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

Bruno Blanchet

INRIA Paris
Bruno.Blanchet@inria.fr

Year 2022-23
Outline

1. Using CryptoVerif
2. Proof technique: game transformations, proof strategy
3. Two examples:
   - Encrypt-then-MAC
   - FDH
4. Conclusion, future directions
CryptoVerif, http://cryptoverif.inria.fr/

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.
Process calculus for games

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).

\[ M ::= \text{ terms} \]
\[ x, y, z, x[M_1, \ldots, M_n] \quad \text{variable} \]
\[ f(M_1, \ldots, M_n) \quad \text{function application} \]

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

\[ Q ::= \]

\[ 0 \quad \text{end} \]

\[ Q | Q' \quad \text{parallel composition} \]

\[ !i \leq N \ Q \quad \text{replication } N \text{ times} \]

\[ \text{newChannel } c; Q \quad \text{restriction for channels} \]

\[ \text{in}(c,(x_1 : T_1, \ldots, x_m : T_m)); P \quad \text{input} \]

\[ P ::= \]

\[ \text{yield} \quad \text{end} \]

\[ \text{out}(c,(M_1, \ldots, M_m)); Q \quad \text{output} \]

\[ \text{event } e(M_1, \ldots, M_m); P \quad \text{event} \]

\[ \text{new } x : T; P \quad \text{random number generation (uniform)} \]

\[ \text{let } x : T = M \text{ in } P \quad \text{assignment} \]

\[ \text{if } M \text{ then } P \text{ else } P' \quad \text{conditional} \]

\[ \text{find } j \leq N \text{ such that defined}(x[j], \ldots) \wedge M \text{ then } P \text{ else } P' \quad \text{array lookup} \]
Example: 1. symmetric encryption

We consider a probabilistic, length-revealing encryption scheme.

**Definition (Symmetric encryption scheme SE)**

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

$$\text{dec}(\text{enc}(m, k, r'), k) = i_\bot(m)$$

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)

- MAC function $\text{mac}(m, k)$ takes as input a message $m$ and a key $k$.
- Verification function $\text{verify}(m, k, t)$ such that

$$\text{verify}(m, k, \text{mac}(m, k)) = \text{true}.$$  

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.
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

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

\[ Q_0 = \text{in}(\text{start}, ()); \textbf{new} k : \text{key}; \textbf{new} mk : \text{mkey}; \textbf{out}(c, ()); \]
\[ (Q_A(k, mk) | Q_B(k, mk)) \]

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.

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

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

$$Q_A(k, mk) = !^{i \leq n} \text{in}(c_A, ()); \textbf{new} \ k' : \text{key}; \textbf{new} \ r'' : \text{coins};$$

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

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

Role of $A$:

1. $!^{i \leq n}$ 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 $B$

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

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

if verify($e', mk, ma'$) then

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

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

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

$$Q_0 = \text{in}(\text{start}, ()); \textbf{new} k : \text{key}; \textbf{new} mk : \text{mkey}; \textbf{out}(c, ());$$

$$\quad (Q_A(k, mk) \mid Q_B(k, mk))$$

$$Q_A(k, mk) = !^{i \leq n} \text{in}(c_A, ()); \textbf{new} k' : \text{key}; \textbf{new} r'' : \text{coins};$$

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

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

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

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

$$\quad \text{let } i_\perp(k2b(k'')) = \text{dec}(e', k) \textbf{ in} \ \textbf{out}(c_B, ())$$
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 **SUF-CMA** (strongly unforgeable under chosen message attacks).
  An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (not produced by the MAC oracle).
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 **SUF-CMA** (strongly unforgeable under chosen message attacks).
  An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (not produced by the MAC oracle).

- 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 **SUF-CMA** (strongly unforgeable under chosen message attacks).
  
  An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (not produced by the MAC oracle).

- 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: \[ \forall y : \text{key}; \ Z(k2b(y)) = Z_k. \]
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:

\[ Q_A(k, mk) = !i \leq n \text{ in } (c_A, ()); \textbf{new } k'[i] : \text{key}; \textbf{new } r''[i] : \text{coins}; \]

\[ \text{let } e[i] = enc(k2b(k'[i]), k, r''[i]) \text{ in } \]

\[ \textbf{out}(c_A, (e[i], mac(e[i], mk))) \]

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.
find performs an array lookup:

\[ !i \leq N \ldots \text{let } x = M \text{ in } P \]

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

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.

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

written by cryptographers

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

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

Definition (Indistinguishability)

We write $Q \approx^V_p Q'$ 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 $G_0$. 

