CryptoVerif: Mechanizing Game-Based Proofs

Bruno Blanchet

INRIA Paris
Bruno.Blanchet@inria.fr

December 2020
Outline

1. Introduction
2. Example: Encrypt-then-MAC
3. Encrypt-then-MAC is IND-CPA
4. Encrypt-then-MAC is INT-CTXT
5. Conclusion, future directions
CryptoVerif, http://crypto-verif.inria.fr/

CryptoVerif is a mechanized prover that works in the computational model of cryptography (the model typically used by cryptographers):

- Messages are bitstrings.
- Cryptographic primitives are functions from bitstrings to bitstrings.
- The adversary is a probabilistic Turing machine.
CryptoVerif

- generates **proofs by sequences of games**.
- proves **secrecy, authentication, and indistinguishability** 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).
- has an **automatic** proof strategy and can also be **manually guided**.
Proofs by sequences of games

Proofs in the computational model are typically proofs by sequences of games [Shoup, Bellare & Rogaway]:

- The first game is the real protocol.
- One goes from one game to the next by syntactic transformations or by applying the definition of security of a cryptographic primitive. The difference of probability between consecutive games is negligible.
- The last game is “ideal”: the security property is obvious from the form of the game. (The advantage of the adversary is 0 for this game.)
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 probabilistic process calculus: a small, specialized programming language.

The processes define the oracles that the adversary can call.

The runtime of processes is bounded:

- bounded number of copies of processes,
- bounded length of messages given as input to oracles.
Process calculus for games: terms

Terms represent computations on messages (bitstrings).

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

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

\[ Q ::= \]
\[ 0 \quad \text{oracle definitions} \]
\[ Q | Q' \quad \text{end} \]
\[ \text{foreach } i \leq N \text{ do } Q \quad \text{parallel composition} \]
\[ O(x_1 : T_1, \ldots, x_m : T_m) ::= P \quad \text{replication } N \text{ times} \]

\[ P ::= \]
\[ \text{yield} \quad \text{oracle body} \]
\[ \text{return} (M_1, \ldots, M_m); Q \quad \text{end} \]
\[ \text{event } e(M_1, \ldots, M_m); P \quad \text{result} \]
\[ x \leftarrow^R T; P \quad \text{event} \]
\[ x : T \leftarrow M; P \quad \text{random number generation (uniform)} \]
\[ \text{if } M \text{ then } P \text{ else } P' \quad \text{assignment} \]
\[ \text{insert } L(M_1, \ldots, M_m); P \quad \text{conditional} \]
\[ \text{get } L(x_1, \ldots, x_m) \quad \text{add an entry to list } L \]
\[ \text{such that } M \text{ in } P \text{ else } P' \quad \text{list lookup} \]
Example: 1. symmetric encryption

We consider a probabilistic, length-revealing encryption scheme.

**Definition (Symmetric encryption scheme SE)**

- (Randomized) encryption function $\text{enc}_r(m, k, r)$ takes as input a message $m$, a key $k$, and random coins $r$.
  We define $\text{enc}(m, k) = r \leftarrow \text{enc\_seed}; \text{enc}_r(m, k, r)$.
- Decryption function $\text{dec}(c, k)$ such that
  \[
  \text{dec}(\text{enc}_r(m, k, r'), k) = \text{injbot}(m)
  \]

The decryption returns a bitstring or bottom:
- bottom when decryption fails,
- the cleartext when decryption succeeds.

The injection injbot maps a bitstring to the same bitstring in
bitstring $\cup \{\text{bottom}\}$.

Bruno Blanchet (INRIA)
Example: 2. MAC

## Definition (Message Authentication Code scheme MAC)

- 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:

\[ enc'(m, (k, mk)) = c1 \| mac(c1, mk) \text{ where } c1 = enc(m, k). \]

\[
\text{letfun } \text{full enc}(m : \text{bitstring}, k : \text{key}, mk : \text{mkey}) = \\
\quad c1 \leftarrow enc(m, k); \\
\quad \text{concat}(c1, \text{mac}(c1, mk)).
\]

\[
\text{letfun } \text{full dec}(c : \text{bitstring}, k : \text{key}, mk : \text{mkey}) = \\
\quad \text{let } \text{concat}(c1, \text{mac1}) = c \text{ in} \\
\quad \text{(if verify}(c1, mk, \text{mac1}) \text{ then dec}(c1, k) \text{ else bottom)} \\
\quad \text{else} \\
\quad \text{bottom.}
\]
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 properties to prove

In the example:

- The encrypt-then-MAC scheme is IND-CPA.
- The encrypt-then-MAC scheme is INT-CTXT.
Example: encrypt-then-MAC 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)

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

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.

