstring} = \text{enc}(k2b(k'), k, r'') \text{ in }
\text{out}(c_A, (m, \text{mac'}(m, \text{mkgen'}(r''))))
\]

\[
Q_B = i' \leq n \text{ in}(c_B, (m' : \text{bitstring}, ma : \text{macstring}));
\text{find } j \leq n \text{ such that } \text{defined}(m[j]) \land m' = m[j] \land
\text{verify'}(m', \text{mkgen'}(r'), ma) \text{ then }
\text{let } k'' = k'[j] \text{ in out}(c_B, ())
\]

\[
dec(m', k) = dec(\text{enc}(k2b(k'[j]), k, r''[j]), k) = \perp(k2b(k'[j]))
\]
Proof of the example: remove assignments $k$

$$Q_0 = \text{in}(\text{start}, ()); \text{new } r : \text{keyseed}; \text{new } r' : \text{mkeyseed}; \text{out}(c, ()) ;
(Q_A \parallel Q_B)$$

$$Q_A = !^{i \leq n} \text{in}(c_A, ()) ; \text{new } k' : \text{key}; \text{new } r'' : \text{coins};$$
$$\text{let } m : \text{bitstring} = \text{enc}(k2b(k'), k\text{gen}(r), r'') \text{ in}$$
$$\text{out}(c_A, (m, \text{mac'}(m, m\text{gen'}(r''))))$$

$$Q_B = !^{i' \leq n} \text{in}(c_B, (m' : \text{bitstring}, m' : \text{macstring}));$$
$$\text{find } j \leq n \text{ such that defined}(m[j]) \land m' = m[j] \land$$
$$\text{verify'}(m', m\text{gen'}(r'), ma) \text{ then}$$
$$\text{let } k'' = k'[j] \text{ in out}(c_B, ())$$
Proof of the example: security of the encryption

\[ Q_0 = \text{in}(\text{start}, ()); \textbf{new} \ r : \text{keyseed}; \textbf{new} \ r' : \text{mkeyseed}; \textbf{out}(c, ()); \]
\[ (Q_A \parallel Q_B) \]

\[ Q_A = !i \leq n \text{in}(c_A, ()); \textbf{new} \ k' : \text{key}; \textbf{new} \ r'' : \text{coins}; \]
\[ \text{let} \ m : \text{bitstring} = \text{enc}'(Z(k2b(k')), kgen'(r), r'') \text{ in} \]
\[ \text{out}(c_A, (m, \text{mac}'(m, mkgen'(r')))) \]

\[ Q_B = !i' \leq n \text{in}(c_B, (m' : \text{bitstring}, ma : \text{macstring})); \]
\[ \text{find} \ j \leq n \text{ suchthat defined}(m[j]) \land m' = m[j] \land \]
\[ \text{verify}'(m', mkgen'(r'), ma) \text{ then} \]
\[ \text{let} \ k'' = k'[j] \text{ in} \text{out}(c_B, ()) \]

Probability: \[ \text{Succ}_{SE}^{\text{ind-cpa}}(\text{time} + (n + n^2)\text{time}(mkgen) + n \text{time}(mac, \text{maxl}(m)) + n^2 \text{time}(\text{verify}, \text{maxl}(m')) + n^2 \text{time}(= \text{bitstring}, \text{maxl}(m'), \text{maxl}(m)), n, \text{length(key)))} \]
Proof of the example: security of the encryption

\[ Q_0 = \text{in}(\text{start},()); \text{new } r : \text{keyseed}; \text{new } r' : \text{mkeyseed}; \text{out}(c,()); \]
\[ (Q_A \parallel Q_B) \]

\[ Q_A = !^i \leq n \text{in}(c_A,()); \text{new } k' : \text{key}; \text{new } r'' : \text{coins}; \]
\[ \text{let } m : \text{bitstring} = \text{enc}'(Z(k2b(k')), k\text{gen}'(r), r'') \text{ in} \]
\[ \text{out}(c_A, (m, \text{mac}'(m, \text{mk\text{gen}}'(r')))) \]

\[ Q_B = !^j \leq n \text{in}(c_B, (m' : \text{bitstring, ma : macstring})); \]
\[ \text{find } j \leq n \text{ suchthat defined}(m[j]) \land m' = m[j] \land \]
\[ \text{verify}'(m', \text{mk\text{gen}}'(r'), ma) \text{ then} \]
\[ \text{let } k'' = k'[j] \text{ in out}(c_B,()) \]