Bruno Blanchet (INRIA)
A MAC guarantees the integrity and authenticity of the message because only someone who knows the secret key can build the MAC. More formally, \( \text{Succ}_{\text{MAC}}^{\text{suf-cma}}(t, q_m, q_v, l) \) is negligible if \( t \) is polynomial in the security parameter:

\[
\text{Succ}_{\text{MAC}}^{\text{suf-cma}}(t, q_m, q_v, l) = \max_A \Pr \left[ k \xleftarrow{\text{R}} \text{mkey}; (m, s) \leftarrow A^{\text{mac}(., k), \text{verify}(., k, .)} : \text{verify}(m, k, s) \land \text{no query to the oracle mac}(., k) \text{ with message } m \text{ returned } s \right]
\]

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 \( l \).
MAC: intuition behind the CryptoVerif definition

By the previous definition, up to negligible probability,

- the adversary cannot forge a correct MAC

- so, assuming $k \leftarrow mkey$ 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, with result $t$ by the protocol

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

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

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

\[
\text{new } k : mkey; (\quad \\
\quad \text{!}^N Omac(x : \text{bitstring}) := \text{return}(\text{mac}(x, k)) | \quad \\
\quad \text{!}^N' \text{Verify}(m : \text{bitstring}, t : \text{macstring}) := \text{return}(\text{verify}(m, k, t))) \\
\approx \quad \\
\text{new } k : mkey; (\quad \\
\quad \text{!}^N Omac(x : \text{bitstring}) := \text{let } ma = \text{mac}(x, k) \text{ in return}(ma) | \quad \\
\quad \text{!}^N' \text{Verify}(m : \text{bitstring}, t : \text{macstring}) := \quad \\
\quad \text{find } j \leq N \text{ such that defined}(x[j], ma[j]) \land (m = x[j]) \land \quad \\
\quad (t = ma[j]) \text{ then true else false})
\]
MAC: CryptoVerif definition

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

\[
\text{new } k : mkey; ( \\
\quad \text{!}^N \text{Omac}(x : \text{bitstring}) := \text{return}(\text{mac}(x, k)) | \\
\quad \text{!}^N' \text{Verify}(m : \text{bitstring}, t : \text{macstring}) := \text{return}(\text{verify}(m, k, t)))
\]

\[
\approx \text{Succ}^{\text{suf-cma}}_{\text{MAC}}(\text{time}, N, N', \text{max}(\text{maxl}(x), \text{maxl}(m)))
\]

\[
\text{new } k : mkey; ( \\
\quad \text{!}^N \text{Omac}(x : \text{bitstring}) := \text{let } ma = \text{mac}'(x, k) \text{ in return}(ma) | \\
\quad \text{!}^N' \text{Verify}(m : \text{bitstring}, t : \text{macstring}) := \\
\quad \quad \text{find } j \leq N \text{ such that defined}(x[j], ma[j]) \land (m = x[j]) \land \\
\quad \quad (t = ma[j]) \text{ then true else false}
\]

CryptoVerif understands such specifications of primitives. They can be reused in the proof of many protocols.
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 \( mac(x_i, k) \) is replaced with
  \[
  \textbf{let } \text{ma}_i = \text{mac}'(x_i, k) \textbf{ in ma}_i.
  \]

- Each occurrence of *verify* is replaced with a **find** that looks in all arrays \( x_i, ma_i \) 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)

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

where \( \mathcal{A} \) runs in time at most \( t \),
calls \( \text{enc}(LR(\ldots, b, k)) \) 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

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

\[
\textbf{new } k : \text{key}; \forall^N \text{Oenc}(x : \text{bitstring}) := \\
\quad \textbf{new } r' : \text{coins}; \text{return}(\text{enc}(x, k, r'))
\]

\[
\approx 
\textbf{new } k : \text{key}; \forall^N \text{Oenc}(x : \text{bitstring}) := \\
\quad \textbf{new } r' : \text{coins}; \text{return}(\text{enc}(Z(x), k, r'))
\]

\(Z(x)\) 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

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

\textbf{new} \ k : \text{key}; \ !^N O_{\text{enc}}(x : \text{bitstring}) :=
\begin{align*}
\text{new} \ r' : \text{coins}; \ &\text{return}(\text{enc}(x, k, r')) \\
\cong \text{Succ}_{\text{SE}}^{\text{ind-cpa}}(\text{time}, N, \max l(x))
\end{align*}

\textbf{new} \ k : \text{key}; \ !^N O_{\text{enc}}(x : \text{bitstring}) :=
\begin{align*}
\text{new} \ r' : \text{coins}; \ &\text{return}(\text{enc}'(Z(x), k, r'))
\end{align*}

\(Z(x)\) is the bitstring of the same length as \(x\) containing only zeroes (for all \(x : nonce\), \(Z(x) = Z_{nonce}, \ldots\)).
Expansion of assignments: replacing a variable with its value. (Not completely trivial because of array references.)