We program the IND-CPA experiment in CryptoVerif, for the encrypt-then-MAC scheme.
**IND-CPA: initialization**

\[
O_{\text{start}}() := b \xleftarrow{\mathcal{R}} \text{bool}; k \xleftarrow{\mathcal{R}} \text{key}; mk \xleftarrow{\mathcal{R}} \text{mkey}; \text{return}
\]

Initialization:

1. Define an oracle \(O_{\text{start}}\). (The adversary will call this oracle.)
2. \(O_{\text{start}}\) chooses a random boolean \(b\)
3. Then it generates the key for the encrypt-then-MAC scheme, hence an encryption key and a MAC key.
4. It returns nothing.
IND-CPA: left-or-right encryption oracle

$enc(LR(.,., b), k)$ called at most $q_{Enc}$ times
$L R(x, y, 0) = x$, $LR(x, y, 1) = y$, and $LR(x, y, b)$ is defined only when $x$ and $y$ have the same length.

\[
\text{foreach } i \leq q_{Enc} \text{ do} \\
O_{\text{enc}}(m_1 : \text{bitstring}, m_2 : \text{bitstring}) := \\
\text{if } Z(m_1) = Z(m_2) \text{ then} \\
m_0 \leftarrow \text{if } b \text{ then } m_1 \text{ else } m_2; \\
\text{return}(\text{full}_\text{enc}(m_0, k, mk)).
\]

1. foreach $i \leq q_{Enc}$ do represents $q_{Enc}$ copies, indexed by $i \in [1, q_{Enc}]$. The oracle can be called $q_{Enc}$ times.
2. The oracle takes two messages as input, $m_1$ and $m_2$.
3. It verifies that they have the same length ($Z(m_1) = Z(m_2)$). $Z(x)$ is the bitstring of the same length as $x$ containing only zeroes.
IND-CPA: left-or-right encryption oracle

\[ \text{enc}(\text{LR}(., ., b), k) \text{ called at most } q_{\text{Enc}} \text{ times} \]

\[ \text{LR}(x, y, 0) = x, \text{ LR}(x, y, 1) = y, \text{ and LR}(x, y, b) \text{ is defined only when } x \]

\( \text{and } y \text{ have the same length.} \)

\begin{verbatim}
foreach \( i \leq q_{\text{Enc}} \) do
    \( O_{\text{enc}}(m_1 : \text{bitstring}, m_2 : \text{bitstring}) := \)
    if \( Z(m_1) = Z(m_2) \) then
        \( m_0 \leftarrow \text{if } b \text{ then } m_1 \text{ else } m_2; \)
    return(\( \text{full}\_\text{enc}(m_0, k, mk) \)).
\end{verbatim}

\( m_0 \) is set to \( \text{LR}(m_1, m_2, b) \).

\( \text{The oracle returns the encryption of } m_0. \)
Example: summary of the initial game

\[ O_{\text{start}}() := b \xleftarrow{\text{R}} \text{bool}; k \xleftarrow{\text{R}} \text{key}; mk \xleftarrow{\text{R}} m\text{key}; \text{return}; \]

\[ \text{foreach } i \leq q_{\text{Enc}} \text{ do} \]

\[ O_{\text{enc}}(m_1 : \text{bitstring}, m_2 : \text{bitstring}) := \]

\[ \text{if } Z(m_1) = Z(m_2) \text{ then} \]

\[ m_0 \leftarrow \text{if } b \text{ then } m_1 \text{ else } m_2; \]

\[ \text{return}(\text{full}_\text{enc}(m_0, k, mk)). \]

We prove secrecy of \( b \):

\[ \text{query secret } b \]
Demo

- CryptoVerif input file: enc-then-MAC-IND-CPA.ocv
- run CryptoVerif
- output
Indistinguishability

\[ Q_1 \approx_p Q_2 \]

means that an adversary has at most probability \( p \) of distinguishing the two processes (games) \( Q_1 \) and \( Q_2 \).

(\( p \) is a function of the adversary, more precisely of its runtime and of the numbers of queries it makes to oracles.)

**Lemma**

1. **Reflexivity:** \( Q \approx_0 Q \).
2. **Symmetry:** \( \approx_p \) is symmetric.
3. **Transitivity:** if \( Q \approx_p Q' \) and \( Q' \approx_{p'} Q'' \), then \( Q \approx_{p+p'} Q'' \).
4. **Proof by reduction:** if \( Q \approx_p Q' \) and \( C \) is an adversary that calls oracles of \( Q \) resp. \( Q' \) then \( C[Q] \approx_{p'} C[Q'] \), where \( p'(C') = p(C'[C[]]) \) for any adversary \( C' \).
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 bigger game, using a proof by reduction:

  $$ G_1 \approx_0 C[L] \approx_p' C[R] \approx_0 G_2 $$

- **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 $p + p_1 + \cdots + p_m$ in $G_0$. 
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)

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

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.
IND-CPA symmetric encryption: CryptoVerif definition

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

\[
k \xleftarrow{\text{R}} \text{key}; \text{foreach } i \leq q_e \text{ do } \text{Oenc}(x: \text{bitstring}) := \\
\qquad r' \xleftarrow{\text{R}} \text{enc}_\text{seed}; \text{return}(\text{enc}_r(x, k, r'))
\approx \\
\qquad k \xleftarrow{\text{R}} \text{key}; \text{foreach } i \leq q_e \text{ do } \text{Oenc}(x: \text{bitstring}) := \\
\qquad \qquad r' \xleftarrow{\text{R}} \text{enc}_\text{seed}; \text{return}(\text{enc}_r(\mathcal{Z}(x), k, r'))
\]

