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 developped 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.)

Game 0	$\longleftrightarrow$	Game 1	$\longleftrightarrow$		$\longleftrightarrow$	Game n
Protocol to prove	p_1 negligible		p_2 negligible	$\cdots$	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 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
$\quad x, y, z, x[M_1, \dots, M_n]$	variable
$\quad f(M_1, \dots, M_n)$	function application

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

Process calculus for games: processes

$Q ::=$	oracle definitions
$\quad 0$	end
$\quad Q \parallel Q'$	parallel composition
$\quad$ **foreach** $i \leq N$ **do** Q	replication N times
$\quad$ **newOracle** $O; Q$	restriction for oracles
$\quad O(x_1 : T_1, \ldots, x_m : T_m) := P$	oracle declaration
$P ::=$	oracle bodies
$\quad$ **yield**	end
$\quad$ **return**$(M_1, \ldots, M_m); Q$	return
$\quad$ **event** $e(M_1, \ldots, M_m); P$	event
$\quad x \stackrel{R}{\leftarrow} T; P$	random number generation (uniform)
$\quad x : T \leftarrow M; P$	assignment
$\quad$ **if** M **then** P **else** P'	conditional
$\quad$ **find** $j \leq N$ **suchthat defined**$(x[j], \ldots) \wedge M$ **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 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** $y : key; Z(k2b(y)) = Z_k$.

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.

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

k' should remain secret.

Example: initialization

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

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

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

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

$$\begin{aligned} Q_A(k, mk) = \textbf{foreach } i \leq n \textbf{ do} \\ & O_A() := k' \xleftarrow{R} key; r'' \xleftarrow{R} coins; \\ & e \leftarrow enc(k2b(k'), k, r''); \\ & \textbf{return}(e, mac(e, mk)) \end{aligned}$$

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

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

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

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

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

$$\begin{aligned} Q_0 = O_{start} &:= k \xleftarrow{R} key; mk \xleftarrow{R} mkey; \textbf{return}(); \\ &(\textbf{run}\ Q_A(k, mk) \parallel \textbf{run}\ Q_B(k, mk)) \end{aligned}$$

$$\begin{aligned} Q_A(k, mk) = {} & \textbf{foreach}\ i \le n\ \textbf{do}\ O_A() := k' \xleftarrow{R} key; r'' \xleftarrow{R} coins; \\ & e \leftarrow enc(k2b(k'), k, r''); \\ & \textbf{return}(e, mac(e, mk)) \end{aligned}$$

$$\begin{aligned} Q_B(k, mk) = {} & \textbf{foreach}\ i' \le n\ \textbf{do}\ O_B(e' : bitstring, ma' : macstring) := \\ & \textbf{if}\ verify(e', mk, ma')\ \textbf{then} \\ & i_\perp(k2b(k'')) \leftarrow dec(e', k); \textbf{return}() \end{aligned}$$

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:

$$\begin{aligned} Q_A(k, mk) = \ &\textbf{foreach } i \leq n \textbf{ do } O_A[i]() := k'[i] \xleftarrow{R} key; r''[i] \xleftarrow{R} coins; \\ &e[i] \leftarrow enc(k2b(k'[i]), k, r''[i]); \\ &\textbf{return}(e[i], mac(e[i], mk)) \end{aligned}$$

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:

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

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:

$$\begin{array}{l} \quad\textbf{foreach}\ i \leq N\ \textbf{do}\ \ldots x[i] \leftarrow M; P \\ \| \textbf{foreach}\ i' \leq N'\ \textbf{do}\ \textbf{in}(c, y : T); \\ \qquad\quad \textbf{find}\ j \leq N\ \textbf{suchthat defined}(x[j]) \wedge y = x[j]\ \textbf{then}\ \ldots \end{array}$$

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.