Example

If $pk$ is defined by

$$\textbf{let } pk = pkgen(r)$$

and there are no array references to $pk$, then $pk$ is replaced with $pkgen(r)$ in the game and the definition of $pk$ 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

\[
\text{in}(\text{start},()); \text{new } k_A : T_k; \text{new } k_B : T_k; \text{out}(c,()); (Q_K \mid Q_S)
\]

\[
Q_K = !^{i \leq n} \text{in}(c, (h : T_h, k : T_k))
\]

if \( h = A \) then let \( k' = k_A \) else

if \( h = B \) then let \( k' = k_B \) else let \( k' = k \)

\[
Q_S = !^{i' \leq n'} \text{in}(c', h' : T_h);
\]

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

\[ \text{in}(\text{start},()); \text{new } k_A : T_k; \text{new } k_B : T_k; \text{out}(c,()); (Q_K \mid Q_S) \]

\[ Q_K = \forall i \leq n \text{ in}(c, (h : T_h, k : T_k)) \]

\[ \text{if } h = A \text{ then let } k'_1 = k_A \text{ else } \]

\[ \text{if } h = B \text{ then let } k'_2 = k_B \text{ else let } k'_3 = k \]

\[ Q_S = \forall i' \leq n' \text{ in}(c', h' : T_h); \]

\[ \text{find } j \leq n \text{ suchthat defined}(h[j], k'_1[j]) \land h' = h[j] \text{ then } P_1(k'_1[j]) \]

\[ \text{orfind } j \leq n \text{ suchthat defined}(h[j], k'_2[j]) \land h' = h[j] \text{ then } P_1(k'_2[j]) \]

\[ \text{orfind } j \leq n \text{ suchthat defined}(h[j], k'_3[j]) \land h' = h[j] \text{ then } P_1(k'_3[j]) \]

\[ \text{else } P_2 \]
Syntactic transformations (3)

**Move new**: move restrictions downwards in the game as much as possible, when there is no array reference to them.

(Moving `new x : T` under a `if` or a `find` duplicates it. A subsequent single assignment renaming will distinguish cases.)

**Example**

```
new x : nonce; if c then P_1 else P_2
```

becomes

```
if c then new x : nonce; P_1 else new x : nonce; P_2
```
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 without array accesses.
Syntactic transformations (5): manual transformations

**Insert an instruction:** insert a test to distinguish cases; insert a variable definition; ...
Preserves the semantics of the game (e.g., the rest of the code is copied in both branches of the inserted test).

**Example**

$P$ becomes

```
if cond then P else P
```

Subsequent transformations can transform $P$ differently, depending on whether $cond$ holds.
Syntactic transformations (6): manual transformations

- **Insert an event**: to apply Shoup’s lemma.
  - A subprocess $P$ becomes **event** $e$.
  - The probability of distinguishing the two games is the probability of executing event $e$. It will be bound by a proof by sequences of games.

- **Replace a term with an equal term**: CryptoVerif verifies that the terms are really equal.
Simplification and elimination of collisions

- CryptoVerif collects equalities that come from:
  - **Assignments**: \( \text{let } x = M \text{ in } P \) implies that \( x = M \) in \( P \)
  - **Tests**: \( \text{if } M = N \text{ then } P \) 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** \( M \text{ then } P_1 \text{ else } P_2 \) simplifies to \( P_1 \).
  - 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 \( G \mid Q_x \) with no public variables that do not contain \( S \),

\[
2 \Pr[C[G \mid Q_x] : S] - 1 \leq p(C),
\]

where

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

\( c_0, c, c', b, b', j, y, \) and \( S \) do not occur in \( G \).
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 $\textbf{new } x[i] : T$ 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 $x$. 
Proof of security properties: secrecy

**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 $i \neq i'$. 
Proof strategy: advice

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

\[ Q_0 = \textbf{in}(\text{start}, ()); \textbf{new } k : \text{key}; \textbf{new } mk : \text{mkey}; \textbf{out}(c, ()); \]
\[ (Q_A(k, mk) \mid Q_B(k, mk)) \]

