Part 4:
Towards High-Assurance Cryptographic Software
Modern applications employ crypto to protect sensitive data and transactions

- Protocols: TLS, SSH, Signal, OpenID
- End-to-End encryption: cloud storage, ...

Coding with crypto is easy to get wrong, even for security experts!

Formal verification can provide higher assurance (and find new attacks)
What goes wrong in TLS?

Legacy, obsolete, broken crypto
- RC4, RSA PKCS#1v1.5, 3DES, MD5

Protocol design flaws
- 3Shake, Logjam, SLOTH

Implementation bugs
- HeartBleed, GotoFail, SKIP, FREAK
Common Implementation Bugs

Bugs in Crypto Primitives
- Incorrect results, side-channels

Bugs in Crypto Usage
- Nonce reuse, Mac-Encode-Encrypt

Bugs in Protocol State Machine
- Incorrect composition

Bugs in Packet Parsing
- Buffer overrun
How do we fix it?

- **CRYPTO Library**
  - Understand crypto assumptions, eliminate weak crypto (MD5, RC4, …)
  - Use verified crypto libraries (HACL*, Vale, Jasmin, Fiat-Crypto, …)

- **PROTOCOL Design**
  - Symbolic analysis: find all logical flaws in protocol (ProVerif, Tamarin, …)
  - Computational analysis: prove security under classic crypto assumptions and definitions (CryptoVerif, EasyCrypt)

- **IMPLEMENTATIONS and APPLICATIONS**
  - Software verification: Prove secrecy and authentication for user data (F*, …)
  - Symbolic proofs: Link software verification to symbolic analysis (e.g. SignThenEncrypt)
  - Computational proofs: Link software verification to computational proofs (e.g. miTLS)
How do we fix it?

• **CRYPTO Library**
  - Understand **crypto assumptions**, eliminate **weak crypto** (MD5, RC4, …)
  - Use **verified crypto libraries** (HACL*, Vale, Jasmin, Fiat-Crypto, …)

• **PROTOCOL Design**
  - **Symbolic analysis**: find all logical flaws in protocol (ProVerif, Tamarin, …)
  - **Computational analysis**: prove security under classic crypto assumptions and definitions (CryptoVerif, EasyCrypt)

• **IMPLEMENTATIONS and APPLICATIONS**
  - **Software verification**: Prove secrecy and authentication for user data (F*, …)
  - **Symbolic proofs**: Link software verification to symbolic analysis (e.g. SignThenEncrypt)
  - **Computational proofs**: Link software verification to computational proofs (e.g. miTLS)
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 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.)

\[\text{Game 0} \quad \text{Protocol to prove} \quad \leftrightarrow \quad \text{Game 1} \quad \leftrightarrow \quad \text{Game } n \quad \leftrightarrow \quad \text{Property obvious}\]

\[p_1 \quad \text{negligible} \quad \leftrightarrow \quad p_2 \quad \text{negligible} \quad \cdots \quad p_n \quad \text{negligible}\]
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.
Games are formalized in a probabilistic process calculus: a small, specialized programming language.

The processes define the oracles that the adversary can call.

The runtime of processes is bounded:

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

Terms represent computations on messages (bitstrings).

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

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

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

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

We consider a probabilistic, length-revealing encryption scheme.

**Definition (Symmetric encryption scheme SE)**

- (Randomized) encryption function $\text{enc}_r(m, k, r)$ takes as input a message $m$, a key $k$, and random coins $r$.
  
  We define $\text{enc}(m, k) = r \xleftarrow{\text{R}} \text{enc\_seed}; \text{enc}_r(m, k, r)$.

- Decryption function $\text{dec}(c, k)$ such that
  
  $$\text{dec}(\text{enc}_r(m, k, r'), k) = \text{injbot}(m)$$

The decryption returns a bitstring or bottom:

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

The injection injbot maps a bitstring to the same bitstring in bitstring $\cup \{\text{bottom}\}$.
Example: 2. MAC

Definition (Message Authentication Code scheme MAC)

- MAC function \( \text{mac}(m, k) \) takes as input a message \( m \) and a key \( k \).
- Verification function \( \text{verify}(m, k, t) \) such that

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

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:

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

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

\[
c1 \leftarrow \text{enc}(m, k);
\]

\[
\text{concat}(c1, \text{mac}(c1, mk)).
\]

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

\[
\text{let } \text{concat}(c1, \text{mac}1) = c \text{ in}
\]

\[
(\text{if } \text{verify}(c1, mk, \text{mac}1) \text{ then } \text{dec}(c1, k) \text{ else } \text{bottom})
\]

else

\[
\text{bottom}.
\]
Security assumptions on primitives

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

In the example:

- The MAC is **SUF-CMA** (strongly unforgeable under chosen message attacks).
  An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (not produced by the MAC oracle).
Security assumptions on primitives

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

In the example:

- The MAC is **SUF-CMA** (strongly unforgeable under chosen message attacks).
  An adversary that has access to the MAC and verification oracles has a negligible probability of forging a MAC (not produced by the MAC oracle).