Better probability: \( \text{Succ}^{\text{ind-cpa}}_{\text{SE}}(\text{time} + (n + n^2)\text{time}(\text{mk\text{gen}}) + \)
\( n \text{time}(\text{mac, maxl}(m)) + n^2 \text{time}(\text{verify, maxl}(m')) + \)
\( n^2 \text{time}(= \text{bitstring, maxl}(m'), \text{maxl}(m)), n, \text{length(key)))} \)
Proof of the example: simplify

\[ Q_0 = \text{in}(\text{start}, ()); \text{new } r : \text{keyseed}; \text{new } r' : \text{mkeyseed}; \text{out}(c, ()); (Q_A \parallel Q_B) \]

\[ Q_A = !^i \leq n \text{ in}(c_A, ()); \text{new } k' : \text{key}; \text{new } r'' : \text{coins}; \]
\[ \text{let } m : \text{bitstring} = \text{enc}'(Z_k, kgen'(r), r'') \text{ in} \]
\[ \text{out}(c_A, (m, mac'(m, mkgen'(r'))))) \]

\[ Q_B = !^i' \leq n \text{ in}(c_B, (m' : \text{bitstring}, ma : \text{macstring})); \]
\[ \text{find } j \leq n \text{ suchthat defined}(m[j]) \land m' = m[j] \land \]
\[ \text{verify}'(m', mkgen'(r'), ma) \text{ then} \]
\[ \text{let } k'' = k'[j] \text{ in out}(c_B, ()) \]

\[ Z(k2b(k')) = Z_k \]
Proof of the example: secrecy

\[ Q_0 = \text{in}(\text{start}, ()); \text{new } r : \text{keyseed}; \text{new } r' : \text{mkeyseed}; \text{out}(c, ()); \]
\[ (Q_A \parallel Q_B) \]

\[ Q_A = \lnot i \leq n \text{ in}(c_A, ()); \text{new } k' : \text{key}; \text{new } r'' : \text{coins}; \]
\[ \text{let } m : \text{bitstring} = \text{enc}'(Z_k, kgen'(r), r'') \text{ in} \]
\[ \text{out}(c_A, (m, \text{mac'}(m, mkgen'(r'))))) \]

\[ Q_B = \lnot i' \leq n \text{ in}(c_B, (m' : \text{bitstring}, ma : \text{macstring})); \]
\[ \text{find } j \leq n \text{ such that defined}(m[j]) \land m' = m[j] \land \]
\[ \text{verify}'(m', mkgen'(r'), ma) \text{ then} \]
\[ \text{let } k'' = k'[j] \text{ in out}(c_B, ()) \]

Preserves the one-session secrecy of \( k'' \) but not its secrecy.
Adding the probabilities, we obtain:

\[
\text{Result}
\]

The probability that an adversary that runs in time at most \( t \), that executes \( n \) sessions of \( A \) and \( B \) and sends messages of length at most \( l_{mB} \) to \( B \) breaks the one-session secrecy of \( k'' \) is

\[
2\text{Succ}_{\text{MAC}}^{\text{uf-cma}}(t'_1, n, n, \max(l_{mB}, l_c)) + 2\text{Succ}_{\text{SE}}^{\text{ind-cpa}}(t'_2, n, l_k)
\]

where

\[
t'_1 = t + \text{time}(kgen) + n \text{ time}(enc, l_k) + n \text{ time}(dec, l_{mB})
\]

\[
t'_2 = t + (n + n^2) \text{ time}(mkgen) + n \text{ time}(mac, l_c) + n^2 \text{ time}(\text{verify}, l_{mB}) + n^2 \text{ time}(= \text{bitstring}, l_{mB}, l_c)
\]

\( l_k \) is the length of keys, \( l_c \) the length of encryptions of keys.

The factor 2 comes from the definition of secrecy.
Exercise 5: preliminary definition

Definition (IND-CCA2 symmetric encryption)

A symmetric encryption scheme SE is indistinguishable under adaptive chosen-ciphertext attacks (IND-CCA2) if and only if

$$\text{Succ}_{\text{SE}}^{\text{ind-cca2}}(t, q_e, q_d, l_e, l_d)$$

is negligible when $t$ is polynomial in the security parameter:

$$\text{Succ}_{\text{SE}}^{\text{ind-cca2}}(t, q_e, q_d, l_e, l_d) = \max_{A} 2 \Pr \left[ b \stackrel{R}{\leftarrow} \{0, 1\}; \, k \stackrel{R}{\leftarrow} kgen; \atop b' \leftarrow A^{\text{enc}(LR(., ., b), k),\text{dec}(., k)} : b' = b \wedge A \text{ has not called dec}(., k) \text{ on the result of enc}(LR(., ., b), k) \right] - 1$$

where $A$ runs in time at most $t$,
calls $\text{enc}(LR(., ., b), k)$ at most $q_e$ times on messages of length at most $l_e$,
calls $\text{dec}(., k)$ at most $q_d$ times on messages of length at most $l_d$. 
**Exercise 5**

1. Show using CryptoVerif that, if the MAC scheme is SUF-CMA and the encryption scheme is IND-CPA, then the encrypt-then-MAC scheme is IND-CCA2.

