

CryptoVerif: a computationally-sound security protocol verifier

Bruno Blanchet

INRIA Paris
Bruno.Blanchet@inria.fr

November 2025



The computational model of cryptography

The **computational model** has been developed at the beginning of the 1980's by Goldwasser, Micali, Rivest, Yao, and others.

- Messages are **bitstrings**. 01100100
- Cryptographic primitives are **functions on bitstrings**.
 $enc(011, 100100) = 111$
- The attacker is any **probabilistic polynomial-time Turing machine**.
 - The security assumptions on primitives specify what the attacker **cannot** do.

CryptoVerif, <http://cryptoverif.inria.fr/>

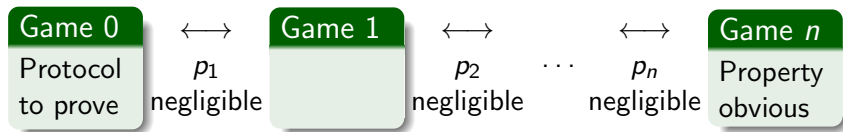
CryptoVerif is a **mechanized prover** that:

- works in the **computational model**.
- generates **proofs by sequences of games**.
- proves **secrecy**, **authentication**, and **indistinguishability** properties.
- provides a **generic** method for specifying properties of **cryptographic primitives**.
- 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** strategy or can be **manually guided**.

Proofs by sequences of games

CryptoVerif produces **proofs by sequences of games**, like those of cryptographers [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 **process calculus**, a small specialized programming language:

- The processes define **oracles** that the adversary can call.
- The semantics is **purely probabilistic** (no non-determinism).
- The runtime of processes is **bounded**:
 - bounded number of copies of processes,
 - bounded length of messages.
- Extension to **arrays**.

Process calculus for games: terms

Terms represent computations on messages (bitstrings).

$M ::=$	terms
$x, y, z, x[M_1, \dots, M_n]$	variable
$f(M_1, \dots, M_n)$	function application

Function symbols f correspond to functions computable by deterministic Turing machines that always terminate.

Process calculus for games: processes

$Q ::=$	oracle definitions
0	end
$Q \parallel Q'$	parallel composition
foreach $i \leq N$ do Q	replication N times
newOracle $O; Q$	restriction for oracles
$O(x_1 : T_1, \dots, x_m : T_m) := P$	oracle declaration
$P ::=$	oracle bodies
yield	end
return (M_1, \dots, M_m); Q	return
event $e(M_1, \dots, M_m); P$	event
$x \stackrel{R}{\leftarrow} T; P$	random number generation (uniform)
$x : T \leftarrow M; P$	assignment
if M then P else P'	conditional
find $j \leq N$ suchthat defined ($x[j], \dots$) $\wedge M$ then P else P'	array lookup

Example: 1. symmetric encryption

We consider a probabilistic, length-revealing encryption scheme.

Definition (Symmetric encryption scheme SE)

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

$$dec(enc(x, k, r'), k) = i_{\perp}(x)$$

The decryption returns a bitstring or \perp :

- \perp when decryption fails,
- the cleartext when decryption succeeds.

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

Example: 1. symmetric encryption

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

- 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** $y : \text{key}; Z(k2b(y)) = Z_k$.

Example: 2. MAC

Definition (Message Authentication Code scheme MAC)

- MAC function $mac(x, k)$ takes as input a message x and a key k .
- Verification function $verify(x, k, m)$ such that

$$verify(x, k, mac(x, k)) = true.$$

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 correct MAC (not produced by the MAC oracle).

Example: 3. encrypt-then-MAC

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

$$enc'(x, (k, mk), r'') = e, mac(e, mk) \text{ where } e = enc(x, k, r'').$$

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.

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

k' should remain secret.

Example: initialization

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

$$Q_0 = O_{\text{start}} := k \stackrel{R}{\leftarrow} \text{key}; mk \stackrel{R}{\leftarrow} \text{mkey}; \mathbf{return}(); \\ (\mathbf{run} \ Q_A(k, mk) \parallel \mathbf{run} \ Q_B(k, mk))$$

Initialization of keys:

- ① The process Q_0 waits for a call to oracle O_{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 **return()**.
 Q_A and Q_B represent the actions of A and B (see next slides).