- The encryption is **IND-CPA** (indistinguishable under chosen plaintext attacks).
  An adversary has a negligible probability of distinguishing the encryption of two messages of the same length.
Security properties to prove

In the example:

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

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

Definition (INDistinguishability under Chosen Plaintext Attacks, IND-CPA)

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

where \(\mathcal{A}\) runs in time at most \(t\),
calls \(\text{enc}(LR(\ldots, b), k)\) at most \(q_e\) times on messages of length at most \(l\),
\(LR(x, y, 0) = x\), \(LR(x, y, 1) = y\), and \(LR(x, y, b)\) is defined only when \(x\)
and \(y\) have the same length.

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

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

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

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

```plaintext
foreach \( i \leq qEnc \) do
Oenc(m1 : bitstring, m2 : bitstring) :=
if \( Z(m1) = Z(m2) \) then
m0 ← if \( b \) then \( m1 \) else \( m2 \);
return(full_enc(m0, k, mk)).
```

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

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

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

4. \(m_0\) is set to \(LR(m_1, m_2, b)\).

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

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

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

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

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

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

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

We prove secrecy of \(b\):

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

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

\[ Q_1 \approx_p Q_2 \]

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

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

**Lemma**

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

We transform a game $G_0$ into an indistinguishable one using:

- **indistinguishability properties** $L \approx_p R$ given as axioms and that come from security assumptions on primitives. These equivalences are used inside a bigger game, using a proof by reduction:

$$G_1 \approx_0 C[L] \approx_{p'} C[R] \approx_0 G_2$$

- **syntactic transformations**: simplification, expansion of assignments, ...

We obtain a sequence of games $G_0 \approx_{p_1} G_1 \approx \ldots \approx_{p_m} G_m$, which implies $G_0 \approx_{p_1 + \ldots + p_m} G_m$.

If some trace property holds up to probability $p$ in $G_m$, then it holds up to probability $p + p_1 + \ldots + p_m$ 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)

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

where \( A \) runs in time at most \( t \), calls \( \text{enc}(LR(.,.,b),k) \) at most \( q_e \) times on messages of length at most \( l \), \( LR(x, y, 0) = x, LR(x, y, 1) = y \), and \( LR(x, y, b) \) is defined only when \( x \) and \( y \) have the same length.
IND-CPA symmetric encryption: CryptoVerif definition

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CryptoVerif inlines the definition of \textit{full} \_\textit{enc}.

---

Bruno Blanchet (INRIA)  
CryptoVerif  
December 2020  
24 / 62
IND-CPA proof: expand terms into processes

\[
\begin{align*}
O_{\text{start}}() & := b \xleftarrow{\$} \text{bool}; \ k \xleftarrow{\$} \text{key}; \ mk \xleftarrow{\$} \text{mkey}; \ \text{return}; \\
\text{foreach } i \leq q_{\text{Enc}} \text{ do} \\
O_{\text{enc}}(m_1 : \text{bitstring}, m_2 : \text{bitstring}) & := \\
\text{if } Z(m_1) = Z(m_2) \text{ then} \\
\text{if } b \text{ then} \\
\quad r \xleftarrow{\$} \text{enc\_seed}; \ c_1 \xleftarrow{\$} \text{enc\_r}(m_1, k, r); \ \text{return}(\text{concat}(c_1, \text{mac}(c_1, mk))) \\
\text{else} \\
\quad r \xleftarrow{\$} \text{enc\_seed}; \ c_1 \xleftarrow{\$} \text{enc\_r}(m_2, k, r); \ \text{return}(\text{concat}(c_1, \text{mac}(c_1, mk)))
\end{align*}
\]
IND-CPA proof: renaming variables

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

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

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

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

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

\[ r_2 \xleftarrow{\$} \text{enc\_seed}; c_1 \xleftarrow{\$} \text{enc\_r}(m_1, k, r_2); \text{return}(\text{concat}(c_1, \text{mac}(c_1, mk))) \]

\textbf{else}

\[ r_1 \xleftarrow{\$} \text{enc\_seed}; c_1 \xleftarrow{\$} \text{enc\_r}(m_2, k, r_1); \text{return}(\text{concat}(c_1, \text{mac}(c_1, mk))) \]

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

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

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

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

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

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

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

\textbf{else}

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

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

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

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

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

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

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

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

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

Result

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

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

where

$t' = t + q_e(\text{time}(\equiv, l) + \text{time}(\text{mac}, l') + \text{time}(\text{concat}, l') + 2\text{time}(Z, l))$

$l'$ is the length of ciphertexts for cleartexts of length $l$.

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

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

\[ \text{Succ}^\text{int-ctxt}_{SE}(t, q_e, q_d, l_e, l_d) = \]

\[ \max_A \Pr \left[ k \stackrel{R}{\leftarrow} \text{key}; c \leftarrow A^{\text{enc}(.,k), \text{dec}(.,k) \neq \bot} : \text{dec}(c, k) \neq \bot \land \right. \]

\[ \left. c \text{ is not the result of a call to the } \text{enc}(., k) \text{ oracle} \right] \]

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

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

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

((\textbf{foreach} ienc \leq qEnc \textbf{do})
\[
O_{\text{enc}}(m_0 : \text{bitstring}) :=
\]
\[
c_0 \leftarrow \text{full\_enc}(m_0, k, mk); \text{insert ciphertexts}(c_0); \text{return}(c_0))
\]
|  
(\textbf{foreach} idec \leq qDec \textbf{do})
\[
O_{\text{dec\_Test}}(c : \text{bitstring}) :=
\]
\[
\text{get ciphertexts}(= c) \text{ in return}(true) \text{ else}
\]
\[
\text{if } \text{full\_dec}(c, k, mk) \neq \text{bottom}
\]
\[
\text{then event bad; return}(true)
\]
\[
\text{else return}(false))
\]
Demo

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

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

```
foreach ienc <= qEnc do
Oenc(m0[ienc] : bitstring) := c0[ienc] <- full_enc(m0[ienc], k, mk); ...   
```

Requirements:

- Only variables with the current indices can be assigned.
- Variables may be defined at several places, but only one definition can be executed for the same indices. 
  