\(\mathcal{Z}(x)\) is the bitstring of the same length as \(x\) containing only zeroes.
IND-CPA symmetric encryption: CryptoVerif definition

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

\[
k \overset{R}{\leftarrow} \text{key}; \textbf{foreach } i \leq q_e \textbf{ do } O_{\text{enc}}(x : \text{bitstring}) := \\
   r' \overset{R}{\leftarrow} \text{enc}_\text{seed}; \textbf{return}(\text{enc}_r(x, k, r'))
\]

\[
\approx_{\text{Succ}_{\text{SE}}^{\text{ind}-\text{cpa}}(\text{time}, q_e, \text{maxl}(x))}
\]

\[
k \overset{R}{\leftarrow} \text{key}; \textbf{foreach } i \leq q_e \textbf{ do } O_{\text{enc}}(x : \text{bitstring}) := \\
   r' \overset{R}{\leftarrow} \text{enc}_\text{seed}; \textbf{return}(\text{enc}_r'(Z(x), k, r'))
\]

\(Z(x)\) is the bitstring of the same length as \(x\) containing only zeroes.

CryptoVerif understands such specifications of primitives. They can be reused in the proof of many protocols.
IND-CPA proof: initial game

\[ O_{\text{start}}() \ := \ b \xleftarrow{\text{R}} \ \text{bool}; \ k \xleftarrow{\text{R}} \ \text{key}; \ mk \xleftarrow{\text{R}} \ mkey; \ \text{return}; \]

\text{foreach \ } i \ \leq \ q_{\text{Enc}} \ \text{do}

\[ O_{\text{enc}}(m_1 : \text{bitstring}, m_2 : \text{bitstring}) \ := \]

\text{if } Z(m_1) = Z(m_2) \ \text{then}

\[ m_0 \xleftarrow{\text{if \ } b \ \text{then \ } m_1 \ \text{else \ } m_2; \]

\text{return}((c_1 \xleftarrow{\text{R}} \ \text{enc}\_\text{seed}; \ \text{enc}\_r(m_0, k, r)); \ \text{concat}(c_1, \ \text{mac}(c_1, mk))))

CryptoVerif inlines the definition of \textit{full}\_\textit{enc}.
IND-CPA proof: expand terms into processes

\[ O_{\text{start}}() \coloneqq b \overset{R}{\leftarrow} \text{bool}; k \overset{R}{\leftarrow} \text{key}; mk \overset{R}{\leftarrow} \text{mkey}; \text{return}; \]

\textbf{foreach} \( i \leq q_{\text{Enc}} \) \textbf{do}

\[ O_{\text{enc}}(m_1 : \text{bitstring}, m_2 : \text{bitstring}) \coloneqq \]

\textbf{if} \( Z(m_1) = Z(m_2) \) \textbf{then}

\textbf{if} \( b \) \textbf{then}

\[ r \overset{R}{\leftarrow} \text{enc\_seed}; c_1 \leftarrow \text{enc\_r}(m_1, k, r); \text{return}(\text{concat}(c_1, \text{mac}(c_1, mk))) \]

\textbf{else}

\[ r \overset{R}{\leftarrow} \text{enc\_seed}; c_1 \leftarrow \text{enc\_r}(m_2, k, r); \text{return}(\text{concat}(c_1, \text{mac}(c_1, mk))) \]

Bruno Blanchet (INRIA)
IND-CPA proof: renaming variables

\[
O_{\text{start}}() := b \leftarrow \text{bool}; k \leftarrow \text{key}; mk \leftarrow \text{mkey}; \text{return;}
\]

foreach \( i \leq q_{\text{Enc}} \) do
\[
O_{\text{enc}}(m_1 : \text{bitstring}, m_2 : \text{bitstring}) :=
\]
if \( Z(m_1) = Z(m_2) \) then
\[
\text{if } b \text{ then } \quad r_2 \leftarrow \text{enc}_\text{seed}; c_1 \leftarrow \text{enc}_r(m_1, k, r_2); \text{return(concat(c_1, mac(c_1, mk))))}
\]
else
\[
\text{r_1 \leftarrow enc}_\text{seed}; c_1 \leftarrow \text{enc}_r(m_2, k, r_1); \text{return(concat(c_1, mac(c_1, mk))))}
\]

CryptoVerif renames the two definitions of \( r \) to distinct names.
IND-CPA proof: apply the IND-CPA assumption

\[ O_{\text{start}}() := b \xleftarrow{\$} \text{bool}; k \xleftarrow{\$} \text{key}; mk \xleftarrow{\$} \text{mkey}; \text{return; } \]

\textbf{foreach } i \leq q_{\text{Enc}} \textbf{ do}

\[ O_{\text{enc}}(m_1 : \text{bitstring}, m_2 : \text{bitstring}) := \]

\[ \text{if } Z(m_1) = Z(m_2) \text{ then } \]

\[ \text{if } b \text{ then } \]

\[ r_4 \xleftarrow{\$} \text{enc}_\text{seed}; c_1 \leftarrow \text{enc}_r'(Z(m_1), k, r_4); \text{return}(\text{concat}(c_1, \text{mac}(c_1, mk)) \]

\[ \text{else } \]

\[ r_3 \xleftarrow{\$} \text{enc}_\text{seed}; c_1 \leftarrow \text{enc}_r'(Z(m_2), k, r_3); \text{return}(\text{concat}(c_1, \text{mac}(c_1, mk)) \]

CryptoVerif uses the IND-CPA assumption. It replaces the cleartext messages \((m_1 \text{ and } m_2)\) with bitstrings of the same length containing only zeroes \((Z(m_1), Z(m_2))\).