Example: role of A

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

$$Q_A(k, mk) = \text{foreach } i \leq n \text{ do}$$

$$O_A() := k' \stackrel{R}{\leftarrow} \text{key}; r'' \stackrel{R}{\leftarrow} \text{coins};$$

$$e \leftarrow \text{enc}(k2b(k'), k, r'');$$

$$\text{return}(e, \text{mac}(e, mk))$$

Role of A:

- ① **foreach** $i \leq n$ **do** 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 by a call to oracle O_A by the adversary.
- ③ The process **chooses a fresh key k' and returns the message.**

Example: role of B

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

$Q_B(k, mk) = \text{foreach } i' \leq n \text{ do}$
 $O_B(e' : \text{bitstring}, ma' : \text{macstring}) :=$
if $\text{verify}(e', mk, ma')$ **then**
 $i_{\perp}(k2b(k'')) \leftarrow \text{dec}(e', k); \text{return}()$

Role of B :

- ① n copies, as for Q_A .
- ② The process Q_B waits for a call to oracle O_B , with the message as argument.
- ③ It verifies the MAC, decrypts, and stores the key in k'' .

Example: summary of the initial game

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

$Q_0 = O_{\text{start}} := k \stackrel{R}{\leftarrow} \text{key}; mk \stackrel{R}{\leftarrow} \text{mkey}; \mathbf{return}();$
 $(\mathbf{run} \ Q_A(k, mk) \parallel \mathbf{run} \ Q_B(k, mk))$

$Q_A(k, mk) = \mathbf{foreach} \ i \leq n \ \mathbf{do} \ O_A() := k' \stackrel{R}{\leftarrow} \text{key}; r'' \stackrel{R}{\leftarrow} \text{coins};$
 $e \leftarrow \text{enc}(k2b(k'), k, r'');$
 $\mathbf{return}(e, \text{mac}(e, mk))$

$Q_B(k, mk) = \mathbf{foreach} \ i' \leq n \ \mathbf{do} \ O_B(e' : \text{bitstring}, ma' : \text{macstring}) :=$
 $\mathbf{if} \ \text{verify}(e', mk, ma') \ \mathbf{then}$
 $i_{\perp}(k2b(k'')) \leftarrow \text{dec}(e', k); \mathbf{return}()$

Security properties to prove

In the example:

- **One-session secrecy** of k'' : each k'' is indistinguishable from a random number.
 - Just one test query on k''
- **Secrecy** of k'' : the k'' are indistinguishable from independent random numbers.
 - Several test queries on k''
 - Or one test query and reveal queries on k''

Demo

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

Arrays

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

$$Q_A(k, mk) = \mathbf{foreach} \ i \leq n \ \mathbf{do} \ O_A[i]() := k'[i] \stackrel{R}{\leftarrow} \mathit{key}; r''[i] \stackrel{R}{\leftarrow} \mathit{coins};$$

$$e[i] \leftarrow \mathit{enc}(k2b(k'[i]), k, r''[i]);$$

$$\mathbf{return}(e[i], \mathit{mac}(e[i], mk))$$

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.
(**if** ... **then** $x \leftarrow M; P$ **else** $x \leftarrow M'; P'$ is ok)

So each array cell can be **assigned at most once**.

Arrays allow one to remember the values of all variables during the whole execution

Arrays (continued)

find performs an **array lookup**:

$$\begin{array}{l} \text{foreach } i \leq N \text{ do } \dots x \leftarrow M; P \\ \parallel \text{foreach } i' \leq N' \text{ do } O(y : T) := \\ \quad \text{find } j \leq N \text{ suchthat defined}(x[j]) \wedge y = x[j] \text{ then } \dots \end{array}$$

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 (continued)

find performs an **array lookup**:

$$\begin{array}{l} \text{foreach } i \leq N \text{ do } \dots x[i] \leftarrow M; P \\ \parallel \text{foreach } i' \leq N' \text{ do } O(y : T) := \\ \quad \text{find } j \leq N \text{ suchthat defined}(x[j]) \wedge y = x[j] \text{ then } \dots \end{array}$$

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.