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

\[ Q_B(k, mk) = !^{i' \leq n} \textbf{in}(c_B, (e' : \text{bitstring}, ma' : \text{macstring})); \]
\[ \text{if } \text{verify}(e', mk, ma') \text{ then} \]
\[ \text{let } i_\bot(k2b(k'')) = \text{dec}(e', k) \textbf{ in } \textbf{out}(c_B, ()) \]
Proof of the example: security of the MAC

\[ Q_0 = \textbf{in}(\text{start}, ()); \textbf{new} \ k : \text{key}; \textbf{new} \ mk : \text{mkey}; \textbf{out}(c, ()); \]
\[ (Q_A(k, mk) | Q_B(k, mk)) \]
\[ Q_A(k, mk) = !^{i \leq n} \textbf{in}(c_A, ()); \textbf{new} \ k' : \text{key}; \textbf{new} \ r'' : \text{coins}; \]
\[ \textbf{let} \ e = \text{enc}(k2b(k'), k, r'') \textbf{ in} \]
\[ \textbf{let} \ ma = \text{mac}'(e, mk) \textbf{ in} \textbf{out}(c_A, (e, ma)) \]
\[ Q_B(k, mk) = !^{i' \leq n} \textbf{in}(c_B, (e' : \text{bitstring}, ma' : \text{macstring})); \]
\[ \textbf{find} \ j \leq n \textbf{ suchthat} \text{defined}(e[j], ma[j]) \wedge e' = e[j] \wedge \]
\[ ma' = ma[j] \textbf{ then} \]
\[ \textbf{let} \ i_\perp(k2b(k'')) = \text{dec}(e', k) \textbf{ in} \textbf{out}(c_B, ()) \]

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

\[ Q_0 = \text{in}(\text{start},()); \text{new } k : \text{key}; \text{new } mk : \text{mkey}; \text{out}(c,()); \]

\[(QA(k, mk) \mid QB(k, mk)) \]

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

\[ QB(k, mk) = !^{i'} \leq n \text{in}(c_B, (e' : \text{bitstring}, ma' : \text{macstring})); \]
\[
\begin{align*}
  & \text{find } j \leq n \text{ such that defined}(e[j], ma[j]) \land e' = e[j] \land \\
  & \quad ma' = ma[j] \text{ then} \\
  & \text{let } k'' = k'[j] \text{ in } \text{out}(c_B,())
\end{align*}
\]

\[ \text{dec}(e', k) = \text{dec}(\text{enc}(k2b(k'[j]), k, r''[j]), k) = i_\perp(k2b(k'[j])) \]
Proof of the example: security of the encryption

\[ Q_0 = \text{in}(\text{start}, ()); \textbf{new} \ k : \text{key}; \textbf{new} \ mk : \text{mkey}; \textbf{out}(c, ()); \]
\[ (Q_A(k, mk) \mid Q_B(k, mk)) \]

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

\[ Q_B(k, mk) = \!^{i'}\leq^n \text{in}(c_B, (e' : \text{bitstring}, ma' : \text{macstring})); \]
\[ \text{find} \ j \leq n \ \text{such that} \ \text{defined}(e[j], ma[j]) \land e' = e[j] \land \]
\[ ma' = ma[j] \ \text{then} \]
\[ \text{let} \ k'' = k'[j] \ \textbf{in} \ \textbf{out}(c_B, ()) \]

Probability: \( \text{Suc}_{\text{ind-cpa}}^\text{SE}(\text{time} + n \text{ time}(\text{mac, maxl(e)}) + n^2 \text{ time}(= \text{bitstring}, \text{maxl(e')}, \text{maxl(e)), n, length(key})) \)
Proof of the example: simplify

\[ Q_0 = \text{in}(\text{start},()); \text{new } k : \text{key}; \text{new } mk : \text{mkey}; \text{out}(c,()); \]
\[ (Q_A(k, mk) \mid Q_B(k, mk)) \]

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

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

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

\[
Q_0 = \text{in}(\text{start},()); \ \text{new} \ k : \text{key}; \ \text{new} \ mk : \text{mkey}; \ \text{out}(c,()); \\
(Q_A(k, mk) | Q_B(k, mk))
\]

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

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

Preserves the one-session secrecy of \(k''\) but not its secrecy.
Final result

Adding the probabilities, we obtain:

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

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

where $t'_1 = t + n \text{time}(\text{enc}, l_k) + n \text{time}(\text{dec}, l_{mB})$

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

$l_k$ is the length of keys, $l_c$ the length of encryptions of keys.