Probability: \( \text{Succ}^{\text{ind-cpa}}_{\text{SE}}(t', q_{\text{Enc}}, l_m) \) with \( t' = t + q_{\text{Enc}}(\text{time}(=, l_m) + \text{time}(\text{mac}, l_{c_1}) + \text{time}(\text{concat}, l_{c_1}) + 2\text{time}(Z, l_m)) \).
**IND-CPA proof: merge**

\[ O_{\text{start}}() := b \xleftarrow{\text{R}} \text{bool}; k \xleftarrow{\text{R}} \text{key}; mk \xleftarrow{\text{R}} \text{mkey}; \text{return}; \]

\[ \text{foreach } i \leq q_{\text{Enc}} \text{ do} \]

\[ O_{\text{enc}}(m_1 : \text{bitstring}, m_2 : \text{bitstring}) := \]

\[ \text{if } Z(m_1) = Z(m_2) \text{ then} \]

\[ r_3 \xleftarrow{\text{R}} \text{enc}_{\text{seed}}; c_1 \xleftarrow{} \text{enc}_{\text{r}'}(Z(m_2), k, r_3); \text{return}(\text{concat}(c_1, \text{mac}(c_1, mk))) \]

CryptoVerif merges the two branches of the test \textbf{if} \( b \) \textbf{then}, because they execute the same code, knowing that \( Z(m_1) = Z(m_2) \) by the test above. \( b \) is no longer used in the game, hence it is secret.
Final result

Result

The probability that an adversary that runs in time at most \( t \), makes at most \( q_e \) encryption queries of length at most \( l \) breaks the IND-CPA property of encrypt-then-MAC is

\[
2 \text{Succ}_{SE}^{\text{ind-\text{cpa}}} (t', q_e, l)
\]

where
\[
t' = t + q_e (\text{time}(\text{=}, l) + \text{time}(\text{mac}, l') + \text{time}(\text{concat}, l') + 2\text{time}(Z, l))
\]

\( l' \) is the length of ciphertexts for cleartexts of length \( l \).

The factor 2 is added due to the definition of secrecy.
(It could be removed with a different proof.)
Definition (INT-CTX symmetric encryption)

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

$$\operatorname{Succ}_{\text{SE}}^{\text{int-ctxt}}(t, q_e, q_d, l_e, l_d) = \max_A \Pr \left[ k \xleftarrow{\text{R}} \text{key}; c \leftarrow A^{\text{enc}(.,k),\text{dec}(.,k) \neq \bot} : \text{dec}(c, k) \neq \bot \land c \text{ is not the result of a call to the enc(., k) oracle} \right]$$

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

We program the INT-CTX experiment in CryptoVerif, for the encrypt-then-MAC scheme.
INT-CTXT experiment in CryptoVerif

\[ O_{\text{start}}() := k \overset{R}{\leftarrow} \text{key}; mk \overset{R}{\leftarrow} \text{mkey}; \text{return}; \]

\((\text{foreach } i_{\text{enc}} \leq q_{\text{Enc}} \text{ do}) \)
\[ O_{\text{enc}}(m_0 : \text{bitstring}) := \]
\[ c_0 \leftarrow \text{full}_\text{enc}(m_0, k, mk); \text{insert ciphertexts}(c_0); \text{return}(c_0) \]

| \]

\((\text{foreach } i_{\text{dec}} \leq q_{\text{Dec}} \text{ do}) \)
\[ O_{\text{decTest}}(c : \text{bitstring}) := \]
\[ \text{get ciphertexts}(= c) \text{ in return}(true) \text{ else} \]
\[ \text{if } \text{full}_\text{dec}(c, k, mk) \neq \text{bottom} \]
\[ \text{then event } \text{bad}; \text{return}(true) \]
\[ \text{else return}(false)) \)
Demo

- CryptoVerif input file: enc-then-MAC-INT\_CTXT.ocv
- run CryptoVerif
- output
Arrays

A variable defined under a replication is implicitly an **array**:

```latex
\textbf{foreach } ienc \leq q_{\text{Enc}} \textbf{ do}

O_{\text{enc}}(m0[ienc]: \text{bitstring}) := c0[ienc] \leftarrow \text{full\_enc}(m0[ienc], k, mk); \ldots
```

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.
  
  \begin{align*}
  \text{(if } \ldots \text{ then } x \leftarrow M; P \text{ else } x \leftarrow M'; P' \text{ is ok)}
  \end{align*}

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:

\[
\text{foreach } i \leq N \text{ do } \ldots x \leftarrow M; P \\
| O(y : T) := \text{find } j \leq N \text{ suchthat defined}(x[j]) \land y = x[j] \text{ then } \ldots 
\]

Note that `find` is here used outside the scope of `x`.