foreach $i \leq N$ **do** ... $x \leftarrow M; y \leftarrow M'; P$

|| foreach $i' \leq N'$ **do** $O(x' : T) :=$

find $j \leq N$ **suchthat** $\text{defined}(x[j], y[j]) \wedge x' = x[j]$ **then** $P'(y[j])$

might be written with lists:

foreach $i \leq N$ **do** ... $x \leftarrow M; y \leftarrow M';$ **insert** $L(x, y); P$

|| foreach $i' \leq N'$ **do** $O(x' : T) :=$ **get** $L(x, y)$ **suchthat** $x' = x$ **in** $P'(y)$

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

Arrays versus lists

Arrays replace **lists** often used in cryptographic proofs.

```

foreach  $i \leq N$  do ...  $x[i] \leftarrow M; y[i] \leftarrow M'; P$ 
|| foreach  $i' \leq N'$  do  $O(x' : T) :=$ 
    find  $j \leq N$  suchthat  $\text{defined}(x[j], y[j]) \wedge x' = x[j]$  then  $P'(y[j])$ 

```

might be written with lists:

```

foreach  $i \leq N$  do ...  $x \leftarrow M; y \leftarrow M'; \text{insert } L(x, y); P$ 
|| foreach  $i' \leq N'$  do  $O(x' : T) := \text{get } L(x, y) \text{ suchthat } x' = x$  in  $P'(y)$ 

```

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

Indistinguishability

Two processes (games) Q , Q' are **indistinguishable** up to probability p when the adversary has probability at most p of distinguishing them:

$$Q \approx_p Q'$$

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 Application of context: if $Q \approx_p Q'$ and C is an evaluation context acceptable for Q and Q' , then $C[Q] \approx_{p'} C[Q']$, where $p'(C', D) = p(C'[C[]], D)$.

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** $G_0 \approx_{p_1} G_1 \approx \dots \approx_{p_m} G_m$, which implies $G_0 \approx_{p_1 + \dots + p_m} G_m$.

If some trace property holds up to probability p in G_m , then it holds up to probability $p + p_1 + \dots + p_m$ in G_0 .

MAC: definition of security (SUF-CMA)

An adversary that has access to the MAC and verification oracles has a negligible probability of forging a correct MAC (not produced by the MAC oracle).

SUF-CMA MAC: towards the CryptoVerif definition

```

 $k \stackrel{R}{\leftarrow} \text{mkey}; ($ 
  foreach  $i \leq N$  do  $O\text{mac}(x : \text{bitstring}) := \text{return}(\text{mac}(x, k)) \parallel$ 
  foreach  $i' \leq N'$  do  $O\text{verify}(x' : \text{bitstring}, m' : \text{macstring}) :=$ 
    return}(\text{verify}(x', k, m')))

```

\approx

```

 $k \stackrel{R}{\leftarrow} \text{mkey}; ($ 
  foreach  $i \leq N$  do  $O\text{mac}(x : \text{bitstring}) :=$ 
     $m \leftarrow \text{mac}(x, k); \text{insert } L(x, m); \text{return}(m) \parallel$ 
  foreach  $i' \leq N'$  do  $O\text{verify}(x' : \text{bitstring}, m' : \text{macstring}) :=$ 
    get }  $L(= x', = m')$  in return}(\text{true}) \text{ else return}(\text{false})

```

MAC: CryptoVerif definition

$verify(x, k, mac(x, k)) = \mathbf{true}$

$k \stackrel{R}{\leftarrow} mkey; ($

foreach $i \leq N$ **do** $Omac(x : bitstring) := \mathbf{return}(mac(x, k)) \parallel$

foreach $i' \leq N'$ **do** $Overify(x' : bitstring, m' : macstring) :=$
 $\mathbf{return}(verify(x', k, m')))$

\approx

$k \stackrel{R}{\leftarrow} mkey; ($

foreach $i \leq N$ **do** $Omac(x : bitstring) :=$

$m \leftarrow mac(x, k); \mathbf{return}(m) \parallel$

foreach $i' \leq N'$ **do** $Overify(x' : bitstring, m' : macstring) :=$

find $j \leq N$ **suchthat** $\mathbf{defined}(x[j], m[j]) \wedge (x' = x[j]) \wedge$
 $(m' = m[j])$ **then return(true) else return(false)**)