2. Show using the same assumptions that the encrypt-then-MAC scheme is INT-CTXT.

3. What happens if the MAC scheme is only UF-CMA?
Example of the FDH signature (joint work with D. Pointcheval)

hash hash function (in the random oracle model)
f(pk, m) one-way trapdoor permutation, with inverse invf(sk, m).

We define a **signature scheme** as follows:

- signature \( \text{sign}(m, sk) = \text{invf}(sk, \text{hash}(hk, m)) \)
- verification \( \text{verify}(m, pk, s) = (f(pk, s) = \text{hash}(hk, m)) \)

Our goal is to show that this signature scheme is UF-CMA (secure against existential forgery under chosen message attacks).
Reminder: UF-CMA signatures

The advantage of the adversary:

$$\max_A \Pr \left[ (pk, sk) \xleftarrow{R} kgen; (m, s) \leftarrow A^{\text{sign}(., sk)}(pk) : \text{verify}(m, pk, s) \land m \text{ was never queried to the oracle } \text{sign}(., sk) \right]$$

is small.
Formalizing the security of a signature scheme (1)

Key generation:

\( \text{in}(start, ()); \textbf{new} \ r : \text{keyseed}; \textbf{let} \ pk = \text{pkgen}(r) \textbf{ in let} \ sk = \text{skgen}(r) \textbf{ in} \textbf{ out}(c0, pk) \)

Chooses a random seed uniformly in the set of bit-strings \textit{keyseed} (consisting of all bit-strings of a certain length), generates a public key \( pk \), a secret key \( sk \), and outputs the public key.
Formalizing the security of a signature scheme (2)

Signature:

\[ \text{in}(c_1, m : \text{bitstring}); \text{out}(c_2, \text{sign}(sk, m)) \]
Formalizing the security of a signature scheme (2)

Signature:

\[ \text{in}(c_1, m : \text{bitstring}); \text{out}(c_2, \text{sign}(sk, m)) \]

This process can be called at most \(q_S\) times:

\[ \forall i \leq q_S \quad \text{in}(c_1, m : \text{bitstring}); \text{out}(c_2, \text{sign}(sk, m)) \]
Formalizing the security of a signature scheme (2)

Signature:

\[ \text{in}(c_1, m : \text{bitstring}); \text{out}(c_2, \text{sign}(sk, m)) \]

This process can be called at most \( q_S \) times:

\[ \forall i_S \leq q_S \text{ in}(c_1, m : \text{bitstring}); \text{out}(c_2, \text{sign}(sk, m)) \]

In fact, this is an abbreviation for:

\[ \forall i_S \leq q_S \text{ in}(c_1, m[i_S] : \text{bitstring}); \text{out}(c_2, \text{sign}(sk, m[i_S])) \]

The variables in repeated oracles are arrays, with one cell for each call, to remember the values used in each oracle call. These arrays are indexed with the call number \( i_S \).
Test:

\[ \text{in}(c3, (m' : \text{bitstring}, s : D)); \text{if verify}(m', pk, s) \text{ then find } j \leq qS \text{ suchthat defined}(m[j]) \land (m' = m[j]) \text{ then yield else event bad) } \]

If \( s \) is a signature for \( m' \) and the signed message \( m' \) is not contained in the array \( m \) of messages passed to signing oracle, then the signature is a forgery, so we execute \text{event bad}. 

Bruno Blanchet (INRIA)
Formalizing the security of a signature scheme (summary)

The signature and test oracles make sense only after the key generation oracle has been called, hence a sequential composition.