$$\begin{array}{l} \quad\textbf{foreach } i \leq N \textbf{ do } \ \ldots x \leftarrow M; y \leftarrow M'; P \\ \| \textbf{ foreach } i' \leq N' \textbf{ do in}(c, x' : T); \\ \qquad\quad \textbf{find } j \leq N \textbf{ suchthat defined}(x[j]) \wedge x' = x[j] \textbf{ then } P'(y[j]) \end{array}$$

might be written with lists:

$$\begin{array}{l} \quad\textbf{foreach } i \leq N \textbf{ do } \ \ldots x \leftarrow M; y \leftarrow M'; \textbf{insert } L(x, y); P \\ \| \textbf{ foreach } i' \leq N' \textbf{ do in}(c, x' : T); \textbf{get } L(x, y) \textbf{ suchthat } x' = x \textbf{ in } P'(y) \end{array}$$

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.

$$\begin{array}{l} \quad \textbf{foreach } i \le N \textbf{ do } \ldots x[i] \leftarrow M; y[i] \leftarrow M'; P \\ \| \textbf{ foreach } i' \le N' \textbf{ do in}(c, x' : T); \\ \qquad \textbf{find } j \le N \textbf{ suchthat defined}(x[j]) \wedge x' = x[j] \textbf{ then } P'(y[j]) \end{array}$$

might be written with lists:

$$\begin{array}{l} \quad \textbf{foreach } i \le N \textbf{ do } \ldots x \leftarrow M; y \leftarrow M'; \textbf{insert } L(x, y); P \\ \| \textbf{ foreach } i' \le N' \textbf{ do in}(c, x' : T); \textbf{get } L(x, y) \textbf{ suchthat } x' = x \textbf{ in } P'(y) \end{array}$$

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

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

$$
\begin{array}{l}
k \stackrel{R}{\leftarrow} \mathit{mkey}; (\\
\quad \textbf{foreach } i \leq N \textbf{ do } \mathit{Omac}(x : \mathit{bitstring}) := \textbf{return}(\mathit{mac}(x, k)) \parallel \\
\quad \textbf{foreach } i' \leq N' \textbf{ do } \mathit{Overify}(x' : \mathit{bitstring}, m' : \mathit{macstring}) := \\
\quad\quad \textbf{return}(\mathit{verify}(x', k, m')))
\end{array}
$$

$$\approx$$

$$
\begin{array}{l}
k \stackrel{R}{\leftarrow} \mathit{mkey}; (\\
\quad \textbf{foreach } i \leq N \textbf{ do } \mathit{Omac}(x : \mathit{bitstring}) := \\
\quad\quad m \leftarrow \mathit{mac}(x, k); \textbf{insert } L(x, m); \textbf{return}(m) \parallel \\
\quad \textbf{foreach } i' \leq N' \textbf{ do } \mathit{Overify}(x' : \mathit{bitstring}, m' : \mathit{macstring}) := \\
\quad\quad \textbf{get } L(= x', = m') \textbf{ in return}(\textbf{true}) \textbf{ else return}(\textbf{false}))
\end{array}
$$

MAC: CryptoVerif definition

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

$$\begin{array}{l}
k \xleftarrow{R} mkey; (\\
\quad \textbf{foreach}\ i \leq N\ \textbf{do}\ Omac(x : bitstring) := \textbf{return}(mac(x, k)) \parallel \\
\quad \textbf{foreach}\ i' \leq N'\ \textbf{do}\ Overify(x' : bitstring, m' : macstring) := \\
\qquad \textbf{return}(verify(x', k, m')))
\end{array}$$

$$\approx$$

$$\begin{array}{l}
k \xleftarrow{R} mkey; (\\
\quad \textbf{foreach}\ i \leq N\ \textbf{do}\ Omac(x : bitstring) := \\
\qquad m \leftarrow mac(x, k); \textbf{return}(m) \parallel \\
\quad \textbf{foreach}\ i' \leq N'\ \textbf{do}\ Overify(x' : bitstring, m' : macstring) := \\
\qquad \textbf{find}\ j \leq N\ \textbf{suchthat}\ \textbf{defined}(x[j], m[j]) \wedge (x' = x[j]) \wedge \\
\qquad\quad (m' = m[j])\ \textbf{then return}(\textbf{true})\ \textbf{else return}(\textbf{false}))
\end{array}$$

