CryptoVerif: Mechanizing Game-Based Proofs

Bruno Blanchet

INRIA Paris
Bruno.Blanchet@inria.fr

December 2020

Outline

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.)

Game 0		Game 1				Game n
Protocol to prove	$\longleftrightarrow$ p_1 negligible		$\longleftrightarrow$ p_2 negligible	$\cdots$	$\longleftrightarrow$ p_n negligible	Property obvious

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
foreach $i \leq N$ **do** Q	replication N times
$O(x_1 : T_1, \dots, x_m : T_m) := P$	oracle definition
$P ::=$	oracle body
yield	end
return$(M_1, \dots, M_m); Q$	result
event $e(M_1, \dots, M_m); P$	event
$x \xleftarrow{R} T; P$	random number generation (uniform)
$x : T \leftarrow M; P$	assignment
if M **then** P **else** P'	conditional
insert $L(M_1, \dots, M_m); P$	add an entry to list L
get $L(x_1, \dots, x_m)$ **suchthat** M **in** P **else** P'	list lookup

Example: 1. symmetric encryption

We consider a probabilistic, length-revealing encryption scheme.

Definition (Symmetric encryption scheme SE)

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

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

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 $\cup$ {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:

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

$$
\begin{array}{l}
\textbf{letfun } \mathit{full\_enc}(m : \mathit{bitstring}, k : \mathit{key}, mk : \mathit{mkey}) = \\
\quad c1 \leftarrow enc(m,k); \\
\quad concat(c1, mac(c1,mk)).
\end{array}
$$

$$
\begin{array}{l}
\textbf{letfun } \mathit{full\_dec}(c : \mathit{bitstring}, k : \mathit{key}, mk : \mathit{mkey}) = \\
\quad \textbf{let } concat(c1, mac1) = c \textbf{ in} \\
\qquad (\textbf{if } verify(c1, mk, mac1) \textbf{ then } dec(c1,k) \textbf{ else } \text{bottom}) \\
\quad \textbf{else} \\
\qquad \text{bottom}.
\end{array}
$$

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)

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

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

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

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.

$$\begin{array}{l}\textbf{foreach } i \leq qEnc \textbf{ do}\\ Oenc(m1 : bitstring, m2 : bitstring) :=\\ \textbf{if } Z(m1) = Z(m2) \textbf{ then}\\ m0 \leftarrow \textbf{if } b \textbf{ then } m1 \textbf{ else } m2;\\ \textbf{return}(full\_enc(m0,k,mk)).\end{array}$$

1. **foreach** $i \leq qEnc$ **do** represents $qEnc$ copies, indexed by $i \in [1, qEnc]$. The oracle can be called $qEnc$ times.
2. The oracle takes two messages as input, $m1$ and $m2$.
3. 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.

$$\begin{array}{l}\textbf{foreach } i \leq qEnc \textbf{ do}\\ Oenc(m1 : bitstring, m2 : bitstring) :=\\ \textbf{if } Z(m1) = Z(m2) \textbf{ then}\\ m0 \leftarrow \textbf{if } b \textbf{ then } m1 \textbf{ else } m2;\\ \textbf{return}(full\_enc(m0,k,mk)).\end{array}$$

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

Example: summary of the initial game

$$
\begin{array}{l}
\mathit{Ostart}() := b \xleftarrow{R} \mathit{bool}; k \xleftarrow{R} \mathit{key}; \mathit{mk} \xleftarrow{R} \mathit{mkey}; \textbf{return};\\
\textbf{foreach } i \leq \mathit{qEnc} \textbf{ do}\\
\mathit{Oenc}(m1 : \mathit{bitstring}, m2 : \mathit{bitstring}) :=\\
\textbf{if } Z(m1) = Z(m2) \textbf{ then}\\
m0 \leftarrow \textbf{if } b \textbf{ then } m1 \textbf{ else } m2;\\
\textbf{return}(\mathit{full\_enc}(m0, k, \mathit{mk})).
\end{array}
$$

We prove secrecy of b:

$$\textbf{query secret } b$$

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

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)