MAC: CryptoVerif definition

$verify(x, k, mac(x, k)) = \mathbf{true}$

$k \stackrel{R}{\leftarrow} mkey; ($
foreach $i \leq N$ **do** $Omac(x : bitstring) := \mathbf{return}(mac(x, k)) \parallel$
foreach $i' \leq N'$ **do** $Overify(x' : bitstring, m' : macstring) :=$
 $\mathbf{return}(verify(x', k, m')))$

$\approx Succ_{MAC}^{suf-cma}(\mathbf{time}, N, N', \max(\max l(x), \max l(x')))$

$k \stackrel{R}{\leftarrow} mkey; ($
foreach $i \leq N$ **do** $Omac(x : bitstring) :=$
 $m \leftarrow mac'(x, k); \mathbf{return}(m) \parallel$
foreach $i' \leq N'$ **do** $Overify(x' : bitstring, m' : macstring) :=$
find $j \leq N$ **suchthat** $\mathbf{defined}(x[j], m[j]) \wedge (x' = x[j]) \wedge$
 $(m' = m[j])$ **then return(true) else return(false)**

CryptoVerif understands such specifications of primitives.

They can be reused in the proof of many protocols.

Symmetric encryption (IND-CPA)

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

$$\text{dec}(\text{enc}(x, k, r'), k) = i_{\perp}(x)$$

$$k \stackrel{R}{\leftarrow} \text{key}; \text{foreach } i \leq N \text{ do } O_{\text{enc}}(x : \text{bitstring}) := \\ r' \stackrel{R}{\leftarrow} \text{coins}; \text{return}(\text{enc}(x, k, r'))$$

$$\approx$$

$$k \stackrel{R}{\leftarrow} \text{key}; \text{foreach } i \leq N \text{ do } O_{\text{enc}}(x : \text{bitstring}) := \\ r' \stackrel{R}{\leftarrow} \text{coins}; \text{return}(\text{enc}(Z(x), k, r'))$$

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

Symmetric encryption (IND-CPA)

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

$$\text{dec}(\text{enc}(x, k, r'), k) = i_{\perp}(x)$$

$$k \stackrel{R}{\leftarrow} \text{key}; \textbf{foreach } i \leq N \textbf{ do } O_{\text{enc}}(x : \textit{bitstring}) := \\ r' \stackrel{R}{\leftarrow} \textit{coins}; \textbf{return}(\text{enc}(x, k, r'))$$

$$\approx \text{Succ}_{\text{SE}}^{\text{ind-cpa}}(\text{time}, N, \text{maxl}(x))$$

$$k \stackrel{R}{\leftarrow} \text{key}; \textbf{foreach } i \leq N \textbf{ do } O_{\text{enc}}(x : \textit{bitstring}) := \\ r' \stackrel{R}{\leftarrow} \textit{coins}; \textbf{return}(\text{enc}'(Z(x), k, r'))$$

$Z(x)$ is the bitstring of the same length as x containing only zeroes (for all x : *nonce*, $Z(x) = Z_{\text{nonce}}, \dots$).

Proof strategy: advice

- One tries to execute each transformation given by the definition of a cryptographic primitive.
- When it fails, it 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

$$Q_0 = O_{start} := k \stackrel{R}{\leftarrow} \text{key}; mk \stackrel{R}{\leftarrow} \text{mkey}; \mathbf{return}();$$

$$(\mathbf{run} Q_A(k, mk) \parallel \mathbf{run} Q_B(k, mk))$$

$$Q_A(k, mk) = \mathbf{foreach} i \leq n \mathbf{do} O_A() := k' \stackrel{R}{\leftarrow} \text{key}; r'' \stackrel{R}{\leftarrow} \text{coins};$$

$$e \leftarrow \text{enc}(k2b(k'), k, r'');$$

$$\mathbf{return}(e, \text{mac}(e, mk))$$

$$Q_B(k, mk) = \mathbf{foreach} i' \leq n \mathbf{do} O_B(e' : \text{bitstring}, ma' : \text{macstring}) :=$$