MAC: CryptoVerif definition

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

$$\begin{aligned}
&k \xleftarrow{R} mkey; (\\
&\quad \mathbf{foreach}\ i \leq N\ \mathbf{do}\ Omac(x : bitstring) := \mathbf{return}(mac(x, k)) \parallel\\
&\quad \mathbf{foreach}\ i' \leq N'\ \mathbf{do}\ Overify(x' : bitstring, m' : macstring) :=\\
&\qquad \mathbf{return}(verify(x', k, m')))
\end{aligned}$$

$$\approx_{\mathrm{Succ}^{\mathrm{suf-cma}}_{\mathrm{MAC}}(\mathbf{time}, N, N', \max(\mathrm{maxl}(x), \mathrm{maxl}(x')))}$$

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

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.

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

$$\begin{aligned}
& k \xleftarrow{R} key; \textbf{foreach } i \leq N \textbf{ do } Oenc(x : bitstring) := \\
& \qquad r' \xleftarrow{R} coins; \textbf{return}(enc(x, k, r')) \\
\approx \\
& k \xleftarrow{R} key; \textbf{foreach } i \leq N \textbf{ do } Oenc(x : bitstring) := \\
& \qquad r' \xleftarrow{R} coins; \textbf{return}(enc({\color{blue}Z(x)}, k, r'))
\end{aligned}$$

$Z(x)$ is the bitstring of the same length as x containing only zeroes (for all $x : nonce$, $Z(x) = Znonce$, ...).

Symmetric encryption (IND-CPA)

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

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

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

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

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

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

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

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

Proof of the example: security of the MAC

$$Q_0 = O_{start} := k \xleftarrow{R} key; mk \xleftarrow{R} mkey; \textbf{return}();$$
$$(\textbf{run } Q_A(k, mk) \parallel \textbf{run } Q_B(k, mk))$$

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

$$Q_B(k, mk) = \textbf{foreach } i' \leq n \textbf{ do } O_B(e' : bitstring, ma' : macstring) :=$$
$$\textbf{find } j \leq n \textbf{ suchthat defined}(e[j], ma[j]) \wedge e' = e[j] \wedge$$
$$ma' = ma[j] \textbf{ then}$$
$$i_\perp(k2b(k'')) \leftarrow dec(e', k); \textbf{return}()$$

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

Proof of the example: simplify

$$Q_0 = O_{start} := k \xleftarrow{R} key; mk \xleftarrow{R} mkey; \textbf{return}();$$
$$(\textbf{run}\ Q_A(k, mk) \parallel \textbf{run}\ Q_B(k, mk))$$

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

$$Q_B(k, mk) = \textbf{foreach}\ i' \leq n\ \textbf{do}\ O_B(e' : bitstring, ma' : macstring) :=$$
$$\textbf{find}\ j \leq n\ \textbf{suchthat defined}(e[j], ma[j]) \land e' = e[j] \land$$
$$ma' = ma[j]\ \textbf{then}$$
$$k'' \leftarrow k'[j]; \textbf{return}()$$

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

Proof of the example: security of the encryption

$$Q_0 = O_{start} := k \xleftarrow{R} key; mk \xleftarrow{R} mkey; \textbf{return}();$$
$$(\textbf{run}\ Q_A(k, mk) \parallel \textbf{run}\ Q_B(k, mk))$$

$$Q_A(k, mk) = \textbf{foreach}\ i \leq n\ \textbf{do}\ O_A() := k' \xleftarrow{R} key; r'' \xleftarrow{R} coins;$$
$$e : bitstring \leftarrow enc'(Z(k2b(k')), k, r'');$$
$$ma \leftarrow mac'(e, mk); \textbf{return}(e, ma)$$

$$Q_B(k, mk) = \textbf{foreach}\ i' \leq n\ \textbf{do}\ O_B(e' : bitstring, ma' : macstring) :=$$
$$\textbf{find}\ j \leq n\ \textbf{suchthat defined}(e[j], ma[j]) \wedge e' = e[j] \wedge$$
$$ma' = ma[j]\ \textbf{then}$$
$$k'' \leftarrow k'[j]; \textbf{return}()$$