This is the only way of getting access to values of variables outside their syntactic scope.

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

Lists are converted into arrays:

```plaintext
foreach i ≤ N do . . . insert L(M, M'); P
| O(x': T) := get L(x, y) suchthat x' = x in P'(y)
```

becomes

```plaintext
foreach i ≤ N do . . . x[i] ← M; y[i] ← M'; P
| O(x': T) :=
   find j ≤ N suchthat defined(x[j], y[j]) ∧ x' = x[j] then P'(y[j])
```

Arrays avoid the need for explicit list insertion instructions, which would be hard to guess for an automatic tool.
MAC: definition of security (SUF-CMA)

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

Definition (Strong UnForgeability under Chosen Message Attacks, SUF-CMA)

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

where $\mathcal{A}$ 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

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

Bruno Blanchet (INRIA)
MAC: CryptoVerif definition

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

\[ k \overset{R}{\leftarrow} \text{mkey}; ( \]
\[ \quad \text{foreach } i_m \leq q_m \text{ do } \text{Omac}(x: \text{bitstring}) := \text{return}(\text{mac}(x, k)) | \]
\[ \quad \text{foreach } i_v \leq q_v \text{ do } \text{Overify}(m: \text{bitstring}, t: \text{macstring}) := \]
\[ \quad \quad \text{return}(\text{verify}(m, k, t))) \]
\[ \approx \]

\[ k \overset{R}{\leftarrow} \text{mkey}; ( \]
\[ \quad \text{foreach } i_m \leq q_m \text{ do } \text{Omac}(x: \text{bitstring}) := \text{ma} \leftarrow \text{mac}(x, k); \text{return}(\text{ma}) \]
\[ \quad \text{foreach } i_v \leq q_v \text{ do } \text{Overify}(m: \text{bitstring}, t: \text{macstring}) := \]
\[ \quad \quad \text{find } j \leq N \text{ such that defined}(x[j], \text{ma}[j]) \land m = x[j] \land \]
\[ \quad \quad \quad t = \text{ma}[j] \text{ then true else false} ) \]
MAC: CryptoVerif definition

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

\[
k \xleftarrow{\mathcal{R}} m\text{key}; ( \\
\quad \text{foreach } i_m \leq q_m \text{ do } \text{Omac}(x : \text{bitstring}) := \text{return}(\text{mac}(x, k)) | \\
\quad \text{foreach } i_v \leq q_v \text{ do } \text{Overify}(m : \text{bitstring}, t : \text{macstring}) := \\
\quad \quad \text{return}(\text{verify}(m, k, t)))
\]

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

\[
k \xleftarrow{\mathcal{R}} m\text{key}; ( \\
\quad \text{foreach } i_m \leq q_m \text{ do } \text{Omac}(x : \text{bitstring}) := ma \leftarrow \text{mac}'(x, k); \text{return}(ma) | \\
\quad \text{foreach } i_v \leq q_v \text{ do } \text{Overify}(m : \text{bitstring}, t : \text{macstring}) := \\
\quad \quad \text{find } j \leq N \text{ such that defined}(x[j], ma[j]) \land m = x[j] \land \\
\quad \quad \quad t = ma[j] \text{ then true else false})
\]
MAC: using the CryptoVerif definition

CryptoVerif applies the previous rule automatically in any game, perhaps containing several occurrences of \( mac(\cdot, k) \) and of \( verify(\cdot, k, \cdot) \), provided the key \( k \) is used only for \( mac \) and \( verify \):

- Each occurrence of \( mac(x_i, k) \) is replaced with \( ma_i \leftarrow mac'(x_i, k); ma_i \).

- Each occurrence of \( verify(\cdot, k, \cdot) \) is replaced with a find that looks in all arrays \( x_i, ma_i \) of computed MACs (one array for each occurrence of function \( mac \)).
INT-CTXT proof: initial game

\[ O_{\text{start}}() := k \overset{R}{\leftarrow} \text{key}; mk \overset{R}{\leftarrow} \text{mkey}; \text{return}; \]

\[
\text{(foreach } i_{\text{enc}} \leq q_{\text{Enc}} \text{ do } O_{\text{enc}}(m_0 : \text{bitstring}) :=
\]
\[
c_0 \leftarrow (c_1 \leftarrow (r \overset{R}{\leftarrow} \text{enc\_seed}; \text{enc\_r}(m_0, k, r)); \text{concat}(c_1, \text{mac}(c_1, mk)));
\]
\[
\text{insert ciphertexts}(c_0); \text{return}(c_0))
\]

\[
| \text{(foreach } i_{\text{dec}} \leq q_{\text{Dec}} \text{ do } O_{\text{dec\_Test}}(c : \text{bitstring}) :=
\]
\[
\text{get ciphertexts}(= c) \text{ in return}(true) \text{ else}
\]
\[
\text{if (let concat}(c_2, \text{mac}_1) = c \text{ in}
\]
\[
\text{if verify}(c_2, mk, \text{mac}_1) \text{ then } \text{dec}(c_2, k) \text{ else } \text{bottom}
\]
\[
\text{else } \text{bottom} \neq \text{bottom}
\]
\[
\text{then event } \text{bad}; \text{return}(true)
\]
\[
\text{else return}(false))
\]