  (if ... then x <- M; P else x <- 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 ≤ N do ... x ← M; P
| O(y : T) := find j ≤ N such that defined(x[j]) ∧ y = x[j] then ...
```

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:

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

becomes

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

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

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

where \( \mathcal{A} \) runs in time at most \( t \), calls \( \text{mac}(., k) \) at most \( q_m \) times with messages of length at most \( l \), calls \( \text{verify}(., k, .) \) at most \( q_v \) times with messages of length at most \( l \).
MAC: intuition behind the CryptoVerif definition

By the previous definition, up to negligible probability,

- the adversary cannot forge a correct MAC

- so, assuming $k \xleftarrow{R} mkey$ is used only for generating and verifying MACs, the verification of a MAC with $\text{verify}(m, k, t)$ can succeed only if $m$ is in the list (array) of messages whose $\text{mac}(\cdot, k)$ has been computed, with result $t$ by the protocol

- so we can replace a call to $\text{verify}$ with an array lookup:
  if the call to $\text{mac}$ is $\text{mac}(x, k)$, we replace $\text{verify}(m, k, t)$ with

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

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

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

\[
\approx
\]

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

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

\[
k \xleftarrow{\$} mkey; (\text{\texttt{Omac}}(x : \text{bitstring}) := \text{\texttt{return}}(\text{mac}(x, k)) | \text{\texttt{Overify}}(m : \text{bitstring}, t : \text{macstring}) := \text{\texttt{return}}(\text{\texttt{verify}}(m, k, t)))
\]

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

\[
k \xleftarrow{\$} mkey; (\text{\texttt{Omac}}(x : \text{bitstring}) := m_{0} \leftarrow \text{\texttt{mac}}(x, k); \text{\texttt{return}}(ma) | \text{\texttt{Overify}}(m : \text{bitstring}, t : \text{macstring}) := \text{\texttt{find}} j \leq N \text{ suchthat } \text{\texttt{defined}}(x[j], ma[j]) \land m = x[j] \land t = ma[j] \text{ then true else false})
\]
MAC: using the CryptoVerif definition

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

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

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

(\textbf{foreach} i_{\text{enc}} \leq q_{\text{Enc}} \textbf{do} \ O_{\text{enc}}(m_0 : \text{bitstring}) :=

\quad c_0 \leftarrow (c_1 \leftarrow (r \leftarrow \text{enc}_{\text{seed}}; \text{enc}_r(m_0, k, r)); \text{concat}(c_1, \text{mac}(c_1, mk))); \]
\textbf{insert} ciphertexts(c_0); \text{return}(c_0))

| (\textbf{foreach} i_{\text{dec}} \leq q_{\text{Dec}} \textbf{do} \ O_{\text{decTest}}(c : \text{bitstring}) :=

\quad \text{get} \ \text{ciphertexts}(= c) \ \textbf{in} \ \text{return}(\text{true}) \ \textbf{else}
\textbf{if} \ \textbf{(let} \ \text{concat}(c_2, \text{mac}_1) = c \ \textbf{in}

\quad \textbf{if} \ \text{verify}(c_2, mk, \text{mac}_1) \ \textbf{then} \ \text{dec}(c_2, k) \ \textbf{else} \ \text{bottom}
\textbf{else} \ \text{bottom}) \neq \text{bottom}

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

\textbf{else} \ \text{return}(\text{false}))\]

CryptoVerif inlines \texttt{full_enc} and \texttt{full_dec}.
INT-CTXT proof: encode **insert** and **get**

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

\[
((\text{foreach } i_{\text{enc}} \leq q_{\text{Enc}} \text{ do } O_{\text{enc}}(m_0 : \text{bitstring}) := \\
\quad c_0 \leftarrow (c_1 \leftarrow (r \xleftarrow{\$} \text{enc}_\text{seed}; \text{enc}_r(m_0, k, r)); \text{concat}(c_1, \text{mac}(c_1, mk)))); \\
\quad \text{ciphertexts}_1 \leftarrow c_0; \text{return}(c_0)) \\
\mid (\text{foreach } i_{\text{dec}} \leq q_{\text{Dec}} \text{ do } O_{\text{dec Test}}(c : \text{bitstring}) := \\
\quad \text{find } u \leq q_{\text{Enc}} \text{ suchthat } \text{defined}((\text{ciphertexts}_1[u]) \land \text{ciphertexts}_1[u] = c \\
\quad \text{then } \text{return}(\text{true}) \\
\quad \text{else if } (\text{let } \text{concat}(c_2, \text{mac}1) = c \text{ in} \\
\quad \quad \text{if } \text{verify}(c_2, \text{mk}, \text{mac}1) \text{ then } \text{dec}(c_2, k) \text{ else } \text{bottom} \\
\quad \quad \text{else } \text{bottom}) \neq \text{bottom} \\
\quad \text{then } \text{event } \text{bad}; \text{return}(\text{true}) \\
\quad \text{else } \text{return}(\text{false}))))
\]
**INT-CTXT proof: expand terms into processes**

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

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

r \xleftarrow{R} \text{enc\_seed}; c_1 \leftarrow \text{enc\_r}(m_0, k, r); c_0 \leftarrow \text{concat}(c_1, \text{mac}(c_1, mk)));
\]

\[
\text{return}(c_0))
\]

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

\text{find } u \leq q_{\text{Enc}} \text{ such that } \text{defined}(c_0[u]) \land c_0[u] = c
\]

then \text{return}(true)
\]

else let \text{concat}(c_2, \text{mac}_1) = c in

\[
\text{if } \text{verify}(c_2, mk, \text{mac}_1) \text{ then}

\text{if } \text{dec}(c_2, k) \neq \text{bottom} \text{ then } \text{event bad}; \text{return}(true)
\]

else \text{return}(false)
\]

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

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

\[
((\text{foreach } i_{\text{enc}} \leq q_{\text{Enc}} \text{ do } O_{\text{enc}}(m_0 : \text{bitstring}) :=
\hspace{1cm} r \overset{R}{\leftarrow} \text{enc\_seed}; \ c_1 \overset{R}{\leftarrow} \text{enc\_r}(m_0, k, r);
\hspace{1cm} c_0 \leftarrow \text{concat}(c_1, (ma_2 \leftarrow \text{mac}'(c_1, mk); ma_2)); \ \text{return}(c_0))
\]

\[
| (\text{foreach } i_{\text{dec}} \leq q_{\text{Dec}} \text{ do } O_{\text{dec Test}}(c : \text{bitstring}) :=