The signature and test oracles are simultaneously available, hence a parallel composition.

```plaintext
in(start, ()); new r : keyseed; let pk = pkgen(r) in let sk = skgen(r) in
out(c0, pk);

(* signature oracle *)

!\forall s \leq q_s in(c1, m : bitstring); out(c2, sign(sk, m))

|| (* forged signature? *)

in(c3, (m' : bitstring, s : D)); if verify(m', pk, s) then
find j \leq q_s such that defined(m[j]) \land (m' = m[j])
then yield else event bad)
```

Probability of executing event bad = probability of forging a signature.
Application to the FDH signature scheme

We add a hash oracle because the adversary must be able to call the random oracle (even though it cannot be implemented).

```plaintext
\textbf{in}(\text{start},()) \textbf{new} \ hk : \ hashkey; \ \textbf{new} \ r : \ keyseed;
\textbf{let} \ sk = \text{skgen}(r) \ \textbf{in} \ \textbf{let} \ pk = \text{pkgen}(r) \ \textbf{in} \ \textbf{out}(c0, pk);

(* hash oracle *) \! ^{i_H \leq q_H} \textbf{in}(hc1, x : \text{bitstring}); \ \textbf{out}(hc2, \text{hash}(hk, x))

| (* signature oracle *)
| \! ^{i_S \leq q_S} \textbf{in}(c1, m : \text{bitstring}); \ \textbf{out}(c2, \text{invf}(sk, \text{hash}(hk, m)))
| (* forged signature? *)
| \textbf{in}(c3, (m' : \text{bitstring}, s : D)); \textbf{if} \ f(pk, s) = \text{hash}(hk, m') \ \textbf{then}
| \textbf{find} \ j \leq q_S \ \textbf{suchthat} \ \text{defined}(m[j]) \land (m' = m[j])
| \ \textbf{then} \ \textbf{yield} \ \textbf{else} \ \textbf{event} \ \text{bad})
```

Goal: bound the probability that \textbf{event} bad is executed in this game. This game is given as input to the prover in the syntax above.
FDH: security of a hash function

A hash function is equivalent to a "random function": a function that
- returns a new random number when it is called on a new argument,
- and returns the same result when it is called on the same argument.

\[ \begin{align*}
!^N h \ new \ k : \ hashkey; \ !^N Ohash(x : \ bitstring) & := \ hash(k, x) \\
\approx_0 \\
!^N h \ new \ k : \ hashkey; \ !^N Ohash(x : \ bitstring) & := \\
\text{find } j \leq N \text{ suchthat } \text{defined}(x[j], r[j]) \ & \& \ (x = x[j]) \\
\text{then } r[j] \\
\text{else new } r : D; \ r
\end{align*} \]
FDH: security of a hash function (optimized)

For a test $r' = h(x')$, we can avoid computing $h(x')$ explicitly:

- if $x'$ has been passed to the hash function previously, compare $r'$ with the previous result;
- otherwise, return false.
FDH: security of a hash function (optimized)

For a test $r' = h(x')$, we can avoid computing $h(x')$ explicitly:

- if $x'$ has been passed to the hash function previously, compare $r'$ with the previous result;
- otherwise, return false.

In the latter case, test indeed false, except when the fresh random number $h(x')$ collides with $r'$ (probability $1/|D|$).
FDH: security of a hash function (optimized)

For a test $r' = h(x')$, we can avoid computing $h(x')$ explicitly:

- if $x'$ has been passed to the hash function previously, compare $r'$ with the previous result;
- otherwise, return false.

In the latter case, test indeed false, except when the fresh random number $h(x')$ collides with $r'$ (probability $1/|D|$).

\[ \text{\texttt{new} } k : \text{hashkey}; \]
\[ (\text{\texttt{new} } h_k : \text{hashkey}) := \text{hash}(k, x), \]
\[ (\text{\texttt{new} } \text{Oeq}(x' : \text{bitstring}; r' : D) := r' = \text{hash}(k, x')) \]
\[ \approx \# \text{Oeq} / |D| \]
\[ \text{\texttt{new} } (\text{\texttt{new} } h_k) := \text{find} j \leq N \text{ such that} \]
\[ \text{defined}(x[j], r[j]) \text{ && } (x = x[j]) \text{ then } r[j] \text{ else new } r : D; r, \]
\[ \text{\texttt{new} } \text{Oeq}(x' : \text{bitstring}; r' : D) := \text{find} j \leq N \text{ such that} \]
\[ \text{defined}(x[j], r[j]) \text{ && } (x' = x[j]) \text{ then } r' = r[j] \text{ else false} \]
FDH: one-wayness

The adversary inverts $f$ when, given the public key $pk = \text{pkgen}(r)$ and the image of some $x$ by $f(pk, \cdot)$, it manages to find $x$ (without having the trapdoor).

The function $f$ is one-way when the adversary has negligible probability of inverting $f$.