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

Proof of the example: simplify

$$
\begin{aligned}
Q_0 = O_{start} &:= k \xleftarrow{R} key; mk \xleftarrow{R} mkey; \mathbf{return}();\\
&(\mathbf{run}\ Q_A(k, mk) \parallel \mathbf{run}\ Q_B(k, mk))
\end{aligned}
$$

$$
\begin{aligned}
Q_A(k, mk) = \mathbf{foreach}\ i \leq n\ \mathbf{do}\ & O_A() := k' \xleftarrow{R} key; r'' \xleftarrow{R} coins;\\
& e : bitstring \leftarrow enc'({\color{red}Z_k}, k, r'');\\
& ma \leftarrow mac'(e, mk); \mathbf{return}(e, ma)
\end{aligned}
$$

$$
\begin{aligned}
Q_B(k, mk) = \mathbf{foreach}\ i' \leq n\ \mathbf{do}\ & O_B(e' : bitstring, ma' : macstring) :=\\
& \mathbf{find}\ j \leq n\ \mathbf{suchthat}\ \mathbf{defined}(e[j], ma[j]) \wedge e' = e[j] \wedge\\
& \quad ma' = ma[j]\ \mathbf{then}\\
& k'' \leftarrow k'[j]; \mathbf{return}()
\end{aligned}
$$

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

Proof of the example: secrecy

$$Q_0 = O_{start} := k \xleftarrow{R} key; mk \xleftarrow{R} mkey; \textbf{return}();$$
$$(\textbf{run}\ Q_A(k, mk) \parallel \textbf{run}\ Q_B(k, mk))$$

$$Q_A(k, mk) = \textbf{foreach}\ i \leq n\ \textbf{do}\ O_A() := k' \xleftarrow{R} key; r'' \xleftarrow{R} coins;$$
$$e : bitstring \leftarrow enc'(Z_k, k, r'');$$
$$ma \leftarrow mac'(e, mk); \textbf{return}(e, ma)$$

$$Q_B(k, mk) = \textbf{foreach}\ i' \leq n\ \textbf{do}\ O_B(e' : bitstring, ma' : macstring) :=$$
$$\textbf{find}\ j \leq n\ \textbf{suchthat}\ \textbf{defined}(e[j], ma[j]) \wedge e' = e[j] \wedge$$
$$ma' = ma[j]\ \textbf{then}$$
$$k'' \leftarrow k'[j]; \textbf{return}()$$

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): see next.

Basic treatment of key compromise

Include the compromise in the specification of the primitive itself.
Example: INT-CTXT = the adversary cannot forge a ciphertext that decrypts successfully (simplified).

$$
\begin{array}{l}
k \stackrel{R}{\leftarrow} key; (\\
\quad \textbf{foreach } i \leq n \textbf{ do } \mathrm{Oenc}(x[i] : cleartext) := \textbf{return}(\mathrm{enc}(x[i], k)) \parallel \\
\quad \textbf{foreach } i' \leq n' \textbf{ do } \mathrm{Odec}(y : ciphertext) := \textbf{return}(\mathrm{dec}(y, k))\\
\quad)\\
\approx\\
k \stackrel{R}{\leftarrow} key; (\\
\quad \textbf{foreach } i \leq n \textbf{ do } \mathrm{Oenc}(x[i] : cleartext) := z[i] \leftarrow \mathrm{enc}(x[i], k); \textbf{return}(z[i]) \parallel \\
\quad \textbf{foreach } i' \leq n' \textbf{ do } \mathrm{Odec}(y : ciphertext) :=\\
\\
\qquad \textbf{find } j \leq n \textbf{ suchthat defined}(x[j], z[j]) \wedge z[j] = y\\
\qquad \textbf{then return}(x[j]) \textbf{ else return}(\perp)\\
\quad)
\end{array}
$$

Basic treatment of key compromise

Include the compromise in the specification of the primitive itself.
Example: INT-CTXT = the adversary cannot forge a ciphertext that decrypts successfully (simplified).