The factor 2 comes from the definition of secrecy.
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).
UF-CMA signatures

The advantage of the adversary:

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

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

Key generation:

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

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 \textit{pk}, a secret key \textit{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:

\[ !^{i_s \leq q_s} \text{in}(c_1, m : \text{bitstring}); \text{out}(c_2, \text{sign}(sk, m)) \]

In fact, this is an abbreviation for:

\[ !^{i_s \leq q_s} \text{in}(c_1, m[i_s] : \text{bitstring}); \text{out}(c_2, \text{sign}(sk, m[i_s])) \]

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:

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

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 \textbf{event bad}. 

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 *)

!is≤qs 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 ≤ qs suchthat defined(m[j]) ∧ (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).

\[
\text{in}(\text{start}, ()) \text{; new } hk : \text{hashkey}; \text{new } r : \text{keyseed};
\]
\[
\text{let } sk = \text{skgen}(r) \text{ in let } pk = \text{pkgen}(r) \text{ in out}(c0, pk);
\]
\[
(\text{!* hash oracle */}) \text{!}^{i_H \leq q_H}\text{in}(hc1, x : \text{bitstring}); \text{ out}(hc2, \text{hash}(hk, x))
\]
\[
| (\text{!* signature oracle */})
\]
\[
\text{!}^{i_S \leq q_S}\text{in}(c1, m : \text{bitstring}); \text{ out}(c2, \text{invf}(sk,\text{hash}(hk, m)))
\]
\[
| (\text{!* forged signature? */})
\]
\[
\text{in}(c3, (m' : \text{bitstring}, s : D)); \text{ if } f(pk, s) = \text{hash}(hk, m') \text{ then find } j \leq q_S \text{ suchthat defined}(m[j]) \land (m' = m[j])
\]
\[
\text{then yield else event bad)
\]

Goal: bound the probability that event bad is executed in this game. This game is given as input to the prover in the syntax above.
A hash function is equivalent to a “random function”: a function that
\begin{itemize}
  \item returns a new random number when it is called on a new argument,
  \item and returns the same result when it is called on the same argument.
\end{itemize}

\begin{verbatim}
new k : hashkey; \!^N Ohash(x : bitstring) := return(hash(k, x))
\end{verbatim}