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

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

IND-CPA symmetric encryption: CryptoVerif definition

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

$$k \xleftarrow{R} key; \textbf{foreach } i \le q_e \textbf{ do } Oenc(x : bitstring) :=$$
$$r' \xleftarrow{R} enc\_seed; \textbf{return}(enc\_r(x, k, r'))$$
$$\approx$$
$$k \xleftarrow{R} key; \textbf{foreach } i \le q_e \textbf{ do } Oenc(x : bitstring) :=$$
$$r' \xleftarrow{R} enc\_seed; \textbf{return}(enc\_r(Z(x), k, r'))$$

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

IND-CPA symmetric encryption: CryptoVerif definition

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

$$k \xleftarrow{R} key; \textbf{foreach } i \leq q_e \textbf{ do } Oenc(x : bitstring) :=$$
$$r' \xleftarrow{R} enc\_seed; \textbf{return}(enc\_r(x,k,r'))$$
$$\approx_{\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(\textbf{time}, q_e, \mathrm{maxl}(x))}$$
$$k \xleftarrow{R} key; \textbf{foreach } i \leq q_e \textbf{ do } Oenc(x : bitstring) :=$$
$$r' \xleftarrow{R} enc\_seed; \textbf{return}(enc\_r'(Z(x),k,r'))$$

$Z(x)$ 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

$Ostart() := b \stackrel{R}{\leftarrow} bool; k \stackrel{R}{\leftarrow} key; mk \stackrel{R}{\leftarrow} mkey; \textbf{return};$

$\textbf{foreach } i \leq qEnc \textbf{ do}$

$Oenc(m1 : bitstring, m2 : bitstring) :=$

$\textbf{if } Z(m1) = Z(m2) \textbf{ then}$

$m0 \leftarrow \textbf{if } b \textbf{ then } m1 \textbf{ else } m2;$

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

CryptoVerif inlines the definition of *full_enc*.

IND-CPA proof: expand terms into processes

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

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

$Oenc(m1 : bitstring, m2 : bitstring) :=$

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

if b **then**

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

else

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

IND-CPA proof: renaming variables

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

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

$Oenc(m1 : bitstring, m2 : bitstring) :=$

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

if b **then**

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

else

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

CryptoVerif renames the two definitions of r to distinct names.

IND-CPA proof: apply the IND-CPA assumption

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

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

$Oenc(m1 : bitstring, m2 : bitstring) :=$

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

if b **then**

$r_4 \xleftarrow{R} enc\_seed; c1 \leftarrow enc\_r'(Z(m1), k, r_4);$ **return**$(concat(c1, mac(c1, mk$

else

$r_3 \xleftarrow{R} enc\_seed; c1 \leftarrow enc\_r'(Z(m2), k, r_3);$ **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: $\mathrm{Succ}_{\mathrm{SE}}^{\mathrm{ind-cpa}}(t', qEnc, l_m)$ with $t' = t + qEnc(\mathbf{time}(=, l_m) + \mathbf{time}(mac, l_{c1}) + \mathbf{time}(concat, l_{c1}) + 2\mathbf{time}(Z, l_m))$.

IND-CPA proof: merge

$Ostart() := b \stackrel{R}{\leftarrow} bool; k \stackrel{R}{\leftarrow} key; mk \stackrel{R}{\leftarrow} mkey;$ **return**;

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

$Oenc(m1 : bitstring, m2 : bitstring) :=$

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

$r_3 \stackrel{R}{\leftarrow} enc\_seed; c1 \leftarrow enc\_r'(Z(m2), k, r_3);$ **return**$(concat(c1, mac(c1, mk)))$

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

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

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

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

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

$Ostart() := k \stackrel{R}{\leftarrow} key; mk \stackrel{R}{\leftarrow} mkey;$ **return**;