$$
\begin{array}{l}
k \stackrel{R}{\leftarrow} \mathit{key}; (\\
\quad \textbf{foreach } i \leq n \textbf{ do } \mathrm{Oenc}(x[i] : \mathit{cleartext}) := \textbf{return}(\mathrm{enc}(x[i], k)) \parallel \\
\quad \textbf{foreach } i' \leq n' \textbf{ do } \mathrm{Odec}(y : \mathit{ciphertext}) := \textbf{return}(\mathrm{dec}(y, k)) \parallel \\
\quad \mathrm{Ocorrupt}() := \textbf{return}(k))\\
\approx\\
k \stackrel{R}{\leftarrow} \mathit{key}; (\\
\quad \textbf{foreach } i \leq n \textbf{ do } \mathrm{Oenc}(x[i] : \mathit{cleartext}) := z[i] \leftarrow \mathrm{enc}(x[i], k); \textbf{return}(z[i]) \parallel \\
\quad \textbf{foreach } i' \leq n' \textbf{ do } \mathrm{Odec}(y : \mathit{ciphertext}) := \\
\qquad \textbf{if defined}(\mathit{corrupt}) \textbf{ then return}(\mathrm{dec}(y, k)) \textbf{ else}\\
\qquad \textbf{find } j \leq n \textbf{ suchthat defined}(x[j], z[j]) \wedge z[j] = y\\
\qquad \textbf{then return}(x[j]) \textbf{ else return}(\perp) \parallel \\
\quad \mathrm{Ocorrupt}() := \mathit{corrupt} \leftarrow \mathrm{true}; \textbf{return}(k))
\end{array}
$$

Applications

- INT-CTXT encryption in WireGuard [EuroS&P'19]
- one-wayness in FDH [Crypto'06]
- UF-CMA signatures in
  - TLS 1.3 [S&P'17],
  - Signal [EuroS&P'17],
  - fixed ARINC823 public key protocol [CSF'17]

Limitations

- Works for computational assumptions, not for decisional assumptions.
- Does not work when the compromised “key” is used as argument in a sequence of key derivations using hash functions.
  - E.g., pre-shared key in TLS 1.3 and WireGuard.
- Does not allow proving in CryptoVerif properties with compromise of keys from assumptions without key compromise.

Extensions

1. Extension of the proof of secrecy useful for dynamic key compromise.
2. Extensions to overcome the limitations of the basic treatment.

Proving secrecy: toy example

Prove secrecy for a part of array k.

$$
\begin{aligned}
\textbf{foreach } i \leq n \textbf{ do } & O1() := k[i] \xleftarrow{R} key; \textbf{return}(); \\
& O2(compr[i] : bool) := \\
& \textbf{if } compr[i] \textbf{ then} \\
& \quad \textbf{return}(k[i]) \\
& \textbf{else} \\
& \quad s[i] \leftarrow k[i]; \textbf{return}()
\end{aligned}
$$

s is secret

Application: forward secrecy in a signed Diffie-Hellman protocol.

How to overcome limitations of basic treatment

Two steps:

1. Prove an authentication property, assuming the key is not compromised until the end of the session.
    - We can remove the compromise.
    - If the key is compromised after the end of the session, the property will be preserved (because it is an authentication property).
2. Use that property to prove other properties, including secrecy, in the presence of key compromise.

focus

focus $q_1, \ldots, q_m$ tells CryptoVerif to prove only the properties $q_1, \ldots, q_m$, as a first step.

- The other properties to prove are (temporarily) ignored.
- Allows more transformations:
  - events that do not occur in $q_1, \ldots, q_m$ can be removed;
  - only $q_1, \ldots, q_m$ are considered in the transformation **success simplify**.

When $q_1, \ldots, q_m$ are proved, CryptoVerif automatically goes back to before the **focus** command to prove the remaining properties.

