

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

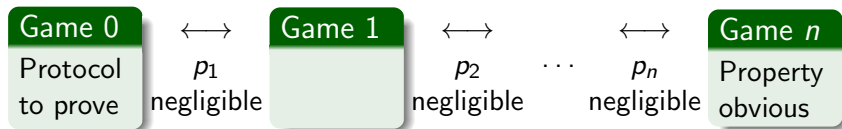
## CryptoVerif

- 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).

$M ::=$	terms
$x, y, z$	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
$Q \mid Q'$	parallel composition
<b>foreach</b> $i \leq N$ <b>do</b> $Q$	replication $N$ times
$O(x_1 : T_1, \dots, x_m : T_m) := P$	oracle definition
$P ::=$	oracle body
<b>yield</b>	end
<b>return</b> $(M_1, \dots, M_m); Q$	result
<b>event</b> $e(M_1, \dots, M_m); P$	event
$x \xleftarrow{R} T; P$	random number generation (uniform)
$x : T \leftarrow M; P$	assignment
<b>if</b> $M$ <b>then</b> $P$ <b>else</b> $P'$	conditional
<b>insert</b> $L(M_1, \dots, M_m); P$	add an entry to list $L$
<b>get</b> $L(x_1, \dots, x_m)$ <b>suchthat</b> $M$ <b>in</b> $P$ <b>else</b> $P'$	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  $\text{enc}(m, k) = r \xleftarrow{R} \text{enc\_seed}; \text{enc}_r(m, k, r)$ .

- Decryption function  $\text{dec}(c, k)$  such that

$$\text{dec}(\text{enc}_r(m, k, r'), k) = \text{injb}ot(m)$$

The decryption returns a bitstring or bottom:

- bottom when decryption fails,
- the cleartext when decryption succeeds.

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

## 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:

$$enc'(m, (k, mk)) = c1 \parallel mac(c1, mk) \text{ where } c1 = enc(m, k).$$

**letfun** *full\_enc*(*m* : *bitstring*, *k* : *key*, *mk* : *mkey*) =

*c1*  $\leftarrow$  *enc*(*m*, *k*);

*concat*(*c1*, *mac*(*c1*, *mk*)).

**letfun** *full\_dec*(*c* : *bitstring*, *k* : *key*, *mk* : *mkey*) =

**let** *concat*(*c1*, *mac1*) = *c* **in**

(**if** *verify*(*c1*, *mk*, *mac1*) **then** *dec*(*c1*, *k*) **else** bottom)

**else**

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)**

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

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

$$Ostart() := b \xleftarrow{R} \text{bool}; k \xleftarrow{R} \text{key}; mk \xleftarrow{R} \text{mkey}; \mathbf{return}$$

Initialization:

- ① Define an oracle  $Ostart$ . (The adversary will call this oracle.)
- ②  $Ostart$  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  $qEnc$  times

$LR(x, y, 0) = x$ ,  $LR(x, y, 1) = y$ , and  $LR(x, y, b)$  is defined only when  $x$  and  $y$  have the same length.

**foreach**  $i \leq qEnc$  **do**

$Oenc(m1 : \text{bitstring}, m2 : \text{bitstring}) :=$

**if**  $Z(m1) = Z(m2)$  **then**

$m0 \leftarrow$  **if**  $b$  **then**  $m1$  **else**  $m2$ ;

**return**( $full\_enc(m0, k, mk)$ ).

- ❶ **foreach**  $i \leq qEnc$  **do** represents  $qEnc$  copies, indexed by  $i \in [1, qEnc]$ . The oracle can be called  $qEnc$  times.
- ❷ The oracle takes two messages as input,  $m1$  and  $m2$ .
- ❸ It verifies that they have the same length ( $Z(m1) = Z(m2)$ ).  
 $Z(x)$  is the bitstring of the same length as  $x$  containing only zeroes.