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

$Oenc(m0 : bitstring) :=$

$c0 \leftarrow full\_enc(m0, k, mk);$ **insert** $ciphertexts(c0);$ **return**$(c0)$)

|

(**foreach** $idec \leq qDec$ **do**

$OdecTest(c : bitstring) :=$

get $ciphertexts(= c)$ **in return**$(true)$ **else**

if $full\_dec(c, k, mk) \neq bottom$

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:

$$\textbf{foreach } ienc \leq qEnc \textbf{ do}$$
$$Oenc(m0[ienc] : bitstring) := c0[ienc] \leftarrow full\_enc(m0[ienc], k, mk); \ldots$$

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. (**if** $\ldots$ **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:

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

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:

$$\begin{aligned}&\textbf{foreach } i \leq N \textbf{ do } \ldots \textbf{insert } L(M, M'); P\\ &\mid O(x' : T) := \textbf{get } L(x, y) \textbf{ suchthat } x' = x \textbf{ in } P'(y)\end{aligned}$$

becomes

$$\begin{aligned}&\textbf{foreach } i \leq N \textbf{ do } \ldots x[i] \leftarrow M; y[i] \leftarrow M'; P\\ &\mid O(x' : T) :=\\ &\qquad \textbf{find } j \leq N \textbf{ suchthat defined}(x[j], y[j]) \wedge x' = x[j] \textbf{ then } P'(y[j])\end{aligned}$$

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, $\mathrm{Succ}_{\mathrm{MAC}}^{\mathrm{suf-cma}}(t, q_m, q_v, l)$ is negligible if t is polynomial in the security parameter:

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

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

where $\mathcal{A}$ runs in time at most t,
calls $mac(., k)$ at most q_m times with messages of length at most l,
calls $verify(., k, .)$ 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 $k \xleftarrow{R} mkey$ is used only for generating and verifying MACs, the verification of a MAC with $verify(m, k, t)$ can succeed only if m is in the list (array) of messages whose $mac(\cdot, k)$ has been computed, with result t by the protocol
- so we can replace a call to $verify$ with an array lookup:
  if the call to mac is $mac(x, k)$, we replace $verify(m, k, t)$ with

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

MAC: CryptoVerif definition

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

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

$$\approx$$

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

MAC: CryptoVerif definition

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

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

$\approx_{\mathrm{Succ}^{\mathrm{suf-cma}}_{\mathrm{MAC}}(\textbf{time}, q_m, q_v, \max(\mathrm{maxl}(x), \mathrm{maxl}(m)))}$

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

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

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

$(($**foreach** $ienc \leq qEnc$ **do** $Oenc(m0 : bitstring) :=$

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

insert $ciphertexts(c0);$ **return**$(c0))$

$| ($**foreach** $idec \leq qDec$ **do** $OdecTest(c : bitstring) :=$

get $ciphertexts(= c)$ **in return**$(true)$ **else**

if (**let** $concat(c2, mac1) = c$ **in**

if $verify(c2, mk, mac1)$ **then** $dec(c2, k)$ **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**

$Ostart() := k \stackrel{R}{\leftarrow} key; mk \stackrel{R}{\leftarrow} mkey; \textbf{return};$

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

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

$ciphertexts_1 \leftarrow c0; \textbf{return}(c0))$

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

$\textbf{find } u \leq qEnc \textbf{ suchthat defined}(ciphertexts_1[u]) \wedge ciphertexts_1[u] = c$

$\textbf{then return}(true)$

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

$\textbf{if } verify(c2, mk, mac1) \textbf{ then } dec(c2, k) \textbf{ else } bottom$

$\textbf{else } bottom) \neq bottom$

$\textbf{then event } bad; \textbf{return}(true)$

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

INT-CTXT proof: expand terms into processes

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

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

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

$\quad \mathbf{return}(c0))$

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

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

$\quad \mathbf{then}\ \mathbf{return}(true)$

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

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

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

$\qquad\qquad\qquad \mathbf{else\ return}(false)$

$\qquad \mathbf{else\ return}(false)$

$\quad \mathbf{else\ return}(false)))$

INT-CTXT proof: apply SUF-CMA MAC

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

$(($**foreach** $ienc \leq qEnc$ **do** $Oenc(m0 : bitstring) :=$

$r \xleftarrow{R} enc\_seed; c1 \leftarrow enc\_r(m0, k, r);$

$c0 \leftarrow concat(c1, (ma2 \leftarrow mac'(c1, mk); ma2));$ **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**