\hspace{1cm} \text{find } u \leq q_{\text{Enc}} \text{ suchthat defined}(c_0[u]) \land c_0[u] = c
\hspace{1cm} \text{then return}(true)
\hspace{1cm} \text{else let } \text{concat}(c_2, mac_1) = c \text{ in}
\hspace{1cm} \hspace{1cm} \text{if } (\text{find } r_i \leq q_{\text{Enc}} \text{ suchthat defined}(c_1[r_i], ma_2[r_i]) \land c_2 = c_1[r_i] \land
\hspace{1cm} \hspace{1cm} ma_1 = ma_2[r_i] \ \text{then true else false}) \ \text{then}
\hspace{1cm} \hspace{1cm} \hspace{1cm} \text{if } \text{dec}(c_2, k) \neq \text{bottom} \ \text{then event bad; return}(true)
\hspace{1cm} \hspace{1cm} \hspace{1cm} \text{else return}(false)
\hspace{1cm} \hspace{1cm} \text{else return}(false))
\]

Bruno Blanchet (INRIA)
INT-CTX proof: expand terms into processes; simplify

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

((\text{foreach} ienc \leq qEnc \text{ do } O_{\text{enc}}(m_0 : \text{bitstring}) :=

\begin{align*}
    r & \xleftarrow{R} \text{enc\_seed}; c_1 \leftarrow \text{enc\_r}(m_0, k, r); \\
    ma_2 & \leftarrow \text{mac'}(c_1, mk); c_0 \leftarrow \text{concat}(c_1, ma_2); \text{return}(c_0))
\end{align*}

| (\text{foreach} idec \leq qDec \text{ do } O_{\text{dec\_test}}(c : \text{bitstring}) :=

\begin{align*}
    & \text{find } u \leq qEnc \text{ suchthat } \text{defined}(c_0[u]) \land c_0[u] = c \\
    \text{then return}(\text{true})
\end{align*}

else let \text{concat}(c_2, ma_1) = c \text{ in}

\begin{align*}
    & \text{find } ri \leq qEnc \text{ suchthat } \text{defined}(c_1[ri], ma_2[ri]) \land c_2 = c_1[ri] \land \\
    & ma_1 = ma_2[ri] \text{ then}
    \\
    & \text{event } \text{bad}; \text{return}(\text{true})
\end{align*}

else \text{return}(\text{false})
else \text{return}(\text{false}))}
INT-CTX proof: simplify

\[ O_{\text{start}}() := k \overset{R}{\leftarrow} \text{key}; mk \overset{R}{\leftarrow} \text{mkey}; \text{return}; \]
\[ ((\text{foreach ienc} \leq q\text{Enc} \text{ do } O_{\text{enc}}(m_0 : \text{bitstring}) := \]
\[ r \overset{R}{\leftarrow} \text{enc.seed}; c_1 \leftarrow \text{enc.r}(m_0, k, r); \]
\[ ma_2 \leftarrow \text{mac'}(c_1, mk); c_0 \leftarrow \text{concat}(c_1, ma_2); \text{return}(c_0)) \]
\[ | (\text{foreach idec} \leq q\text{Dec} \text{ do } O_{\text{decTest}}(c : \text{bitstring}) := \]
\[ \text{find } u \leq q\text{Enc} \text{ such that } \text{defined}(c_0[u]) \land c_0[u] = c \]
\[ \text{then return}(\text{true}) \]
\[ \text{else let } \text{concat}(c_2, mac_1) = c \text{ in return}(\text{false}) \]
\[ \text{else return}(\text{false})) \]

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

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

where

\[
t' = t + q_e \text{time}(\text{enc}_r, l) + q_e \text{time}(\text{concat}, l') + q_d q_e \text{time}(=, l'') + q_d \text{time}(< \text{concat}, l') + q_d \text{time}(\text{dec}, l')
\]

\( l \) is the maximum length of cleartexts

\( l' \) is the maximum length of ciphertexts

\( l'' \) is the maximum length of ciphertexts with MACs
First experiments

Tested on the following toy protocols (original and corrected versions):
- Otway-Rees (shared-key)
- Yahalom (shared-key)
- Denning-Sacco (public-key)
- Woo-Lam shared-key and public-key
- Needham-Schroeder shared-key and public-key

Shared-key encryption is assumed to be IND-CPA and INT-CTXT (authenticated encryption scheme).

Public-key encryption is assumed to be IND-CCA2.

We prove secrecy of session keys and authentication.
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∗
    (proved security properties can be translated as well; further proofs can be done on the generated F∗ code)
- Improve support for state:
  - Loops with mutable state;
  - Primitives with internal state.
Additional material
## Alternative syntax

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

### Oracles front-end

\[
O(x_1 : T_1, \ldots, x_m : T_m) := P
\]

### Channels front-end

\[
in(c, x : T); P
\]

\[
out(c, M); Q
\]
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**

\[
\begin{align*}
O_{\text{start}}() & := k_A \xleftarrow{R} T_k; k_B \xleftarrow{R} T_k; \text{return; } (Q_K \mid Q_S) \\
Q_K = \text{foreach } i \leq n \text{ do } & O_K(h : T_h, k : T_k) := \\
& \text{if } h = A \text{ then } k' \leftarrow k_A \text{ else } \\
& \text{if } h = B \text{ then } k' \leftarrow k_B \text{ else } k' \leftarrow k \\
Q_S = \text{foreach } i' \leq n' \text{ do } & O_S(h' : T_h) := \\
& \text{find } j \leq n \text{ suchthat defined}(h[j], k'[j]) \land h' = h[j] \text{ then } P_1(k'[j]) \text{ else } P_2
\end{align*}
\]
Syntactic transformations (2)

Single assignment renaming: when a variable is assigned at several places, rename it with a distinct name for each assignment. (Not completely trivial because of array references.)

Example

\begin{align*}
O_{\text{start}}() & := k_A \xleftarrow{R} T_k; k_B \xleftarrow{R} T_k; \text{return}; (Q_K \mid Q_S) \\
Q_K & = \text{foreach } i \leq n \text{ do } O_K(h : T_h, k : T_k) := \\
& \quad \text{if } h = A \text{ then } k'_1 \leftarrow k_A \text{ else} \\
& \quad \text{if } h = B \text{ then } k'_2 \leftarrow k_B \text{ else } k'_3 \leftarrow k \\
Q_S & = \text{foreach } i' \leq n' \text{ do } O_S(h' : T_h) := \\
& \quad \text{find } j \leq n \text{ suchthat defined}(h[j], k'_1[j]) \land h' = h[j] \text{ then } P_1(k'_1[j]) \\
& \quad \text{orfind } j \leq n \text{ suchthat defined}(h[j], k'_2[j]) \land h' = h[j] \text{ then } P_1(k'_2[j]) \\
& \quad \text{orfind } j \leq n \text{ suchthat defined}(h[j], k'_3[j]) \land h' = h[j] \text{ then } P_1(k'_3[j]) \\
& \quad \text{else } P_2
\end{align*}
Syntactic transformations (3)

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

**Example**

\[
x \leftarrow R \ nonce; \text{if } c \text{ then } P_1 \text{ else } P_2
\]

becomes

\[
\text{if } c \text{ then } x \leftarrow R \ nonce; P_1 \text{ else } x \leftarrow R \ nonce; P_2
\]
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