# IND-CPA: left-or-right encryption oracle

$enc(LR(.,., b), k)$  called at most  $qEnc$  times

$LR(x, y, 0) = x$ ,  $LR(x, y, 1) = y$ , and  $LR(x, y, b)$  is defined only when  $x$  and  $y$  have the same length.

**foreach**  $i \leq qEnc$  **do**

$Oenc(m1 : \text{bitstring}, m2 : \text{bitstring}) :=$

**if**  $Z(m1) = Z(m2)$  **then**

$m0 \leftarrow$  **if**  $b$  **then**  $m1$  **else**  $m2$ ;

**return**( $full\_enc(m0, k, mk)$ ).

- ④  $m0$  is set to  $LR(m1, m2, b)$ .
- ⑤ The oracle returns the encryption of  $m0$ .

# Example: summary of the initial game

```
Ostart() :=  $b \xleftarrow{R} \text{bool}$ ;  $k \xleftarrow{R} \text{key}$ ;  $mk \xleftarrow{R} \text{mkey}$ ; return;  
foreach  $i \leq q_{\text{Enc}}$  do  
Oenc( $m1 : \text{bitstring}, m2 : \text{bitstring}$ ) :=  
if  $Z(m1) = Z(m2)$  then  
 $m0 \leftarrow$  if  $b$  then  $m1$  else  $m2$ ;  
return(full_enc( $m0, k, mk$ )).
```

We prove secrecy of  $b$ :

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

# 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)

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

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

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

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

$$\approx$$

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

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

# IND-CPA symmetric encryption: CryptoVerif definition

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

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

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

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

$Z(x)$  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

```
 $O_{start}() := b \stackrel{R}{\leftarrow} \text{bool}; k \stackrel{R}{\leftarrow} \text{key}; mk \stackrel{R}{\leftarrow} \text{mkey}; \text{return};$   
foreach  $i \leq qEnc$  do  
   $O_{enc}(m1 : \text{bitstring}, m2 : \text{bitstring}) :=$   
    if  $Z(m1) = Z(m2)$  then  
       $m0 \leftarrow \text{if } b \text{ then } m1 \text{ else } m2;$   
    return(( $c1 \leftarrow (r \stackrel{R}{\leftarrow} \text{enc\_seed}; \text{enc\_r}(m0, k, r)); \text{concat}(c1, \text{mac}(c1, mk))$ )))
```

CryptoVerif inlines the definition of *full\_enc*.

# IND-CPA proof: expand terms into processes

```
 $Ostart() := b \stackrel{R}{\leftarrow} \text{bool}; k \stackrel{R}{\leftarrow} \text{key}; mk \stackrel{R}{\leftarrow} \text{mkey}; \textbf{return};$   
foreach  $i \leq qEnc$  do  
   $Oenc(m1 : \text{bitstring}, m2 : \text{bitstring}) :=$   
    if  $Z(m1) = Z(m2)$  then  
      if  $b$  then  
         $r \stackrel{R}{\leftarrow} \text{enc\_seed}; c1 \leftarrow \text{enc\_r}(m1, k, r); \textbf{return}(\text{concat}(c1, \text{mac}(c1, mk)))$   
      else  
         $r \stackrel{R}{\leftarrow} \text{enc\_seed}; c1 \leftarrow \text{enc\_r}(m2, k, r); \textbf{return}(\text{concat}(c1, \text{mac}(c1, mk)))$ 
```

# IND-CPA proof: renaming variables

$Ostart() := b \xleftarrow{R} \text{bool}; k \xleftarrow{R} \text{key}; mk \xleftarrow{R} \text{mkey}; \mathbf{return};$

**foreach**  $i \leq qEnc$  **do**

$Oenc(m1 : \text{bitstring}, m2 : \text{bitstring}) :=$

**if**  $Z(m1) = Z(m2)$  **then**

**if**  $b$  **then**

$r_2 \xleftarrow{R} \text{enc\_seed}; c1 \leftarrow \text{enc\_r}(m1, k, r_2); \mathbf{return}(\text{concat}(c1, \text{mac}(c1, mk)))$

**else**

$r_1 \xleftarrow{R} \text{enc\_seed}; c1 \leftarrow \text{enc\_r}(m2, k, r_1); \mathbf{return}(\text{concat}(c1, \text{mac}(c1, mk)))$

CryptoVerif renames the two definitions of  $r$  to distinct names.