**Definition (One-wayness)**

$$\text{Succ}^{ow}_{\mathcal{P}}(t) = \max_{\mathcal{A}} \Pr \left[ r \overset{R}{\leftarrow} \text{keyseed}, pk \leftarrow \text{pkgen}(r), x \overset{R}{\leftarrow} D, y \leftarrow f(pk, x), x' \leftarrow \mathcal{A}(pk, y) : x = x' \right]$$

where $\mathcal{A}$ runs in time at most $t$. 
FDH: one-wayness (preliminary version)

\[ !^N_k \textbf{new } r : \text{keyseed}; ( \\
Opk() := \text{pkgen}(r), \\
!^N_f \textbf{new } x : D; ( \\
\quad Oy() := f(\text{pkgen}(r), x), \\
\quad !^N_2 \text{Oeq}(x' : D) := (x' = x))) \]

\[ \approx \]

\[ !^N_k \textbf{new } r : \text{keyseed}; ( \\
Opk() := \text{pkgen}(r), \\
!^N_f \textbf{new } x : D; ( \\
\quad Oy() := f(\text{pkgen}(r), x), \\
\quad !^N_2 \text{Oeq}(x' : D) := \text{false})) \]
FDH: one-wayness

\[\begin{align*}
!^{N_k} & \textbf{new } r : \text{keyseed}; \\
& Opk() := \text{pkgen}(r), \\
!^{N_f} & \textbf{new } x : D; \\
& \quad Oy() := f(\text{pkgen}(r), x), \\
& \quad !^{N_2} \text{Oeq}(x' : D) := (x' = x), \\
& \quad Ox() := x
\end{align*}\]

\[\simeq\]

\[\begin{align*}
!^{N_k} & \textbf{new } r : \text{keyseed}; \\
& Opk() := \text{pkgen}(r), \\
!^{N_f} & \textbf{new } x : D; \\
& \quad Oy() := f(\text{pkgen}(r), x), \\
& \quad !^{N_2} \text{Oeq}(x' : D) := \textbf{if defined}(k) \text{ then } x' = x \text{ else } \text{false}, \\
& \quad Ox() := \textbf{let } k : \text{bitstring} = \text{mark} \text{ in } x
\end{align*}\]
FDH: one-wayness

\[ ![N_k \textbf{new } r : \text{keyseed}; (} \]
\[ Opk() := \text{pkgen}(r), \]
\[ ![N_f \textbf{new } x : D; (} \]
\[ Oy() := f(\text{pkgen}(r), x), \]
\[ ![N^2 Oeq(x' : D) := (x' = x),} \]
\[ Ox() := x)) \]
\[ \approx N_k \times N_f \times \text{Succ}^\text{sw}_P(\text{time} + (N_k - 1) \times \text{time(pkgen)} + (\#O_y - 1) \times \text{time(f)}) \]
\[ ![N_k \textbf{new } r : \text{keyseed}; (} \]
\[ Opk() := \text{pkgen}'(r), \]
\[ ![N_f \textbf{new } x : D; (} \]
\[ Oy() := f'(\text{pkgen}'(r), x), \]
\[ ![N^2 Oeq(x' : D) := \textbf{if defined}(k) \textbf{ then } x' = x \textbf{ else } \text{false,} \]
\[ Ox() := \textbf{let } k : \text{bitstring} = \text{mark } \textbf{in } x)) \]
FDH: other properties of one-way trapdoor permutations (1)

invf is the inverse of f:

\[ \forall r : keyseed, x : D; \text{invf}(skgen(r), f(pkgen(r), x)) = x \]

f is injective:

\[ \forall k : key, x : D, x' : D; (f(k, x) = f(k, x')) = (x = x') \]
We can replace a uniformly distributed random number $y$ with $f(pkgen(r), x)$ where $x$ is a uniformly distributed random number:

$$
\forall^nf\text{ new }y : D; Oim() := y
$$

$$
\approx_0
\forall^nf\text{ new }x : D; Oim() := f(pkgen(r), x)
$$
FDH: other properties of one-way trapdoor permutations (2)

We can replace a uniformly distributed random number $y$ with $f(pkgen(r), x)$ where $x$ is a uniformly distributed random number:

\[
!^{N_f} \textbf{new } y : D; Oim() := y
\]

\[
\approx_0
\]

\[
!^{N_f} \textbf{new } x : D; Oim() := f(pkgen(r), x)
\]

⇒ Can be applied too often!
FDH: other properties of one-way trapdoor permutations (2)