CryptoVerif inlines \textit{full\_enc} and \textit{full\_dec}.
INT-CTX proof: encode \textbf{insert} and \textbf{get}

\[ O_{\text{start}}() := k \xleftarrow{R} \text{key}; \ mk \xleftarrow{R} \text{mkey}; \ \text{return}; \]

\[
((\text{foreach } i_{\text{enc}} \leq q_{\text{Enc}} \text{ do } \text{O}_{\text{enc}}(m_0 : \text{bitstring}) :=

\begin{align*}
    c_0 &\leftarrow (c_1 \leftarrow (r \xleftarrow{R} \text{enc}_{\text{seed}}; \text{enc}_r(m_0, k, r)); \text{concat}(c_1, \text{mac}(c_1, mk))); \\
    \text{ciphertexts}_1 &\leftarrow c_0; \ \text{return}(c_0)) \\
\end{align*}
\]

| ((\text{foreach } i_{\text{dec}} \leq q_{\text{Dec}} \text{ do } \text{O}_{\text{dec Test}}(c : \text{bitstring}) :=

\begin{align*}
    \text{find } u \leq q_{\text{Enc}} \text{ such that } &\text{defined}(\text{ciphertexts}_1[u]) \land \text{ciphertexts}_1[u] = c \\
    \text{then return}(true) \\
    \text{else if } (\text{let } \text{concat}(c_2, \text{mac}_1) = c \text{ in} \\
    \begin{align*}
    \text{if } &\text{verify}(c_2, mk, \text{mac}_1) \text{ then } \text{dec}(c_2, k) \text{ else } \text{bottom} \\
    \text{else } \text{bottom} \neq \text{bottom} \\
    \text{then event } \text{bad}; \ \text{return}(true) \\
    \text{else return}(false))\end{align*}
\end{align*}
\)

Bruno Blanchet (INRIA)
**INT-CTXT proof: expand terms into processes**

\[
O_{\text{start}}() := k \xleftarrow{\scriptsize{\text{R}}} \text{key}; mk \xleftarrow{\scriptsize{\text{R}}} mkey; \text{return};
\]

\[
((\text{foreach } i_{\text{enc}} \leq q_{\text{Enc}} \text{ do } O_{\text{enc}}(m_0 : \text{bitstring}) :=
\]
\[
r \xleftarrow{\scriptsize{\text{R}}} \text{enc}_\text{seed}; c_1 \leftarrow \text{enc}_r(m_0, k, r); c_0 \leftarrow \text{concat}(c_1, \text{mac}(c_1, mk)));
\]
\[
\text{return}(c_0))
\]

\[
| (\text{foreach } i_{\text{dec}} \leq q_{\text{Dec}} \text{ do } O_{\text{decTest}}(c : \text{bitstring}) :=
\]
\[
\text{find } u \leq q_{\text{Enc}} \text{ such that defined}(c_0[u]) \land c_0[u] = c
\]
\[
\text{then return}(\text{true})
\]

\[
\text{else let } \text{concat}(c_2, \text{mac}_1) = c \text{ in}
\]
\[
\text{if verify}(c_2, mk, \text{mac}_1) \text{ then}
\]
\[
\text{if } \text{dec}(c_2, k) \neq \text{bottom} \text{ then event } \text{bad}; \text{return}(\text{true})
\]
\[
\text{else return}(\text{false})
\]
\[
\text{else return}(\text{false})
\]

\[
\text{else return}(\text{false}))
\]
INT-CTXT proof: apply SUF-CMA MAC

\[
O_{\text{start}}() := k \overset{R}{\leftarrow} \text{key}; mk \overset{R}{\leftarrow} \text{mkey}; \text{return;}
\]

\[
((\text{foreach } i_{\text{enc}} \leq q_{\text{Enc}} \text{ do } O_{\text{enc}}(m_0 : \text{bitstring}) :=
\]
\[
r \overset{R}{\leftarrow} \text{enc\_seed}; c_1 \leftarrow \text{enc\_r}(m_0, k, r);
\]
\[
c_0 \leftarrow \text{concat}(c_1, (ma_2 \leftarrow \text{mac'}(c_1, mk); ma_2)); \text{return}(c_0))
\]
\]
\[
| (\text{foreach } i_{\text{dec}} \leq q_{\text{Dec}} \text{ do } O_{\text{decTest}}(c : \text{bitstring}) :=
\]
\[
\text{find } u \leq q_{\text{Enc}} \text{ such that } \text{defined}(c_0[u]) \wedge c_0[u] = c
\]
\[
\text{then return}(\text{true})
\]
\[
\text{else let } \text{concat}(c_2, \text{mac1}) = c \text{ in}
\]
\[
\text{if } (\text{find } r_i \leq q_{\text{Enc}} \text{ such that } \text{defined}(c_1[r_i], ma_2[r_i]) \wedge c_2 = c_1[r_i] \wedge
\]
\[
\text{mac1} = ma_2[r_i] \text{ then true else false} \text{ then}
\]
\[
\text{if } \text{dec}(c_2, k) \neq \text{bottom} \text{ then event bad; return}(\text{true})
\]
\[
\text{else return}(\text{false})
\]
\[
\text{else return}(\text{false})\text{))}
\]
**INT-CTX** proof: expand terms into processes; simplify