# IND-CPA proof: apply the IND-CPA assumption

```

Ostart() :=  $b \xleftarrow{R} \text{bool}$ ;  $k \xleftarrow{R} \text{key}$ ;  $mk \xleftarrow{R} \text{mkey}$ ; return;
foreach  $i \leq qEnc$  do
  Oenc( $m1 : \text{bitstring}$ ,  $m2 : \text{bitstring}$ ) :=
    if  $Z(m1) = Z(m2)$  then
      if  $b$  then
         $r_4 \xleftarrow{R} \text{enc\_seed}$ ;  $c1 \leftarrow \text{enc\_r}'(Z(m1), k, r_4)$ ; return(concat( $c1$ , mac( $c1$ ,  $mk$ )))
      else
         $r_3 \xleftarrow{R} \text{enc\_seed}$ ;  $c1 \leftarrow \text{enc\_r}'(Z(m2), k, r_3)$ ; return(concat( $c1$ , mac( $c1$ ,  $mk$ )))

```

CryptoVerif uses the IND-CPA assumption. It replaces the cleartext messages ( $m1$  and  $m2$ ) with bitstrings of the same length containing only zeroes ( $Z(m1)$ ,  $Z(m2)$ ).

Probability:  $\text{Succ}_{SE}^{\text{ind-cpa}}(t', qEnc, l_m)$  with  $t' = t + qEnc(\text{time}(=, l_m) + \text{time}(\text{mac}, l_{c1}) + \text{time}(\text{concat}, l_{c1}) + 2\text{time}(Z, l_m))$ .

# IND-CPA proof: merge

```
 $Ostart() := b \stackrel{R}{\leftarrow} \text{bool}; k \stackrel{R}{\leftarrow} \text{key}; mk \stackrel{R}{\leftarrow} \text{mkey}; \textbf{return};$   
foreach  $i \leq qEnc$  do  
   $Oenc(m1 : \text{bitstring}, m2 : \text{bitstring}) :=$   
  if  $Z(m1) = Z(m2)$  then  
     $r_3 \stackrel{R}{\leftarrow} \text{enc\_seed}; c1 \leftarrow \text{enc\_r}'(Z(m2), k, r_3); \textbf{return}(\text{concat}(c1, \text{mac}(c1, mk)))$ 
```

CryptoVerif merges the two branches of the test **if**  $b$  **then**, because they execute the same code, knowing that  $Z(m1) = Z(m2)$  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

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

where

$t' = t + q_e(\mathbf{time}(=, l) + \mathbf{time}(\mathit{mac}, l') + \mathbf{time}(\mathit{concat}, l') + 2\mathbf{time}(Z, l))$   
 $l'$  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.)