```
if cond then $P$ else $P$
```

Subsequent transformations can transform $P$ differently, depending on whether $cond$ holds.
Syntactic transformations (6): manual transformations

- **Insert an event**: to apply Shoup’s lemma.
  - A subprocess $P$ becomes **event** $e$.
  - The probability of distinguishing the two games is the probability of executing event $e$. It will be bound by a proof by sequences of games.

- **Replace a term with an equal term**. CryptoVerif verifies that the terms are really equal.
Simplification and elimination of collisions

- CryptoVerif collects equalities that come from:
  - **Assignments**: $x \leftarrow M; P$ implies that $x = M$ in $P$
  - **Tests**: $\text{if } M = N \text{ then } P$ implies that $M = N$ in $P$
  - **Definitions of cryptographic primitives**
    - When a **find** guarantees that $x[j]$ is defined, equalities that hold at definition of $x$ also hold under the find (after substituting $j$ for the array indices at the definition of $x$)
    - **Elimination of collisions**: if $x$ is created by **new** $x : T$, $x[i] = x[j]$ implies $i = j$, up to negligible probability (when $T$ is large)

- These equalities are combined to simplify terms.
- When terms can be simplified, processes are simplified accordingly. For instance:
  - If $M$ simplifies to **true**, then $\text{if } M \text{ then } P_1 \text{ else } P_2$ simplifies to $P_1$.
  - If a condition of **find** simplifies to **false**, then the corresponding branch is removed.
Security properties

- **Secrecy**: the adversary cannot distinguish the secrets from independent random numbers with several test queries.

- **Correspondence**: \( \text{event}(e_1(x)) \Rightarrow \text{event}(e_2(x)) \) means that, if \( e_1(x) \) has been executed, then \( e_2(x) \) has been executed.

- **Injective correspondence**: \( \text{inj-event}(e_1(x)) \Rightarrow \text{inj-event}(e_2(x)) \) means that each execution of \( e_1(x) \) corresponds to a distinct execution of \( e_2(x) \).
Proof strategy: advice

- One tries to execute each transformation given by the definition of a cryptographic primitive.
- When it fails, CryptoVerif tries to analyze why the transformation failed, and suggests syntactic transformations that could make it work.
- One tries to execute these syntactic transformations. (If they fail, they may also suggest other syntactic transformations, which are then executed.)
- We retry the cryptographic transformation, and so on.
Implementing & Verifying Crypto Protocols in F*
miTLS [2012-2018]

A verified reference implementation of TLS

- Covers TLS 1.0-1.2 (TLS 1.3 ongoing)
- Covers all major protocol modes and ciphersuites
Modular Architecture for miTLS
miTLS Security theorem

Main crypto result: concrete TLS & ideal TLS are computationally indistinguishable

We prove that ideal miTLS meets its secure channel specification using standard program verification (typing)
Modular Type-Based Cryptographic Verification

- MAC (SHA1)
- symmetric encryption (AES-CBC)
- symmetric encryption (RC4)

encrypted then-MAC

authenticated encryption

- fragment-MAC-encode-then-encrypt

Secure RPC

TLS 1.2

secure channel

- cryptographic algorithms
  - typed interfaces: cryptographic constructions
  - typed interfaces: security guarantees

- security protocols
  - typed interfaces: attacker models

- active adversaries

- some attack
  - another attack
Sample modular verification (protocol)

RPC protocol using Authenticated Encryption

some cryptographic implementation

authenticated encryption

Secure RPC

RPC API

any typed F# program

application code

Adversary Model

any typed F# program

active adversaries

Bytes

Networking

system libraries

Formatting

message format

security protocols

any typed F# program

application code
Sample modular verification (crypto)

RPC using Encrypt-then-MAC

cryptographic schemes

cryptographic constructions

probabilistic computational indistinguishability

AES-CBC encryption

MAC authentication

Encrypt-then-MAC authenticated encryption

Secure RPC RPC API

Formatting message format

Bytes Networking

system libraries

security protocols

Adversary Model

any typed F# program

any typed F7 program

active adversaries

application code

any typed F7 program

≈ IDEAL IND-CPA

≈ IDEAL INT-CMA

Encrypt

-then-MAC

authenticated encryption

Secure RPC

RPC API

any typed F7 program

application code

any typed F# program

active adversaries

application code

any typed F7 program

application code

any typed F# program

active adversaries

application code
Modeling Computational Assumptions
Modular Type-Based Cryptographic Verification

- MAC (SHA1)
- symmetric encryption (AES-CBC)
- symmetric encryption (RC4)

- INT-CMA
- IND-CPA

- encrypt then-MAC
- fragment-MAC-encode-then-encrypt

- authenticated encryption

- Secure RPC
- TLS 1.2

- secure channel

- cryptographic assumptions
- cryptographic constructions
- security guarantees
- security protocols
- typed interfaces: attacker models
- active adversaries
- typed interfaces: some attack
- another attack
Sample modular verification (protocol)

RPC protocol using Authenticated Encryption

some cryptographic implementation

authenticated encryption

Secure RPC

RPC API

Formatting

message format

Bytes

Networking

system libraries

security protocols

Adversary Model

any typed F# program

active adversaries

any typed F# program

application code
Sample modular verification (crypto)

RPC using Encrypt-then-MAC

cryptographic schemes

cryptographic constructions

probabilistic computational indistinguishability

AES-CBC encryption

MAC authentication

Encrypt-then-MAC authenticated encryption

Secure RPC RPC API

Adversary Model

any typed F# program

Bytes Networking

Formatting message format

system libraries

security protocols

Active adversaries

any typed F# program

Application code
Sample Typed Interface for Cryptography

MAC : integrity
Sample functionality:

Message Authentication Codes