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

$$i_{\perp}(k2b(k'')) \leftarrow \text{dec}(e', k); \mathbf{return}()$$

Proof of the example: security of the MAC

$$Q_0 = O_{start} := k \stackrel{R}{\leftarrow} key; mk \stackrel{R}{\leftarrow} mkey; \mathbf{return}();$$

$$(\mathbf{run} Q_A(k, mk) \parallel \mathbf{run} Q_B(k, mk))$$

$$Q_A(k, mk) = \mathbf{foreach} i \leq n \mathbf{do} O_A() := k' \stackrel{R}{\leftarrow} key; r'' \stackrel{R}{\leftarrow} coins;$$

$$e \leftarrow enc(k2b(k'), k, r'');$$

$$ma \leftarrow mac'(e, mk); \mathbf{return}(e, ma)$$

$$Q_B(k, mk) = \mathbf{foreach} i' \leq n \mathbf{do} O_B(e' : bitstring, ma' : macstring) :=$$

$$\mathbf{find} j \leq n \mathbf{suchthat} \mathbf{defined}(e[j], ma[j]) \wedge e' = e[j] \wedge$$

$$ma' = ma[j] \mathbf{then}$$

$$i_{\perp}(k2b(k'')) \leftarrow dec(e', k); \mathbf{return}()$$

Probability: $\text{Succ}_{\text{MAC}}^{\text{suf-cma}}(\mathbf{time} + n \mathbf{time}(enc, \text{length}(key)) +$
 $n \mathbf{time}(dec, \text{maxl}(e')), n, n, \max(\text{maxl}(e'), \text{maxl}(e)))$

Proof of the example: simplify

$$Q_0 = O_{start} := k \stackrel{R}{\leftarrow} key; mk \stackrel{R}{\leftarrow} mkey; \mathbf{return}();$$

$$(\mathbf{run} \ Q_A(k, mk) \ \parallel \ \mathbf{run} \ Q_B(k, mk))$$

$$Q_A(k, mk) = \mathbf{foreach} \ i \leq n \ \mathbf{do} \ O_A() := k' \stackrel{R}{\leftarrow} key; r'' \stackrel{R}{\leftarrow} coins;$$

$$e : \mathit{bitstring} \leftarrow \mathit{enc}(k2b(k'), k, r'');$$

$$ma \leftarrow \mathit{mac}'(e, mk); \mathbf{return}(e, ma)$$

$$Q_B(k, mk) = \mathbf{foreach} \ i' \leq n \ \mathbf{do} \ O_B(e' : \mathit{bitstring}, ma' : \mathit{macstring}) :=$$

$$\mathbf{find} \ j \leq n \ \mathbf{suchthat} \ \mathbf{defined}(e[j], ma[j]) \wedge e' = e[j] \wedge$$

$$ma' = ma[j] \ \mathbf{then}$$

$$k'' \leftarrow k'[j]; \mathbf{return}()$$

$$\mathit{dec}(e', k) = \mathit{dec}(\mathit{enc}(k2b(k'[j]), k, r''[j]), k) = i_{\perp}(k2b(k'[j]))$$

Proof of the example: security of the encryption

$$Q_0 = O_{start} := k \stackrel{R}{\leftarrow} key; mk \stackrel{R}{\leftarrow} mkey; \mathbf{return}();$$

$$(\mathbf{run} Q_A(k, mk) \parallel \mathbf{run} Q_B(k, mk))$$

$$Q_A(k, mk) = \mathbf{foreach} i \leq n \mathbf{do} O_A() := k' \stackrel{R}{\leftarrow} key; r'' \stackrel{R}{\leftarrow} coins;$$

$$e : \mathit{bitstring} \leftarrow \mathit{enc}'(Z(k2b(k')), k, r'');$$

$$ma \leftarrow \mathit{mac}'(e, mk); \mathbf{return}(e, ma)$$

$$Q_B(k, mk) = \mathbf{foreach} i' \leq n \mathbf{do} O_B(e' : \mathit{bitstring}, ma' : \mathit{macstring}) :=$$

$$\mathbf{find} j \leq n \mathbf{suchthat} \mathbf{defined}(e[j], ma[j]) \wedge e' = e[j] \wedge$$

$$ma' = ma[j] \mathbf{then}$$

$$k'' \leftarrow k'[j]; \mathbf{return}()$$

Probability: $\text{Succ}_{SE}^{\text{ind-cpa}}(\mathbf{time} + n \mathbf{time}(\mathit{mac}, \text{maxl}(e)) +$
 $n^2 \mathbf{time}(\mathit{bitstring}, \text{maxl}(e'), \text{maxl}(e)), n, \text{length}(key))$