# INT-CTXT

## Definition (INT-CTXT symmetric encryption)

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

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

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

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

# INT-CTXT experiment in CryptoVerif

```
Ostart() :=  $k \xleftarrow{R} \text{key}; mk \xleftarrow{R} \text{mkey}; \text{return};$   
((foreach  $i_{enc} \leq q_{Enc}$  do  
   $O_{enc}(m_0 : \text{bitstring}) :=$   
   $c_0 \leftarrow \text{full\_enc}(m_0, k, mk); \text{insert } \text{ciphertexts}(c_0); \text{return}(c_0))$   
|  
(foreach  $i_{dec} \leq q_{Dec}$  do  
   $O_{decTest}(c : \text{bitstring}) :=$   
  get  $\text{ciphertexts}(= c)$  in return( $true$ ) else  
  if  $\text{full\_dec}(c, k, mk) \neq \text{bottom}$   
    then event bad; return( $true$ )  
    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**:

**foreach**  $ienc \leq qEnc$  **do**

$Oenc(m0[ienc] : \text{bitstring}) := c0[ienc] \leftarrow full\_enc(m0[ienc], k, mk); \dots$

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**:

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

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**:

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

becomes

**foreach**  $i \leq N$  **do** ...  $x[i] \leftarrow M$ ;  $y[i] \leftarrow M'$ ;  $P$   
|  $O(x' : T) :=$   
    **find**  $j \leq N$  **suchthat** **defined**( $x[j], y[j]$ )  $\wedge x' = x[j]$  **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,  $\text{Succ}_{\text{MAC}}^{\text{uf-cma}}(t, q_m, q_v, l)$  is negligible if  $t$  is polynomial in the security parameter:

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

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

where  $\mathcal{A}$  runs in time at most  $t$ ,  
 calls  $\text{mac}(\cdot, k)$  at most  $q_m$  times with messages of length at most  $l$ ,  
 calls  $\text{verify}(\cdot, k, \cdot)$  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  $k \xleftarrow{R} \text{mkey}$  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

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



# MAC: CryptoVerif definition

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

$k \xleftarrow{R} mkey; ($   
     **foreach**  $i_m \leq q_m$  **do**  $Omac(x : bitstring) := \mathbf{return}(mac(x, k)) \mid$   
     **foreach**  $i_v \leq q_v$  **do**  $Overify(m : bitstring, t : macstring) :=$   
         **return** $(verify(m, k, t))$

$\approx$

$k \xleftarrow{R} mkey; ($   
     **foreach**  $i_m \leq q_m$  **do**  $Omac(x : bitstring) := ma \leftarrow mac(x, k); \mathbf{return}(ma)$   
     **foreach**  $i_v \leq q_v$  **do**  $Overify(m : bitstring, t : macstring) :=$   
         **find**  $j \leq N$  **suchthat**  $\mathbf{defined}(x[j], ma[j]) \wedge m = x[j] \wedge$   
              $t = ma[j] \mathbf{ then true else false}$

# MAC: CryptoVerif definition

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

```

 $k \xleftarrow{R} mkey; ($ 
  foreach  $i_m \leq q_m$  do  $Omac(x : \text{bitstring}) := \mathbf{return}(mac(x, k)) \mid$ 
  foreach  $i_v \leq q_v$  do  $Overify(m : \text{bitstring}, t : \text{macstring}) :=$ 
    return( $verify(m, k, t)$ )

```

$\approx_{\text{Succ}_{\text{MAC}}^{\text{uf-cma}}(\mathbf{time}, q_m, q_v, \max(\maxl(x), \maxl(m)))}$

```

 $k \xleftarrow{R} mkey; ($ 
  foreach  $i_m \leq q_m$  do  $Omac(x : \text{bitstring}) := ma \leftarrow mac'(x, k); \mathbf{return}(ma$ 
  foreach  $i_v \leq q_v$  do  $Overify(m : \text{bitstring}, t : \text{macstring}) :=$ 
    find  $j \leq N$  suchthat  $\mathbf{defined}(x[j], ma[j]) \wedge m = x[j] \wedge$ 
       $t = ma[j] \mathbf{ then true else false}$ 

```

# 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