We can replace a uniformly distributed random number $y$ with $f(pkgen(r), x)$ where $x$ is a uniformly distributed random number:

$$\begin{align*}
\forall^N k \new r : \text{keyseed} ; (  \\
\quad \text{Opk}() := \text{pkgen}(r),  \\
\quad \forall^N f \new y : D ; (\text{Oant}() := \text{invf}(skgen(r), y), \text{Oim}() := y))  \\
\approx_0  \\
\forall^N k \new r : \text{keyseed} ; (  \\
\quad \text{Opk}() := \text{pkgen}(r),  \\
\quad \forall^N f \new x : D ; (\text{Oant}() := x, \text{Oim}() := f(pkgen(r), x))
\end{align*}$$
Demo

- CryptoVerif input file: examples/fdh
- library of primitives
- run CryptoVerif
- output
FDH: initial game

\[\text{in}(\text{start}, ()); \ \text{new} \ hk : \text{hashkey}; \ \text{new} \ r : \text{keyseed};\]
\let \text{sk} : \text{key} = \text{skgen}(r) \ \text{in}\]
\let \text{pk} : \text{key} = \text{pkgen}(r) \ \text{in} \ \text{out}(c0, pk);\]
( (* hash oracle *)
\begin{align*}
!^i_{H \leq q_H} \text{in}(hc1[i_H], x : \text{bitstring}); \ \text{out}(hc2[i_H], \text{hash}(hk, x))
\end{align*}
∥ (* signature oracle *)
\begin{align*}
!^{i_S \leq q_S} \text{in}(c1[i_S], m : \text{bitstring}); \ \text{out}(c2[i_S], \text{invf}(sk, \text{hash}(hk, m)))
\end{align*}
∥ (* forged signature? *)
\begin{align*}
\text{in}(c3, (m' : \text{bitstring}, s : D));
\end{align*}
\begin{align*}
\text{if} \ f(pk, s) = \text{hash}(hk, m') \ \text{then} \nonumber
\end{align*}
\begin{align*}
\text{find} \ j \leq q_S \ \text{such that defined}(m[j]) \ \&\& \ (m' = m[j]) \ \text{then} \nonumber
\end{align*}
\begin{align*}
\text{yield} \nonumber
\end{align*}
\begin{align*}
\text{else} \nonumber
\end{align*}
\begin{align*}
\text{event} \ \text{bad} \nonumber
\end{align*}
FDH step 1: apply the security of the hash function

Replace each occurrence of hash($M$) with a lookup in the arguments of previous calls to hash.

- If $M$ is found, return the same result as the previous result.
- Otherwise, pick a new random number and return it.

For instance, $\text{out}(hc_2[i_H], \text{hash}(hk, x))$ is replaced with

$\text{find } @i_1 \leq q_S \text{ such that } \text{defined}(m[@i_1], r_32[@i_1])$

$\&\& (x = m[@i_1]) \text{ then out}(hc_2[i_H], r_32[@i_1])$

$\text{orfind } @i_2 \leq q_H \text{ such that } \text{defined}(x[@i_2], r_34[@i_2])$

$\&\& (x = x[@i_2]) \text{ then out}(hc_2[i_H], r_34[@i_2])$

else

$new \ r_34 : D\; ; \; \text{out}(hc_2[i_H], r_34)$

The test $f(pk, s) = \text{hash}(hk, m')$ uses Oeq. Probability difference $1/|D|$.
FDH step 2: simplify

(* forged signature? *)
\[\text{in}(c_3, (m' : \text{bitstring}, s : D));\]
\[\text{find } @i_5 \leq q_S \text{ such that defined}(m[@i_5], r_{32}@i_5) \&\& (m' = m[@i_5]) \text{ then}\]
\[\text{if } (f(pk, s) = r_{32}@i_5) \text{ then}\]
\[\text{find } j \leq q_S \text{ such that defined}(m[j]) \&\& (m' = m[j]) \text{ then yield else event bad}\]
\[\text{orf\text{find } @i_6 \leq q_H \text{ such that defined}(x[@i_6], r_{34}@i_6) \&\& (m' = x[@i_6]) \text{ then}\]
\[\text{if } (f(pk, s) = r_{34}@i_6) \text{ then}\]
\[\text{find } j \leq q_S \text{ such that defined}(m[j]) \&\& (m' = m[j]) \text{ then yield else event bad}\]
\[\text{else}\]
\[\text{if false then}\]
\[\text{find } j \leq q_S \text{ such that defined}(m[j]) \&\& (m' = m[j]) \text{ then yield else event bad}\]

The red test always succeeds, so the blue part becomes yield.
The magenta part becomes yield.
The variable $sk$ is replaced with $skgen(r)$, and the assignment `let sk : key = skgen(r)` is removed.
This transformation is advised in order to able to apply the permutation property.
FDH step 4: permutation