Proof of the example: simplify

$$Q_0 = O_{start} := k \stackrel{R}{\leftarrow} key; mk \stackrel{R}{\leftarrow} mkey; \mathbf{return}();$$

$$(\mathbf{run} Q_A(k, mk) \parallel \mathbf{run} Q_B(k, mk))$$

$$Q_A(k, mk) = \mathbf{foreach} i \leq n \mathbf{do} O_A() := k' \stackrel{R}{\leftarrow} key; r'' \stackrel{R}{\leftarrow} coins;$$

$$e : \mathit{bitstring} \leftarrow \mathit{enc}'(Z_k, k, r'');$$

$$ma \leftarrow \mathit{mac}'(e, mk); \mathbf{return}(e, ma)$$

$$Q_B(k, mk) = \mathbf{foreach} i' \leq n \mathbf{do} O_B(e' : \mathit{bitstring}, ma' : \mathit{macstring}) :=$$

$$\mathbf{find} j \leq n \mathbf{suchthat} \mathbf{defined}(e[j], ma[j]) \wedge e' = e[j] \wedge$$

$$ma' = ma[j] \mathbf{then}$$

$$k'' \leftarrow k'[j]; \mathbf{return}()$$

$$Z(k2b(k')) = Z_k$$

Proof of the example: secrecy

$$Q_0 = O_{start} := k \stackrel{R}{\leftarrow} \text{key}; mk \stackrel{R}{\leftarrow} \text{mkey}; \mathbf{return}();$$

$$(\mathbf{run} \ Q_A(k, mk) \ \parallel \ \mathbf{run} \ Q_B(k, mk))$$

$$Q_A(k, mk) = \mathbf{foreach} \ i \leq n \ \mathbf{do} \ O_A() := k' \stackrel{R}{\leftarrow} \text{key}; r'' \stackrel{R}{\leftarrow} \text{coins};$$

$$e : \text{bitstring} \leftarrow \text{enc}'(Z_k, k, r'');$$

$$ma \leftarrow \text{mac}'(e, mk); \mathbf{return}(e, ma)$$

$$Q_B(k, mk) = \mathbf{foreach} \ i' \leq n \ \mathbf{do} \ O_B(e' : \text{bitstring}, ma' : \text{macstring}) :=$$

$$\mathbf{find} \ j \leq n \ \mathbf{suchthat} \ \mathbf{defined}(e[j], ma[j]) \wedge e' = e[j] \wedge$$

$$ma' = ma[j] \ \mathbf{then}$$

$$k'' \leftarrow k'[j]; \mathbf{return}()$$

Preserves the one-session secrecy of k'' but not its secrecy.

Recent results

- **PQ-CryptoVerif**, with C. Jacomme (CSF'24): All game transformations of CryptoVerif are sound against quantum adversaries.
 - Library of primitives with post-quantum instantiations.
 - Application to hybrid versions of TLS 1.3 and SSH.
- **CV2EC**, with P. Boutry, C. Doczkal, B. Grégoire, P.-Y. Strub (CSF'24): Translation from CryptoVerif assumptions on primitives to EasyCrypt.
 - Prove assumptions on primitives from lower-level assumptions in EasyCrypt.
 - Prove protocols in CryptoVerif.

Recent results

- **CV2F^{*}**, with K. Bhargavan, A. Fromherz, C. Jacomme, B. Lipp, E. Mera (in progress): Translation from CryptoVerif models of protocols to F^{*} implementations.
 - Translates security properties proved in CryptoVerif to F^{*} axioms.
 - Translates functional assumptions in CryptoVerif to lemmas to prove in F^{*}.
- **key compromise (CSF'24)**.

Case studies

- Full domain hash signature (with David Pointcheval)
Encryption schemes of Bellare-Rogaway'93 (with David Pointcheval)
OAEP
- Kerberos V, with and without PKINIT (with Aaron D. Jaggard, Andre Scedrov, and Joe-Kai Tsay).
- OEKE (variant of Encrypted Key Exchange, with David Pointcheval).
- SSH Transport Layer Protocol (with David Cadé).
- Avionic 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)

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 to interact with the prover to make it follow a specific sequence of games.