```

Ostart() :=  $k \xleftarrow{R} \text{key}; mk \xleftarrow{R} \text{mkey}; \text{return};$ 
((foreach  $i_{\text{enc}} \leq q_{\text{Enc}}$  do Oenc( $m_0 : \text{bitstring}$ ) :=
   $c_0 \leftarrow (c_1 \leftarrow (r \xleftarrow{R} \text{enc\_seed}; \text{enc\_r}(m_0, k, r)); \text{concat}(c_1, \text{mac}(c_1, mk)))$ ;
  insert ciphertexts( $c_0$ ); return( $c_0$ ))
| (foreach  $i_{\text{dec}} \leq q_{\text{Dec}}$  do OdecTest( $c : \text{bitstring}$ ) :=
  get ciphertexts(=  $c$ ) in return(true) else
  if (let  $\text{concat}(c_2, \text{mac}_1) = c$  in
    if verify( $c_2, mk, \text{mac}_1$ ) then dec( $c_2, k$ ) else bottom
    else bottom)  $\neq$  bottom
  then event bad; return(true)
  else return(false)))

```

CryptoVerif inlines *full\_enc* and *full\_dec*.

# INT-CTXT proof: encode **insert** and **get**

```

Ostart() :=  $k \xleftarrow{R} \text{key}; mk \xleftarrow{R} \text{mkey}; \mathbf{return};$ 
((foreach  $i_{enc} \leq q_{Enc}$  do  $O_{enc}(m_0 : \text{bitstring}) :=$ 
   $c_0 \leftarrow (c_1 \leftarrow (r \xleftarrow{R} \text{enc\_seed}; \text{enc\_r}(m_0, k, r)); \text{concat}(c_1, \text{mac}(c_1, mk)))$ ;
   $\text{ciphertexts}_1 \leftarrow c_0; \mathbf{return}(c_0)$ )
| (foreach  $i_{dec} \leq q_{Dec}$  do  $O_{decTest}(c : \text{bitstring}) :=$ 
  find  $u \leq q_{Enc}$  suchthat  $\mathbf{defined}(\text{ciphertexts}_1[u]) \wedge \text{ciphertexts}_1[u] = c$ 
  then return}(true)
  else if (let  $\text{concat}(c_2, \text{mac}_1) = c$  in
    if  $\text{verify}(c_2, mk, \text{mac}_1)$  then  $\text{dec}(c_2, k)$  else bottom
    else bottom)  $\neq \text{bottom}$ 
  then event bad; return}(true)
  else return}(false)))
```

# INT-CTXT proof: expand terms into processes

$Ostart() := k \xleftarrow{R} \text{key}; mk \xleftarrow{R} \text{mkey}; \text{return};$

$((\text{foreach } ienc \leq qEnc \text{ do } Oenc(m0 : \text{bitstring}) :=$

$r \xleftarrow{R} \text{enc\_seed}; c1 \leftarrow \text{enc\_r}(m0, k, r); c0 \leftarrow \text{concat}(c1, \text{mac}(c1, mk));$   
 $\text{return}(c0))$

$| (\text{foreach } idec \leq qDec \text{ do } OdecTest(c : \text{bitstring}) :=$

$\text{find } u \leq qEnc \text{ suchthat } \text{defined}(c0[u]) \wedge c0[u] = c$

$\text{then return}(\text{true})$

$\text{else let } \text{concat}(c2, \text{mac1}) = c \text{ in}$

$\text{if } \text{verify}(c2, mk, \text{mac1}) \text{ then}$

$\text{if } \text{dec}(c2, k) \neq \text{bottom} \text{ then event } \text{bad}; \text{return}(\text{true})$

$\text{else return}(\text{false})$

$\text{else return}(\text{false})$

$\text{else return}(\text{false}))$

# INT-CTXT proof: apply SUF-CMA MAC