(* signature oracle *)
\!i ≤ qs

\textbf{in}(c1[i_S], m : \text{bitstring});
\textbf{find} \!i \leq q_S\textbf{ suchthat defined}(m[\!i3], r_32[\!i3]) \&\& (m = m[\!i3]) \textbf{then}
\textbf{out}(c2[i_S], \text{invf}(\text{skgen}(r), r_32[\!i3]))
\textbf{orfind} \!i \leq q_H\textbf{ suchthat defined}(x[\!i4], r_34[\!i4]) \&\& (m = x[\!i4]) \textbf{then}
\textbf{out}(c2[i_S], \text{invf}(\text{skgen}(r), r_34[\!i4]))
\textbf{else}
\textbf{new} r_32 : D;
\textbf{out}(c2[i_S], \text{invf}(\text{skgen}(r), r_32))

\textbf{new} r_i : D \textbf{ becomes new } y_i : D,
\text{invf}(\text{skgen}(r), r_i) \textbf{ becomes } y_i,
\text{} r_i \textbf{ becomes } f(\text{pkgen}(r), y_i)
FDH step 5: simplify

(* forged signature? *)
\[
\text{in}(c3, (m' : \text{bitstring}, s : D));
\]
\[
\text{find } @i5 \leq q_S \text{ such that defined}(m[@i5], r_{32[@i5]}) \&\& (m' = m[@i5]) \text{ then yield}
\]
\[
\text{or find } @i6 \leq q_H \text{ such that defined}(x[@i6], r_{34[@i6]}) \&\& (m' = x[@i6]) \text{ then}
\]
\[
\text{if } (f(pk, s) = f(pkgen(r), y_{34[@i6]})) \text{ then}
\]
\[
\text{find } j \leq q_S \text{ such that defined}(m[j]) \&\& (m' = m[j]) \text{ then yield else event bad}
\]

\[f(pk, s) = f(pkgen(r), y_i)\]

knowing \( pk = pkgen(r) \) and the injectivity of \( f: \)
\[
\forall k : \text{key}, x : D, x' : D; (f(k, x) = f(k, x')) = (x = x')
\]
FDH step 6: one-wayness

(*) forged signature? *)
\[
\text{in}(c_3, (m' : \text{bitstring}, s : D));
\]
\[
\text{find } @i5 \leq q_S \text{ such that } \text{defined}(m[@i5], r_{32}@i5) \&\&(m' = m[@i5]) \text{ then yield}
\]
\[
\text{orf}\text{ind } @i6 \leq q_H \text{ such that } \text{defined}(x[@i6], r_{34}@i6) \&\&(m' = x[@i6]) \text{ then } \text{if } s = y_{34}@i6 \text{ then}
\]
\[
\text{find } j \leq q_S \text{ such that } \text{defined}(m[j]) \&\&(m' = m[j]) \text{ then yield else event bad}
\]
\[
s = y_{-i} \text{ becomes } \text{find } @j_{-i} \leq q_H \text{ such that } \text{defined}(k_{-i}@j_{-i}) \text{ then } s = y_{-i} \text{ else false,}
\]
In hash oracle, \( f(\text{pkgen}(r), y_{-i}) \) becomes \( f'(\text{pkgen'}(r), y_{-i}) \),
In signature oracle, \( y_{-i} \) becomes let \( k_{-i} : \text{bitstring} = \text{mark in } y_{-i} \).
Difference of probability: \( (q_H + q_S)\text{Succ}_P^{ow}(\text{time} + (q_H - 1)\text{time}(f)) \).
FDH step 7: simplify

(* forged signature? *)
\[
\text{in}(c3, (m' : \text{bitstring}, s : D));
\]
\[
\text{find } @i5 \leq q_S \text{ suchthat defined}(m[@i5], r_{32}@i5) \&\&( m' = m[@i5]) \text{ then yield}
\]
\[
\text{orf \text{ind } @i6 \leq q_H \text{ suchthat defined}(x[@i6], r_{34}@i6) \&\&( m' = x[@i6]) \text{ then find } @j_{34} \leq q_S \text{ suchthat defined}(k_{34}@j_{34}) \&\&( @i4@j_{34} = @i6) \text{ then if } s = y_{34}@i6 \text{ then find } j \leq q_S \text{ suchthat defined}(m[j]) \&\&( m' = m[j]) \text{ then yield else event bad}
\]

The test in red always succeeds, so event bad disappears, which proves the desired property.
(* forged signature? *)
\texttt{in}(c3, (m' : bitstring, s : D));