\begin{verbatim}
\approx_0
\end{verbatim}

\begin{verbatim}
new k : hashkey; \!^N Ohash(x : bitstring) :=
  find j \leq N such that defined(x[j], r[j]) && (x = x[j])
  then return(r[j])
  else new r : D; return(r)
\end{verbatim}
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)

```plaintext
new k : hashkey;

(!^N Ohash(x : bitstring) := return(hash(k, x)) |
 !^Neq Oeq(x' : bitstring; r' : D) := return(r' = hash(k, x'))

≈^Neq/|D|

!^N Ohash(x : bitstring) :=
  find j ≤ N suchthat defined(x[j], r[j]) && (x = x[j])
  then return(r[j])
  else new r : D; return(r) |

!^Neq Oeq(x' : bitstring; r' : D) :=
  find j ≤ N suchthat defined(x[j], r[j]) && (x' = x[j])
  then return(r' = r[j])
  else return(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)**

\[
\text{Succ}^{ow}_P(t) = \max \Pr_{\mathcal{A}} \left[ r \overset{R}{\leftarrow} \text{keyseed}, pk \leftarrow \text{pkgen}(r), x \overset{R}{\leftarrow} D, y \leftarrow f(pk, x), x' \leftarrow \mathcal{A}(pk, y) : x = x' \right]
\]

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

\[ \text{new } r : keyseed; ( \]
\[ \text{Opk()} := \text{return}(\text{pkgen}(r)) \mid \]
\[ !^{|N_f|} \text{new } x : D; ( \]
\[ \text{Oy()} := \text{return}(f(\text{pkgen}(r), x)) \mid \]
\[ !^{|N^2|} \text{Oeq}(x' : D) := \text{return}(x' = x)) \]
\[ \approx \]

\[ \text{new } r : keyseed; ( \]
\[ \text{Opk()} := \text{return}(\text{pkgen}(r)) \mid \]
\[ !^{|N_f|} \text{new } x : D; ( \]
\[ \text{Oy()} := \text{return}(f(\text{pkgen}(r), x)) \mid \]
\[ !^{|N^2|} \text{Oeq}(x' : D) := \text{return}(false)) \]
FDH: one-wayness

\begin{verbatim}
new r : keyseed; (  
  Opk() := return(pkgen(r)) |  
  \text{!}^{Nf} \text{new x : D; (  
    Oy() := return(f(pkgen(r), x)) |  
    \text{!}^{N2} Oeq(x' : D) := return(x' = x) |  
    Ox() := return(x))))

\approx

new r : keyseed; (  
  Opk() := return(pkgen(r)) |  
  \text{!}^{Nf} \text{new x : D; (  
    Oy() := return(f(pkgen(r), x)) |  
    \text{!}^{N2} Oeq(x' : D) :=  
      \text{if defined}(k) \text{ then return}(x' = x) \text{ else return}(false) |  
    Ox() := let k : bool = true in return(x))))
\end{verbatim}
FDH: one-wayness

\[
\begin{align*}
\text{new } r &: \text{keyseed}; \\
\text{Opk}() &:= \text{return}(\text{pkgen}(r)) \\
\end{align*}
\]

\[
\begin{align*}
\text{new } x &: D; \\
\text{Oy}() &:= \text{return}(f(\text{pkgen}(r), x)) \\
\text{Oeq}(x' : D) &:= \text{return}(x' = x) \\
\text{Ox}() &:= \text{return}(x) \\
\end{align*}
\]

\[
\begin{align*}
\approx N_f \times \text{Succ}_P^{\text{OW}}(\text{time} + (\#O_y - 1) \times \text{time}(f))
\end{align*}
\]

\[
\begin{align*}
\text{new } r &: \text{keyseed}; \\
\text{Opk}() &:= \text{return}(\text{pkgen}'(r)) \\
\text{new } x &: D; \\
\text{Oy}() &:= \text{return}(f'(\text{pkgen}'(r), x)) \\
\text{Oeq}(x' : D) := \\
\text{if defined}(k) \text{ then return}(x' = x) \text{ else return}(\text{false}) \\
\text{Ox}() &:= \text{let } k : \text{bool} = \text{true } \text{in return}(x))
\end{align*}
\]
invf is the inverse of f:

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

f is injective:

\[ \forall k : \text{key}, x : D, x' : D; (f(k, x) = f(k, x')) = (x = x') \]
We can replace a uniformly distributed random number \( y \) with 

\[ f(pkgen(r), x) \]

where \( x \) is a uniformly distributed random number:

\[
\begin{align*}
\text{new } y &: D; Oim() := \text{return}(y) \\
\approx_0 \text{new } x &: D; Oim() := \text{return}(f(pkgen(r), x))
\end{align*}
\]
We can replace a uniformly distributed random number $y$ with $f(pkgen(r), x)$ where $x$ is a uniformly distributed random number:

```
new y : D; Oim() := return(y) ≈_0
new x : D; Oim() := return(f(pkgen(r), x))
⇒ Can be applied too often!
```
We can replace a uniformly distributed random number $y$ with $f(pkgen(r), x)$ where $x$ is a uniformly distributed random number:

```plaintext
new r : keyseed; (                      
  Opk() := return(pkgen(r)) |            
  !^Nf new y : D; (Oant() := return(invf(skgen(r), y)) |  
    Oim() := return(y)))                 
≈_0
new r : keyseed; (                      
  Opk() := return(pkgen(r)) |            
  !^Nf new x : D; (Oant() := return(x) |  
    Oim() := return(f(pkgen(r), x))))    
```
Demo

- CryptoVerif input file: fdh.cv
- library of primitives
- run CryptoVerif
- output
FDH: initial game

\[
\text{in}(start, ()); \textbf{new} \ hk : \text{hashkey}; \textbf{new} \ r : \text{keyseed}; \\
\textbf{let} \ sk : \text{key} = \text{skgen}(r) \ \textbf{in} \\
\textbf{let} \ pk : \text{key} = \text{pkgen}(r) \ \textbf{in} \ \textbf{out}(c0, pk); \\
( (* \text{hash oracle} *) \\
!i_H \leq q_H \ \textbf{in}(hc1[i_H], x : \text{bitstring}); \ \textbf{out}(hc2[i_H], \text{hash}(hk, x)) \\
| ( (* \text{signature oracle} *) \\
!i_S \leq q_S \ \textbf{in}(c1[i_S], m : \text{bitstring}); \ \textbf{out}(c2[i_S], \text{invf}(sk, \text{hash}(hk, m))) \\
| ( (* \text{forged signature?} *) \\
\text{in}(c3, (m' : \text{bitstring}, s : D)); \\
\text{if} \ f(pk, s) = \text{hash}(hk, m') \ \text{then} \\
\textbf{find} \ j \leq q_S \ \textbf{suchthat} \ \text{defined}(m[j]) \ \&\& \ (m' = m[j]) \ \text{then} \\
\quad \textbf{yield} \\
\text{else} \\
\quad \textbf{event} \ \text{bad} \\
) \
\]
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, $\text{out}(hc2[i_H], \text{hash}(hk, x))$ is replaced with