```fsharp
module MAC

type text = bytes
val macsize : bytes

val GEN : unit -> key
val MAC : key -> text -> mac
val VERIFY : key -> text -> mac -> bool
```

This interface says nothing on the security of MACs.
MAC keys are abstract

Sample functionality:

Message Authentication Codes

module MAC

type text = bytes  val macsize

type key

type mac = bytes

val GEN : unit -> key
val MAC : key -> text -> mac
val VERIFY: key -> text -> mac -> bool
module MAC

type text = bytes  val macsize

val key

type mac = b:bytes{Length(b)=macsize}

val GEN : unit -> key
val MAC : key -> text -> mac
val VERIFY : key -> text -> mac -> bool
Sample functionality:

Message Authentication Codes

MAC keys are abstract

module MAC

val macsize

type text = bytes

type key

type mac = b:bytes{Length(b)=macsize}

predicate Msg of key * text

val GEN : unit -> key

val MAC : k:key -> t:text{Msg(k,t)} -> mac

val VERIFY: k:key -> t:text -> mac -> b:bool{ b=true => Msg(k,t)}
module MAC

open System.Security.Cryptography

let macsize = 20

let GEN() = randomBytes 16

let MAC k t = (new HMACSHA1(k)).ComputeHash t

let VERIFY k t m = (MAC k t = m)

module MAC

type text = bytes

val macsize

type key

type mac = b:bytes{Length(b)=macsize}

predicate Msg of key * text

val GEN : unit -> key

val MAC : k:key -> t:text{Msg(k,t)} -> mac

val VERIFY: k:key -> t:text -> mac
    -> b:bool{ b=true => Msg(k,t)}

"All verified messages have been MACed"

This can’t be true! (collisions)

"All verified messages have been MACed"

MAC keys are abstract

ideal F* interface

MACs are fixed sized

Msg is specified by protocols using MACs

Concrete F* implementation (using real crypto)

Message Authentication Codes
Sample computational assumption:

Resistance to Chosen-Message Existential Forgery Attacks (INT-CMA)