```

Ostart() :=  $k \xleftarrow{R} \text{key}; mk \xleftarrow{R} \text{mkey}; \text{return};$ 
((foreach  $ienc \leq qEnc$  do Oenc( $m0 : \text{bitstring}$ ) :=
   $r \xleftarrow{R} \text{enc\_seed}; c1 \leftarrow \text{enc\_r}(m0, k, r);$ 
   $c0 \leftarrow \text{concat}(c1, (ma2 \leftarrow \text{mac}(c1, mk); ma2)); \text{return}(c0))$ 
| (foreach  $idec \leq qDec$  do OdecTest( $c : \text{bitstring}$ ) :=
  find  $u \leq qEnc$  suchthat defined( $c0[u]$ )  $\wedge$   $c0[u] = c$ 
  then return(true)
  else let  $\text{concat}(c2, mac1) = c$  in
    if (find  $ri \leq qEnc$  suchthat defined( $c1[ri], ma2[ri]$ )  $\wedge$   $c2 = c1[ri] \wedge$ 
       $mac1 = ma2[ri]$  then true else false) then
      if  $\text{dec}(c2, k) \neq \text{bottom}$  then event bad; return(true)
      else return(false)
    else return(false)
  else return(false)))
```

# INT-CTXT proof: expand terms into processes; simplify

```
Ostart() := k  $\xleftarrow{R}$  key; mk  $\xleftarrow{R}$  mkey; return;  
((foreach ienc  $\leq$  qEnc do Oenc(m0 : bitstring) :=  
  r  $\xleftarrow{R}$  enc_seed; c1  $\leftarrow$  enc_r(m0, k, r);  
  ma2  $\leftarrow$  mac(c1, mk); c0  $\leftarrow$  concat(c1, ma2); return(c0))  
| (foreach idec  $\leq$  qDec do OdecTest(c : bitstring) :=  
  find u  $\leq$  qEnc suchthat defined(c0[u])  $\wedge$  c0[u] = c  
  then return(true)  
  else let concat(c2, mac1) = c in  
    find ri  $\leq$  qEnc suchthat defined(c1[ri], ma2[ri])  $\wedge$  c2 = c1[ri]  $\wedge$   
      ma1 = ma2[ri] then  
        event bad; return(true)  
    else return(false)  
  else return(false)))
```



# INT-CTXT proof: simplify