```plaintext
\textbf{find} \ i1 \leq q_S \ \textbf{suchthat defined}(m[i_1], r_{-1}[i_1])
&& (x = m[i_1]) \ \textbf{then out}(hc2[i_H], r_{-1}[i_1])
\textbf{orfind} \ i2 \leq q_H \ \textbf{suchthat defined}(x[i_2], r_2[i_2])
&& (x = x[i_2]) \ \textbf{then out}(hc2[i_H], r_2[i_2])
```

else

```plaintext
\textbf{new} \ r_2 \ : \ D; \ \textbf{out}(hc2[i_H], r_2)
```

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

(* forged signature? *)
\[\text{in}(c3, (m' : \text{bitstring}, s : D));\]
\[\text{find } i5 \leq q_S \text{ such that defined}(m[i5], r_1[i5]) \&\& (m' = m[i5]) \text{ then}\]
\[\quad \text{if } (f(pk, s) = r_1[i5]) \text{ then}\]
\[\quad \text{find } j \leq q_S \text{ such that defined}(m[j]) \&\& (m' = m[j]) \text{ then yield else event \text{ bad}}\]
\[\text{or find } i6 \leq q_H \text{ such that defined}(x[i6], r_2[i6]) \&\& (m' = x[i6]) \text{ then}\]
\[\quad \text{if } (f(pk, s) = r_2[i6]) \text{ then}\]
\[\quad \text{find } j \leq q_S \text{ such that defined}(m[j]) \&\& (m' = m[j]) \text{ then yield else event \text{ bad}}\]
\[\text{else}\]
\[\quad \text{if false then}\]
\[\quad \text{find } j \leq q_S \text{ such that defined}(m[j]) \&\& (m' = m[j]) \text{ then yield else event \text{ bad}}\]

The red test always succeeds, so the blue part becomes \text{yield.}

The magenta part becomes \text{yield.}
FDH step 3: substitute $sk$ with its value

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 *)

\( i_S \leq q_S \)

\[
\text{in}(c_1[i_S], m : \text{bitstring});
\]

\[
\text{find } i_3 \leq q_S \text{ such that defined}(m[i_3], r_1[i_3]) \land (m = m[i_3]) \text{ then}
\]

\[
\text{out}(c_2[i_S], \text{invf}(\text{skgen}(r), r_1[i_3]))
\]

\[
\text{or find } i_4 \leq q_H \text{ such that defined}(x[i_4], r_2[i_4]) \land (m = x[i_4]) \text{ then}
\]

\[
\text{out}(c_2[i_S], \text{invf}(\text{skgen}(r), r_2[i_4]))
\]

\[
\text{else}
\]

\[
\text{new } r_1 : D;
\]

\[
\text{out}(c_2[i_S], \text{invf}(\text{skgen}(r), r_1))
\]

\[
\text{new } r_i : D \text{ becomes new } y_i : D,
\]

\[
\text{invf}(\text{skgen}(r), r_i) \text{ becomes } y_i,
\]

\[
r_i \text{ becomes } f(\text{pkgen}(r), y_i)
\]
FDH step 5: simplify

(* forged signature? *)
\[\text{in}(c3, (m' : \text{bitstring}, s : D));\]
\[\text{find } i5 \leq q_S \text{ such that defined}(m[i5], r_1[i5]) \&\& (m' = m[i5]) \text{ then}\]
\[\text{yield}\]
\[\text{or find } i6 \leq q_H \text{ such that defined}(x[i6], r_2[i6]) \&\& (m' = x[i6]) \text{ then}\]
\[\text{if } (f(pk, s) = f(pkgen(r), y_2[i6])) \text{ then}\]
\[\text{find } j \leq q_S \text{ such that defined}(m[j]) \&\& (m' = m[j]) \text{ then yield} \text{ else event bad}\]

\[f(pk, s) = f(pkgen(r), y_i) \text{ becomes } s = y_i,\]
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

\[(\ast \text{ forged signature? } \ast)\]
\[\text{in}(c_3, (m' : \text{bitstring}, s : D));\]
\[\text{find } i_5 \leq q_S \text{ such that defined}(m[i_5], r_1[i_5]) \&\& (m' = m[i_5]) \text{ then yield}\]
\[\text{or find } i_6 \leq q_H \text{ such that defined}(x[i_6], r_2[i_6]) \&\& (m' = x[i_6]) \text{ then if } s = y_2[i_6] \text{ then}
\[\text{find } j \leq q_S \text{ such that defined}(m[j]) \&\& (m' = m[j]) \text{ then yield else event} \text{ bad}\]
\[s = y_i \text{ becomes find } j_i \leq q_H \text{ such that defined}(k_i[j_i]) \]
\[\text{then } s = y_i \text{ else false,}\]
In hash oracle, \(f(pkgen(r), y_i)\) becomes \(f'(pkgen'(r), y_i)\),