```fsharp
module INT_CMA_Game
open Mac

let private k = GEN()
let private log = ref []
let mac t =
    log := t::!log
    MAC k t

let verify t m =
    let v = VERIFY k t m in
    if v && not (mem t !log) then FORGERY
    v
```

Computational Safety
a probabilistic polytime program calling `mac` and `verify` forges a MAC only with negligible probability

CMA game (coded in F#)
Computational Safety for MACs

ideal system

Any p.p.t. adversary

perfectly safe by typing

ideal MAC

RPC protocol

secure RPC

Ideal MAC

Ideal filter

concrete system

Any p.p.t. adversary

safe too, with probability $1 - 1/\varepsilon$

secure RPC

RPC protocol

Concrete algorithm assumed INT-CMA computationally

error correction making VERIFY returns false on forgeries

sample protocol typed against ideal MAC interface

protocol adversary typed against RPC interface

INT-CMA adversary

Concrete algorithm assumed INT-CMA computationally

error correction making VERIFY returns false on forgeries

sample protocol typed against ideal MAC interface

protocol adversary typed against RPC interface
Module MAC

type text = bytes
val macsize

type key

type mac = b:bytes\{\text{Length}(b)=\text{macsize}\}

predicate \text{Msg} of key * text

val GEN : unit -> key
val MAC : k:key -> t:text\{\text{Msg}(k,t)\} -> mac
val VERIFY : k:key -> t:text -> mac
  -> b:bool\{ b=true => \text{Msg}(k,t)\}

val keysize

type keybytes = b:bytes\{\text{Length}(b)=\text{keysize}\}

val LEAK : k:key\{!t. \text{Msg}(k,t)\} -> b:keybytes
val COERCION : b:keybytes\{...\} -> k:key\{...\}

Ideal F* Interface

"All verified messages have been MACed"

MAC keys are abstract

MACs are fixed sized

MAC keys have concrete representations

It is safe to turn keys into bytes when all messages are verifiable

Sample ideal functionality:
Supporting Key Compromise

MAC keys are abstract

MACs are fixed sized

Msg is specified by protocols using MACs

"All verified messages have been MACed"

MAC keys have concrete representations

It is safe to turn keys into bytes when all messages are verifiable
Perfect Secrecy by Typing

- Secrecy is expressed using observational equivalences between systems that differ on their secrets.
- We prove (probabilistic, information theoretic) secrecy by typing, relying on type abstraction.

\[ I_\alpha = \alpha, \ldots, x : T_\alpha, \ldots \]
\[ P_\alpha \text{ range over pure modules such that } \vdash P_\alpha \sim I_\alpha. \]

**Theorem** (Secrecy by Typing).
Let \( A \) such that \( I_\alpha \vdash A : \text{bool}. \)
For all \( P^0_\alpha \) and \( P^1_\alpha \), we have \( P^0_\alpha \cdot A \approx P^1_\alpha \cdot A. \)
Encryption is parameterized by a module that abstractly define plaintexts, with interface

```plaintext
module Plaintext
val size: int
type plain
type repr = b:bytes{Length(b)=size}
val coerce : repr -> plain // turning bytes into secrets
val leak : plain -> repr // breaking secrecy!
val respond: plain -> plain // sample protocol code
```

The size of plaintext is fixed (as we cannot hide it)

Plain may also implement any protocol functions that operates on secrets

If we remove the `leak` function, we get secrecy by typing

If we remove the `coerce` function, we get integrity by typing
Ideal Interface for Authenticated Encryption

- Relying on basic cryptographic assumptions (IND-CPA, INT-CTXT) its **ideal implementation** never accesses plaintexts!

  Formally, ideal AE is typed using an abstract **plain** type

  \[
  \text{ENC} \ k \ p \quad \text{encrypts instead zeros to } c \quad \text{and logs } (k, c, p) \\
  \text{DEC} \ k \ c \quad \text{returns Some}(p) \text{ when } (k, c, p) \text{ is in the log, or None}
  \]