\[
O_{\text{start}}() := k \leftarrow \text{key}; mk \leftarrow \text{mkey}; \text{return};
\]

\[
((\text{foreach } i_{\text{enc}} \leq q_{\text{Enc}} \text{ do } O_{\text{enc}}(m_0 : \text{bitstring}) :=
\begin{align*}
   r &\leftarrow \text{enc}_\text{seed}; c_1 \leftarrow \text{enc}_r(m_0, k, r); \\
   ma_2 &\leftarrow \text{mac}'(c_1, mk); c_0 \leftarrow \text{concat}(c_1, ma_2); \text{return}(c_0)) \\
| (\text{foreach } i_{\text{dec}} \leq q_{\text{Dec}} \text{ do } O_{\text{decTest}}(c : \text{bitstring}) :=
\begin{align*}
   &\text{find } u \leq q_{\text{Enc}} \text{ suchthat defined}(c_0[u]) \wedge c_0[u] = c \\
   \text{then return}(\text{true}) \\
   \text{else let } \text{concat}(c_2, ma_1) = c \text{ in}
   \begin{align*}
   &\text{find } ri \leq q_{\text{Enc}} \text{ suchthat defined}(c_1[ri], ma_2[ri]) \wedge c_2 = c_1[ri] \wedge \\
   &ma_1 = ma_2[ri] \text{ then}
   \\
   &\text{event bad; return}(\text{true})
   \end{align*}
   \text{else return}(\text{false})
   \end{align*}
\end{align*}
)\)
\]

Bruno Blanchet (INRIA)
INT-CTXT proof: simplify

\[ O_{\text{start}}() := k \xleftarrow{\text{R}} \text{key}; \ mk \xleftarrow{\text{R}} \text{mkey}; \ \text{return}; \]

\[
\left( \left( \text{foreach } i_{\text{enc}} \leq q_{\text{Enc}} \ \text{do } O_{\text{enc}}(m_0 : \text{bitstring}) :=
\right.
\]

\[
\left. r \xleftarrow{\text{R}} \text{enc\_seed}; \ c_1 \leftarrow \text{enc\_r}(m_0, k, r);
\right.
\]

\[
ma_2 \leftarrow \text{mac'}(c_1, \ mk); \ c_0 \leftarrow \text{concat}(c_1, \ ma_2); \ \text{return}(c_0) \right)
\]

| \left( \text{foreach } i_{\text{dec}} \leq q_{\text{Dec}} \ \text{do } O_{\text{dec\_Test}}(c : \text{bitstring}) :=
\right.
\]

\[
\left. \text{find } u \leq q_{\text{Enc}} \ \text{suchthat defined}(c_0[u]) \wedge c_0[u] = c \text{ then return}(true)\right.
\]

\[
\text{else let } \text{concat}(c_2, \ \text{mac1}) = c \ \text{in return}(false)\right.
\]

\[
\text{else return}(false)\right))
\]

When the first \textbf{find} fails, the second \textbf{find} also fails, so it is removed.

Event \textit{bad} no longer occurs: the proof succeeds.
Final result

**Result**

The probability that an adversary that runs in time at most $t$, makes at most $q_e$ encryption queries and $q_d$ decryption queries breaks the INT-CTX property of encrypt-then-MAC is at most

$$\text{Succ}^{\text{suf-cma}}_{\text{MAC}}(t', q_e, q_d, l')$$

where

$t' = t + q_e \text{time}(\text{enc}_r, l) + q_e \text{time}(\text{concat}, l') + q_d q_e \text{time}(=, l'') + q_d \text{time}(\text{let concatenation}, l') + q_d \text{time}(\text{dec}, l')$

$l$ is the maximum length of cleartexts

$l'$ is the maximum length of ciphertexts

$l''$ is the maximum length of ciphertexts with MACs
First experiments

Tested on the following toy 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 authentication.
In most cases, **CryptoVerif succeeds** in proving the desired properties when they hold. Only exception: Needham-Schroeder public-key when the exchanged key is the nonce $N_A$.

Obviously CryptoVerif always fails to prove properties that do not hold.