```

Ostart() :=  $k \stackrel{R}{\leftarrow} \text{key}; mk \stackrel{R}{\leftarrow} \text{mkey}; \mathbf{return};$ 
((foreach  $ienc \leq qEnc$  do  $Oenc(m0 : \text{bitstring}) :=$ 
   $r \stackrel{R}{\leftarrow} \text{enc\_seed}; c1 \leftarrow \text{enc\_r}(m0, k, r);$ 
   $ma2 \leftarrow \text{mac}(c1, mk); c0 \leftarrow \text{concat}(c1, ma2); \mathbf{return}(c0))$ 
| (foreach  $idec \leq qDec$  do  $OdecTest(c : \text{bitstring}) :=$ 
  find  $u \leq qEnc$  suchthat  $\text{defined}(c0[u]) \wedge c0[u] = c$ 
  then return}(true)
  else let  $\text{concat}(c2, mac1) = c$  in return}(false)
  else return}(false)))

```

When the first **find** fails, the second **find** also fails, so it is removed.

Event *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-CTXT property of encrypt-then-MAC is at most

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

where

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

$l$  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.

# Results

- 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)

# 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.

# 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  $F^\star$   
(proved security properties can be translated as well;  
further proofs can be done on the generated  $F^\star$  code)
- Improve support for **state**:
  - Loops with mutable state;
  - Primitives with internal state.

# Additional material



# Alternative syntax

Shown syntax	Alternative syntax
<b>foreach</b> $i \leq n$ <b>do</b>	$!i \leq n$
<b>foreach</b> $i \leq n$ <b>do</b>	$!n$ (when $i$ is not used)
$x \stackrel{R}{\leftarrow} T; P$	<b>new</b> $x : T; P$
$x \leftarrow M; P$	<b>let</b> $x = M$ <b>in</b> $P$

Oracles front-end	Channels front-end
$O(x_1 : T_1, \dots, x_m : T_m) := P$	<b>in</b> ( $c, x : T$ ); $P$
<b>return</b> ( $M_1, \dots, M_m$ ); $Q$	<b>out</b> ( $c, M$ ); $Q$

# 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

$$Ostart() := k_A \stackrel{R}{\leftarrow} T_k; k_B \stackrel{R}{\leftarrow} T_k; \mathbf{return}; (Q_K \mid Q_S)$$

$$Q_K = \mathbf{foreach} \ i \leq n \ \mathbf{do} \ O_K(h : T_h, k : T_k) :=$$

$$\quad \mathbf{if} \ h = A \ \mathbf{then} \ k' \leftarrow k_A \ \mathbf{else}$$

$$\quad \mathbf{if} \ h = B \ \mathbf{then} \ k' \leftarrow k_B \ \mathbf{else} \ k' \leftarrow k$$

$$Q_S = \mathbf{foreach} \ i' \leq n' \ \mathbf{do} \ O_S(h' : T_h) :=$$

$$\quad \mathbf{find} \ j \leq n \ \mathbf{suchthat} \ \mathbf{defined}(h[j], k'[j]) \wedge h' = h[j] \ \mathbf{then} \ P_1(k'[j])$$

$$\quad \mathbf{else} \ P_2$$

# 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

$Ostart() := k_A \stackrel{R}{\leftarrow} T_k; k_B \stackrel{R}{\leftarrow} T_k; \mathbf{return}; (Q_K \mid Q_S)$

$Q_K = \mathbf{foreach} \ i \leq n \ \mathbf{do} \ O_K(h : T_h, k : T_k) :=$

**if**  $h = A$  **then**  $k'_1 \leftarrow k_A$  **else**

**if**  $h = B$  **then**  $k'_2 \leftarrow k_B$  **else**  $k'_3 \leftarrow k$

$Q_S = \mathbf{foreach} \ i' \leq n' \ \mathbf{do} \ O_S(h' : T_h) :=$

**find**  $j \leq n$  **suchthat**  $\mathbf{defined}(h[j], k'_1[j]) \wedge h' = h[j]$  **then**  $P_1(k'_1[j])$

**orfind**  $j \leq n$  **suchthat**  $\mathbf{defined}(h[j], k'_2[j]) \wedge h' = h[j]$  **then**  $P_1(k'_2[j])$

**orfind**  $j \leq n$  **suchthat**  $\mathbf{defined}(h[j], k'_3[j]) \wedge h' = h[j]$  **then**  $P_1(k'_3[j])$

**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  $x \stackrel{R}{\leftarrow} T$  under a **if** or a **find** duplicates it.

A subsequent single assignment renaming will distinguish cases.)

## Example

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

becomes

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

# Syntactic transformations (4)

- **Merge arrays**: merge several variables  $x_1, \dots, x_n$  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:**  $x \leftarrow M; P$  implies that  $x = M$  in  $P$
  - **Tests:** **if**  $M = N$  **then**  $P$  implies that  $M = N$  in  $P$
  - **Definitions of cryptographic primitives**
  - When a **find** guarantees that  $x[j]$  is **defined**, equalities that hold at definition of  $x$  also hold under the find (after substituting  $j$  for the array indices at the definition of  $x$ )
  - **Elimination of collisions:** if  $x$  is created by **new**  $x : T$ ,  $x[i] = x[j]$  implies  $i = j$ , up to negligible probability (when  $T$  is large)
- These equalities are combined to simplify terms.
- When terms can be simplified, processes are simplified accordingly.  
For instance:
  - If  $M$  simplifies to **true**, then **if**  $M$  **then**  $P_1$  **else**  $P_2$  simplifies 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**:  $\mathbf{event}(e_1(x)) \Rightarrow \mathbf{event}(e_2(x))$  means that, if  $e_1(x)$  has been executed, then  $e_2(x)$  has been executed.
- **Injective correspondence**:  $\mathbf{inj-event}(e_1(x)) \Rightarrow \mathbf{inj-event}(e_2(x))$  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.