In signature oracle, \(y_i\) becomes \text{let } k_i : bool = true in \text{ y}_i.
Difference of probability: \((q_H + q_S)\text{Succ}^{ow}_P(\text{time} + (q_H - 1)\text{time}(f))\).
FDH step 7: simplify

(*) forged signature?*)
\[
\text{in}(c3, (m' : \text{bitstring}, s : D));
\]
\[
\text{find } i5 \leq q_S \text{ such that defined } (m[i5], r_1[i5]) \land (m' = m[i5]) \text{ then yield}
\]
\[
\text{orf find } i6 \leq q_H \text{ such that defined } (x[i6], r_2[i6]) \land (m' = x[i6]) \text{ then}
\]
\[
\text{find } j_2 \leq q_S \text{ such that defined } (k_2[j_2]) \land (i4[j_2] = i6) \text{ then}
\]
\[
\text{if } s = y_2[i6] \text{ then}
\]
\[
\text{find } j \leq q_S \text{ such that defined } (m[j]) \land (m' = m[j]) \text{ then yield else event bad}
\]

The test in red always succeeds, so event bad disappears, which proves the desired property.
FDH step 7: simplify (2)

(* forged signature? *)
\[ \text{in}(c3, (m' : \text{bitstring}, s : D)); \]

\[ \ldots \]
\[ \text{orfind } i6 \leq q_H \text{ suchthat defined}(x[i6], r_2[i6]) \&\& (m' = x[i6]) \text{ then} \]
\[ \text{find } j_2 \leq q_S \text{ suchthat defined}(k_2[j_2]) \&\& (i4[j_2] = i6) \text{ then} \]
\[ \text{if } s = y_2[i6] \text{ then} \]
\[ \text{find } j \leq q_S \text{ suchthat defined}(m[j]) \&\& (m' = m[j]) \text{ then yield else event bad} \]

Definition of \( k_2 \):
\[ !i_S \leq q_S \]
\[ \text{in}(c1[i_S], m : \text{bitstring}); \]

\[ \ldots \]
\[ \text{orfind } i4 \leq q_H \text{ suchthat defined}(x[i4], y_2[i4]) \&\& (m = x[i4]) \text{ then} \]
\[ \text{let } k_2 : \text{bool} = \text{true in} \ldots \]

When \( k_2[j_2] \) is defined, \( m[j_2] \) is defined and \( m[j_2] = x[i4[j_2]] = x[i6] = m' \)
so the red test succeeds with \( j = j_2 \).
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

$$\frac{1}{|D|} + (q_S + q_H) Succ^o_P(t + (q_H - 1)\text{time}(f))$$
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 assumed to be IND-CPA and INT-CTXT (authenticated 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 case in which the prover fails although the property holds:

- Needham-Schroeder public-key when the exchanged key is the nonce $N_A$. 
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)
- A part of an F# implementation of the TLS transport protocol (Microsoft Research and MSR-INRIA)
- SSH Transport Layer Protocol (with David Cadé)
- Avionics protocols (ARINC 823, ICAO9880 3rd edition)
- TextSecure v3 (with Nadim Kobeissi and Karthikeyan Bhargavan)
- TLS 1.3 draft 18 (with Karthikeyan Bhargavan and Nadim Kobeissi)
- WireGuard (with Benjamin Lipp and Karthikeyan Bhargavan)
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.
Recent and future work: CryptoVerif extensions

- Support more primitives:
  - More equations, *e.g.* associativity for XOR [Done]
  - Primitives with internal state
- Improvements in the proof strategy.
  More precise manual hints?
- More case studies.
  - Will suggest more extensions.
- Combine CryptoVerif with EasyCrypt.
- Certify CryptoVerif.
Proof of implementations of protocols in the computational model:
- by analysis of implementations (e.g., FS2CV, F*, C to CryptoVerif),
- by generation of implementations from specifications (e.g., CryptoVerif to OCaml).

Take into account side-channels.
Acknowledgments

- 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), ProSe (VERSO 2010), and TECAP (2017).