Some public-key protocols need **manual guidance**. (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.
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)
- HPKE (with Joël Alwen, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel)
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.
Current and future work

- Improve and generalize some game transformations.
- Combine CryptoVerif with EasyCrypt:
  - E.g., prove properties of primitives in EasyCrypt, and use them to prove protocols in CryptoVerif.
- Prove implementations of protocols in the computational model:
  - CryptoVerif can already generate implementations in OCaml.
  - Extend it to generate implementations in F⋆ (proved security properties can be translated as well; further proofs can be done on the generated F⋆ code)
- Improve support for state:
  - Loops with mutable state;
  - Primitives with internal state.
Additional material
## Alternative syntax

<table>
<thead>
<tr>
<th>Shown syntax</th>
<th>Alternative syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>foreach</strong> $i \leq n$ do</td>
<td><strong>!i \leq n</strong></td>
</tr>
<tr>
<td><strong>foreach</strong> $i \leq n$ do</td>
<td><strong>!n (when $i$ is not used)</strong></td>
</tr>
<tr>
<td>$x \leftarrow T; P$</td>
<td><strong>new $x : T; P$</strong></td>
</tr>
<tr>
<td>$x \leftarrow M; P$</td>
<td><strong>let $x = M$ in $P$</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Oracles front-end</th>
<th>Channels front-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(x_1 : T_1, \ldots, x_m : T_m) := P$</td>
<td>$\text{in}(c, x : T); P$</td>
</tr>
<tr>
<td>$\text{return}(M_1, \ldots, M_m); Q$</td>
<td>$\text{out}(c, M); Q$</td>
</tr>
</tbody>
</table>
Syntactic transformations (1)

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

Example

If $pk$ is defined by

$$pk \leftarrow pkgen(r)$$

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**

\[
O_{\text{start}}() := k_A \leftarrow^R T_k; \quad k_B \leftarrow^R T_k; \quad \text{return;} \quad (Q_K \mid Q_S)
\]

\[
Q_K = \text{foreach } i \leq n \text{ do } O_K(h : T_h, k : T_k) :=
\begin{align*}
&\quad \text{if } h = A \text{ then } k' \leftarrow k_A \text{ else } \\
&\quad \text{if } h = B \text{ then } k' \leftarrow k_B \text{ else } k' \leftarrow k
\end{align*}
\]

\[
Q_S = \text{foreach } i' \leq n' \text{ do } O_S(h' : T_h) :=
\begin{align*}
&\quad \text{find } j \leq n \text{ suchthat defined}(h[j], k'[j]) \land h' = h[j] \text{ then } P_1(k'[j]) \\
&\quad \text{else } P_2
\end{align*}
\]
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**

\[ O_{\text{start}}() := k_A \xleftarrow{R} T_k; k_B \xleftarrow{R} T_k; \text{return}; (Q_K \mid Q_S) \]

\[ Q_K = \textbf{foreach} \ i \leq n \ \textbf{do} \ O_K(h : T_h, k : T_k) := \]

\[ \quad \text{if} \ h = A \ \text{then} \ k_1' \leftarrow k_A \ \text{else} \]

\[ \quad \text{if} \ h = B \ \text{then} \ k_2' \leftarrow k_B \ \text{else} \ k_3' \leftarrow k \]

\[ Q_S = \textbf{foreach} \ i' \leq n' \ \textbf{do} \ O_S(h' : T_h) := \]

\[ \quad \text{find} \ j \leq n \ \text{suchthat} \ \text{defined}(h[j], k_1'[j]) \land h' = h[j] \ \text{then} \ P_1(k_1'[j]) \]

\[ \quad \text{orfind} \ j \leq n \ \text{suchthat} \ \text{defined}(h[j], k_2'[j]) \land h' = h[j] \ \text{then} \ P_1(k_2'[j]) \]

\[ \quad \text{orfind} \ j \leq n \ \text{suchthat} \ \text{defined}(h[j], k_3'[j]) \land h' = h[j] \ \text{then} \ P_1(k_3'[j]) \]

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

**Move new:** move restrictions downwards in the game as much as possible, when there is no array reference to them.
(Moving $x \overset{R}{\leftarrow} T$ under a `if` or a `find` duplicates it. A subsequent single assignment renaming will distinguish cases.)

**Example**

$x \overset{R}{\leftarrow} nonce; \text{if } c \text{ then } P_1 \text{ else } P_2$

becomes

```
if c then x \overset{R}{\leftarrow} nonce; P_1 else x \overset{R}{\leftarrow} 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

```plaintext
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:** \( x \leftarrow M; P \) 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** \( M \) **then** \( P_1 \) **else** \( P_2 \) simplifies to \( P_1 \).
  - If a condition of **find** simplifies to **false**, then the corresponding branch is removed.
Security properties

- **Secrecy**: the adversary cannot distinguish the secrets from independent random numbers with several test queries.
- **Correspondence**: $\text{event}(e_1(x)) \Rightarrow \text{event}(e_2(x))$ means that, if $e_1(x)$ has been executed, then $e_2(x)$ has been executed.
- **Injective correspondence**: $\text{inj-event}(e_1(x)) \Rightarrow \text{inj-event}(e_2(x))$ means that each execution of $e_1(x)$ corresponds to a distinct execution of $e_2(x)$. 
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.