Usage:

- For key compromise, prove the authentication property first.
- More generally, when different properties require different proofs.

success simplify

success simplify

1. first collects information known to be true when the adversary breaks at least one of the properties to prove.
2. then replaces parts of the game that contradict this information with **event_abort** adv_loses.
   - When these parts of the game are executed, the adversary cannot break any of the security properties to prove, so we can safely abort the game.

success simplify: canonical example

Suppose

- the active queries are **event**$(e_i) \Rightarrow$ **false** for events e_i executed by **event_abort** e_i;
- $\mathcal{F}_\mu$ are facts that hold at program point μ;
- μ_j for $j \in J$ are the program points of events e_i.

If for all $j \in J$, $\mathcal{F}_\mu \cup \mathcal{F}_{\mu_j}$ yields a contradiction (possibly up to elimination of collisions), then

- if μ is executed, then μ_j cannot be executed, so the adversary loses
- **success simplify** replaces the code at μ with **event_abort** adv_loses.

success simplify: example

The left- and right-hand sides of the definition of INT-CTXT with corruption can be distinguished from the following game only when event distinguish is executed.

$k \xleftarrow{R} \mathit{key}; ($
$\quad \textbf{foreach } i \leq n \textbf{ do } \mathrm{Oenc}(x[i] : \mathit{cleartext}) := z[i] \leftarrow \mathrm{enc}(x[i], k); \textbf{return}(z[i]) \parallel$
$\quad \textbf{foreach } i' \leq n' \textbf{ do } \mathrm{Odec}(y : \mathit{ciphertext}) :=$
$\qquad \textbf{if defined}(\mathit{corrupt}) \textbf{ then return}(\mathrm{dec}(y, k)) \textbf{ else}$
$\qquad \textbf{find } j \leq n \textbf{ suchthat defined}(x[j], z[j]) \wedge z[j] = y$
$\qquad \textbf{then return}(x[j]) \textbf{ else}$
$\qquad \textbf{if } \mathit{dec}(y, k) \neq \perp \textbf{ then } {}^{\mu}\textbf{event\_abort } \mathit{distinguish}$
$\qquad \textbf{else return}(\perp) \parallel$
$\quad \mathrm{Ocorrupt}() := \mathit{corrupt} \leftarrow \mathrm{true}; {}^{\mu_1}\textbf{return}(k)).$

success simplify: example

$k \stackrel{R}{\leftarrow} key; ($
 foreach $i \leq n$ **do** $\mathrm{Oenc}(x[i] : cleartext) := z[i] \leftarrow \mathrm{enc}(x[i], k);$ **return**$(z[i]) \parallel$
 foreach $i' \leq n'$ **do** $\mathrm{Odec}(y : ciphertext) :=$
  if defined$(corrupt)$ **then return**$(\mathrm{dec}(y, k))$ **else**
  find $j \leq n$ **suchthat defined**$(x[j], z[j]) \wedge z[j] = y$
  then return$(x[j])$ **else**
  if $dec(y, k) \neq \perp$ **then** $^{\mu}$**event_abort** distinguish
  else return$(\perp) \parallel$
 $\mathrm{Ocorrupt}() := corrupt \leftarrow \mathrm{true};$ $^{\mu_1}$**return**$(k)).$

- $\mathcal{F}_\mu \cup \mathcal{F}_{\mu_1}$ yields a contradiction;
- **success simplify** replaces code at μ_1 with **event_abort** adv_loses;
- k is never corrupted;
- ciphertext integrity without corruption shows that the probability of distinguish is negligible.

General strategy

1. Insert events e_i executed when some authentication properties are broken (and the key is not compromised).
2. **focus** on proving **event**$(e_i) \Rightarrow$ **false**.
3. **success simplify** removes the compromise of the key.
4. We prove queries **event**$(e_i) \Rightarrow$ **false**.
5. We go back to before **focus** and prove the other properties (implicitly using the authentication properties already proved).

Applications

- Forward secrecy with respect to the compromise of the pre-shared key in TLS 1.3 and WireGuard.
- PRF-ODH with compromise of Diffie-Hellman exponents, illustrated on Noise NK.
- Forward secrecy for OEKE.

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.