\ldots
\texttt{orfind}\ @i6 \leq q_H\text{ such that defined}(x[@i6], r_34[@i6]) \&\& (m' = x[@i6])\text{ then}

\texttt{find}\ @j_34 \leq q_S\text{ such that defined}(k_34[@j_34]) \&\& (@i4[@j_34] = @i6)\text{ then}

\texttt{if}\ s = y_34[@i6]\text{ then}

\texttt{find}\ j \leq q_S\text{ such that defined}(m[j]) \&\& (m' = m[j])\text{ then yield else event bad}

Definition of \texttt{k}_34:

\texttt{in}(c1[i_S], m : bitstring);

\ldots
\texttt{orfind}\ @i4 \leq q_H\text{ such that defined}(x[@i4], y_34[@i4]) \&\& (m = x[@i4])\text{ then}

\texttt{let}\ k_34 : bitstring = \text{mark in} \ldots

When \texttt{k}_34[@j_34] is defined, \texttt{m}@j_34 is defined and
\texttt{m}@j_34 = x[@i4[@j_34]] = x[@i6] = m'

so the red test succeeds with \texttt{j} = @j_34.
Adding the probabilities, we obtain:

**Result**

The probability that an adversary that runs in time at most $t$ and makes $q_S$ signature queries and $q_H$ hash queries forges a FDH signature is at most

\[
\frac{1}{|D|} + (q_S + q_H)\text{Succ}_P^\text{ow}(t + (q_H - 1)\text{time}(f))
\]
Exercise 6

Suppose that $H$ is a hash function in the Random Oracle Model and that $f$ is a one-way trapdoor permutation. Consider the encryption function $E_{pk}(x) = f_{pk}(r) || H(r) \oplus x$, where $||$ denotes concatenation and $\oplus$ denotes exclusive or (Bellare & Rogaway, CCS’93).

- What is the decryption function?
- Show using CryptoVerif that this public-key encryption scheme is IND-CPA. (IND-CPA is defined like IND-CCA2 except that the adversary does not have access to a decryption oracle.)
Experiments

Tested on the following protocols (original and corrected versions):
- 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

Shared-key encryption is implemented as encrypt-then-MAC, using a IND-CPA encryption scheme.
(For Otway-Rees, we also considered a SPRP encryption scheme,
a IND-CPA + INT-CTXT encryption scheme,
a IND-CCA2 + IND-PTXT encryption scheme.)

Public-key encryption is assumed to be IND-CCA2.
We prove secrecy of session keys and correspondence properties.
In most cases, the prover succeeds in proving the desired properties when they hold, and obviously it always fails to prove them when they do not hold.

Only cases in which the prover fails although the property holds:

- 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.
Results (2)

- Some public-key protocols need **manual proofs**.
  (Give the cryptographic proof steps and single assignment renaming instructions.)

- **Runtime**: 7 ms to 35 s, average: 5 s on a Pentium M 1.8 GHz.
Other case studies

- Full domain hash signature (with David Pointcheval)
  Encryption schemes of Bellare-Rogaway’93 (with David Pointcheval)

- Kerberos V, with and without PKINIT (with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay).

- OEKE (variant of Encrypted Key Exchange, with David Pointcheval).

- A part of an F# implementation of the TLS transport protocol (Microsoft Research and MSR-INRIA).

- SSH Transport Layer Protocol (with David Cadé).
Conclusion

CryptoVerif can automatically prove the security of primitives and protocols.

- The security assumptions are given as indistinguishability properties (proved manually once).
- The protocol or scheme to prove is specified in a process calculus.
- The prover provides a sequence of indistinguishable games that lead to the proof and a bound on the probability of an attack.
- The user is allowed (but does not have) to interact with the prover to make it follow a specific sequence of games.
Future work: CryptoVerif extensions

- Support more **primitives**:
  - More equations, *e.g.* associativity for XOR
  - Primitives with internal state

- Improvements in the **proof strategy**.
  More precise manual hints?

- More **case studies**.
  - Will suggest more extensions.

- **Certify** CryptoVerif; combine it with CertiCrypt.
Future work: grand challenges

- Proof of implementations of protocols in the computational model:
  - by analysis of existing implementations,
  - by generation of implementations from specifications.
- Take into account side-channels.
I warmly thank David Pointcheval for his advice and explanations of the computational proofs of protocols. This project would not have been possible without him.

Work partly supported by the ANR projects FormaCrypt (ARA SSIA 2005) and ProSe (VERSO 2010).