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 $dec(c2, k) \neq bottom$ **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**$(c1[ri], ma2[ri]) \wedge c2 = c1[ri] \wedge$

$mac1 = ma2[ri]$ **then**

event $bad;$ **return**$(true)$

else return$(false)$

else return$(false)))$

INT-CTXT proof: simplify

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

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

$\quad r \xleftarrow{R} enc\_seed; c1 \leftarrow enc\_r(m0, k, r);$

$\quad ma2 \leftarrow mac'(c1, mk); c0 \leftarrow concat(c1, ma2); \textbf{return}(c0))$

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

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

$\quad \textbf{then return}(true)$

$\quad \textbf{else let}\ concat(c2, mac1) = c\ \textbf{in return}(false)$

$\quad \textbf{else return}(false)))$

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

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

where
$t' = t + q_e\mathbf{time}(enc\_r, l) + q_e\mathbf{time}(concat, l') + q_d q_e\mathbf{time}(=, l'') + q_d\mathbf{time}(let\ concat, l') + q_d\mathbf{time}(dec, l')$
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
foreach $i \leq n$ **do**	$!i \leq n$
foreach $i \leq n$ **do**	$!n$ (when i is not used)
$x \xleftarrow{R} T; P$	**new** $x : T; P$
$x \leftarrow M; P$	**let** $x = M$ **in** P

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

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

$$
\begin{aligned}
&Ostart() := k_A \stackrel{R}{\leftarrow} T_k; k_B \stackrel{R}{\leftarrow} T_k; \textbf{return}; (Q_K \mid Q_S)\\
&Q_K = \textbf{foreach } i \leq n \textbf{ do } O_K(h : T_h, k : T_k) :=\\
&\qquad \textbf{if } h = A \textbf{ then } k' \leftarrow k_A \textbf{ else}\\
&\qquad \textbf{if } h = B \textbf{ then } k' \leftarrow k_B \textbf{ else } k' \leftarrow k\\
&Q_S = \textbf{foreach } i' \leq n' \textbf{ do } O_S(h' : T_h) :=\\
&\qquad \textbf{find } j \leq n \textbf{ suchthat defined}(h[j], k'[j]) \wedge h' = h[j] \textbf{ then } P_1(k'[j])\\
&\qquad \textbf{else } P_2
\end{aligned}
$$

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

$$
\begin{aligned}
&Ostart() := k_A \stackrel{R}{\leftarrow} T_k; k_B \stackrel{R}{\leftarrow} T_k; \textbf{return}; (Q_K \mid Q_S)\\
&Q_K = \textbf{foreach } i \leq n \textbf{ do } O_K(h : T_h, k : T_k) :=\\
&\qquad \textbf{if } h = A \textbf{ then } k'_1 \leftarrow k_A \textbf{ else}\\
&\qquad \textbf{if } h = B \textbf{ then } k'_2 \leftarrow k_B \textbf{ else } k'_3 \leftarrow k\\
&Q_S = \textbf{foreach } i' \leq n' \textbf{ do } O_S(h' : T_h) :=\\
&\qquad \textbf{find } j \leq n \textbf{ suchthat defined}(h[j], k'_1[j]) \wedge h' = h[j] \textbf{ then } P_1(k'_1[j])\\
&\qquad \textbf{orfind } j \leq n \textbf{ suchthat defined}(h[j], k'_2[j]) \wedge h' = h[j] \textbf{ then } P_1(k'_2[j])\\
&\qquad \textbf{orfind } j \leq n \textbf{ suchthat defined}(h[j], k'_3[j]) \wedge h' = h[j] \textbf{ then } P_1(k'_3[j])\\
&\qquad \textbf{else } P_2
\end{aligned}
$$

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

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

becomes

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

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

$$\textbf{if } cond \textbf{ then } P \textbf{ else } P$$

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: $\textbf{inj-event}(e_1(x)) \Rightarrow \textbf{inj-event}(e_2(x))$ 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.