```plaintext
module AE
open Plaintext
type key
type cipher = b:bytes\{Length(b)= size + 16\}
val GEN: unit -> key
val ENC: key -> plain -> cipher
val DEC: key -> cipher -> plain option
```
An Ideal Interface for CCA2-Secure Encryption

- Its **ideal implementation** encrypts zeros instead of plaintexts so it never accesses plaintext representations, and can be typed parametrically.
Typed Secrecy from CCA2-Secure Encryption

**THEOREM 7** (Asymptotic Secrecy).
Let $P^0$ and $P^1$ p.p.t. secret with $\vdash P^b \leadsto I_{\text{PLAIN}}$.
Let $C_{\text{ENC}}$ p.p.t. CCA2-secure with $I_{\text{PLAIN}}^C \vdash C_{\text{ENC}} \leadsto I_{\text{ENC}}^C$.
Let $A$ p.p.t. with $I_{\text{PLAIN}}, I_{\text{ENC}} \vdash A : \text{bool}$.

$$P^0 \cdot C_{\text{ENC}} \cdot A \approx_\epsilon P^1 \cdot C_{\text{ENC}} \cdot A.$$

**THEOREM 8** (Ideal Functionality).
Let $P$ p.p.t. with $\vdash P \leadsto I_{\text{PLAIN}}^C$ (not necessarily secret).
Let $C_{\text{ENC}}$ p.p.t. CCA2-secure with $I_{\text{PLAIN}}^C \vdash C_{\text{ENC}} \leadsto I_{\text{ENC}}^C$.
Let $A$ p.p.t. with $I_{\text{PLAIN}}, I_{\text{ENC}} \vdash A$.

$$P \cdot C_{\text{ENC}} \cdot A \approx_\epsilon P \cdot C_{\text{ENC}} \cdot F_{\text{ENC}} \cdot A.$$
Variants: CPA & Authentication

• With **CPA-secure encryption**, we have a **weaker** ideal interface that demands ciphertext integrity before decryption

  \[
  \text{predicate } \text{Encrypted} \text{ of } \text{key} \times \text{cipher} \\
  \text{val } \text{ENC: } k: \text{key} \rightarrow \text{plain} \rightarrow c: \text{cipher}\{\text{Encrypted}(k,c)\} \\
  \text{val } \text{DEC: } k: \text{key} \rightarrow c: \text{cipher}\{\text{Encrypted}(k,c)\} \rightarrow \text{plain}
  \]

• With **authenticated encryption**, we have a **stronger** ideal interface that ensure plaintext integrity (much as MACs)

  \[
  \text{predicate } \text{Msg} \text{ of } \text{key} \times \text{plain} \text{ // defined by protocol} \\
  \text{val } \text{ENC: } k: \text{key} \rightarrow p: \text{plain}\{\text{Msg}(k,p)\} \rightarrow \text{cipher} \\
  \text{val } \text{DEC: } k: \text{key} \rightarrow \text{cipher} \rightarrow p: \text{plain}\{\text{Msg}(k,p)\} \text{ option}
  \]
Modular Architecture for miTLS

Base components:
- CoreCrypto
- Bytes
- TCP
- TLSConstants
- TLSInfo
- Error
- Range

Handshake/CCS:
- Sig
- RSAKey
- DHGroup
- Cert
- CRE
- RAS
- DH
- SessionDB
- Handshake (and CCS)

Alert Protocol:
- Alert

AppData Protocol:
- Datastream
- AppData

TLS API:
- TLS

TLS Record:
- MAC
- Encode
- Enc
- LHAEPoM
- LHAE
- StPPlain
- StAIE
- TLSFragment
- Record

Application:
- AuthPlain
- Auth
- RPCPlain
- RPC

Adversary:
- Untyped API
- Untyped Adversary
our main
TLS API
(outline)

Each application provides
its own plaintext module
for data streams:
• Typing ensures secrecy and authenticity at safe indexes
Each application creates and runs session & connections in parallel
• Parameters select ciphersuites and certificates
• Results provide detailed information on the protocol state

```scala
type cn // for each local instance of the protocol

// creating new client and server instances
val connect: TcpStream -> params -> (;Client) nullCn Result
val accept: TcpStream -> params -> (;Server) nullCn Result

// triggering new handshakes, and closing connections
val rehandshake: c:cn{Role(c)=Client} -> cn Result
val request: c:cn{Role(c)=Server} -> cn Result
val shutdown: c:cn -> TcpStream Result

// writing data
type (;c:cn,data:(;c) msg_o) iresult_o =
  | WriteComplete of c':cn
  | WritePartial of c':cn * rest:(;c') msg_o
  | MustRead of c':cn
val write: c:cn -> data:(;c) msg_o -> (;c,data) iresult_o

// reading data
type (;c:cn) iresult_i =
  | Read of c':cn * data:(;c) msg_i
  | CertQuery of c':cn
  | Handshake of c':cn
  | Close of TcpStream
  | Warning of c':cn * a:alertDescription
  | Fatal of a:alertDescription
val read : c:cn -> (;c) iresult_i
```
Main crypto result: concrete TLS & ideal TLS are computationally indistinguishable

We prove that ideal miTLS meets its secure channel specification using standard program verification (typing)
Final Thoughts

Many pitfalls in cryptographic software
• Need to verify their design+implementation
• Need to verify crypto+protocol+application

Formal security proofs for real-world crypto protocols are now feasible
• TLS 1.3 is an ongoing successful experiment
• Similar results for SSH, Signal, etc.
• Many tools: ProVerif, CryptoVerif, F*, Tamarin, EasyCrypt, VST
• Try them out to build your next proof, or to implement your crypto protocols securely!
End of Part IV