Dealing with Key Compromise in CryptoVerif

Bruno Blanchet

INRIA Paris
Bruno.Blanchet@inria.fr

January 2023

Outline

The computational model

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

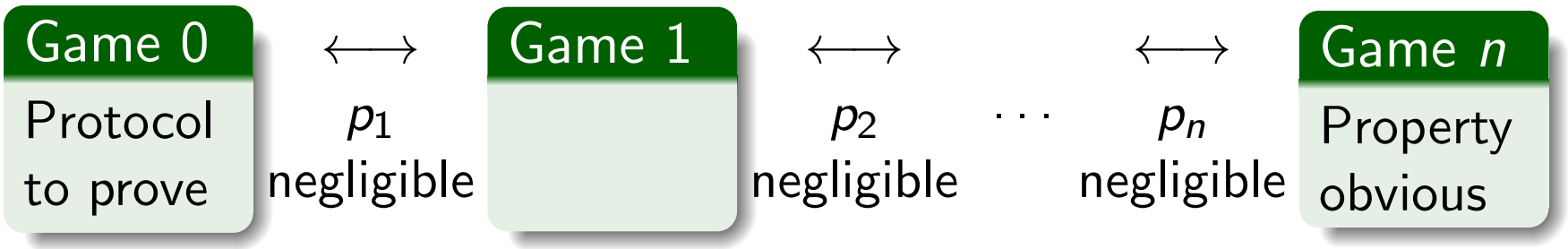
- Messages are bitstrings. 01100100
- Cryptographic primitives are functions on bitstrings.
  $\mathrm{enc}(011, 100100) = 111$
- The attacker is any probabilistic polynomial-time Turing machine.
  - The security assumptions on primitives specify what the attacker cannot do.

This model is more realistic than the symbolic model, but proofs are more difficult to mechanize.

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.)



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

CryptoVerif is a mechanized prover that:

- generates proofs by sequences of games.
- proves secrecy, correspondence, 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 automatic and interactive modes.

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.

Basic treatment of key compromise

Include the compromise in the specification of the cryptographic primitives themselves. Example: INT-CTXT with corruption.

```
1  new k : key; (
2     !^{i≤n} Oenc(x : cleartext) := new r : enc_seed; return(enc_r(x, k, r)) |
3     !^{i'≤n'} Odec(y : ciphertext) := return(dec(y, k)) |
4     Ocorrupt() := return(k))
5  ≈_{Advintctxt(time,n,n',maxlength(x),maxlength(y))}
6  new k : key; (
7     !^{i≤n} Oenc(x : cleartext) := new r : enc_seed;
8        let z : ciphertext = enc_r(x, k, r) in return(z) |
9     !^{i'≤n'} Odec(y : ciphertext) :=
10       if defined(corrupt) then return(dec(y, k)) else
11       find j ≤ n suchthat defined(x[j], z[j]) ∧ z[j] = y
12       then return(injbot(x[j])) else return(bottom) |
13    Ocorrupt() := let corrupt : bool = true in return(k)).
```

Applications

- INT-CTXT encryption in WireGuard [EuroS&P'19]
- one-wayness [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.

How to overcome these limitations?

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.

Proving secrecy

Suppose:

1. x is defined by an assignment $x[i] = z[M]$,
2. we want to prove the secrecy of x.

Old approach [TDSC'08]:

- Show that z and all variables computed using z are secret that is, they are not used in tests and output messages.

Proving secrecy

Suppose:

1. x is defined by an assignment $x[i] = z[M]$,
2. we want to prove the secrecy of x.

New approach:

- Show that the cells of z that are stored in x cannot be the same as those that are leaked (used in tests and output messages).

Proving secrecy

Suppose:

1. x is defined by an assignment $x[i] = z[M]$,
2. we want to prove the secrecy of x.

New approach:

- Show that the cells of z that are stored in x cannot be the same as those that are leaked (used in tests and output messages).

Advantages:

- Allows proving secrecy for a part of array z.
- Especially useful in the presence of key compromise.

Proving secrecy: details

Suppose:

1. x is defined by an assignment $x[i] = z[M]$,
2. we want to prove the secrecy of x.

Sketch of the procedure:

- Collect
  - facts that hold at the definition of x,
  - facts that hold when z leaks, that is, is used in a test or output, possible through assignments to other variables,
  - equality of indices of z in both cases.
- Derive a contradiction (possibly up to elimination of collisions).

Proving secrecy: toy example

$!^{i \leq n}\, \mathbf{in}(c[i], ()); \mathbf{new}\ k : key; \mathbf{out}(c[i], ());$
$\mathbf{in}(d[i], compr : bool);$
$\mathbf{if}\ compr\ \mathbf{then}$
$\quad \mathbf{out}(d[i], {}^{\mu_1} k)$
$\mathbf{else}$
$\quad \mathbf{let}\ s : key = {}^{\mu_2} k\ \mathbf{in}\ {}^{\mu_3}\mathbf{out}(d[i], ())$

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 combines **success** and **simplify**:

- **success** collects information known to be true when the adversary breaks at least one of the desired properties.
- **simplify** removes parts of the game that contradict this information and replaces them 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 they can be safely removed.

success simplify: canonical example

Suppose the active queries are **event**(e_i) $\Rightarrow$ **false** for events e_i executed by **event_abort** e_i.

Let $\mathcal{F}_\mu$ be facts that hold at program point μ.

Let μ_j for $j \in J$ be 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 **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 disting is executed.

new $k : key$; (
 $!^{i \leq n}$ Oenc(x : *cleartext*) := **new** r : *enc_seed*;
  let z : *ciphertext* = enc_r(x, k, r) **in return**(z) |
 $!^{i' \leq n'}$ Odec(y : *ciphertext*) :=
  if defined(*corrupt*) **then return**(dec(y, k)) **else**
  find $j \leq n$ **suchthat defined**($x[j], z[j]$) $\wedge$ $z[j] = y$
  then return(injbot($x[j]$)) **else**
  if *dec*(y, k) <> *bottom* **then** $^{\mu}$**event_abort** disting
  else return(bottom) |
 Ocorrupt() := **let** *corrupt* : *bool* = true **in** $^{\mu_1}$**return**(k)).

success simplify: example

new $k : key$; (
$!^{i \leq n}$ Oenc($x : cleartext$) := **new** $r : enc\_seed$;
let $z : ciphertext = \text{enc\_r}(x, k, r)$ **in return**(z) |
$!^{i' \leq n'}$ Odec($y : ciphertext$) :=
if defined($corrupt$) **then return**(dec(y, k)) **else**
find $j \leq n$ **suchthat defined**($x[j], z[j]$) $\wedge$ $z[j] = y$
then return(injbot($x[j]$)) **else**
if $dec(y, k) <> bottom$ **then** $^{\mu}$**event_abort** disting
else return(bottom) |
Ocorrupt() := **let** $corrupt : bool = \text{true}$ **in** $^{\mu_1}$**return**(k)).

$\mathcal{F}_{\mu} \cup \mathcal{F}_{\mu_1}$ yields a contradiction, so **success simplify** replaces the code at μ_1 with **event_abort** adv_loses.

In the transformed game, k is never corrupted, so we can apply the standard ciphertext integrity assumption without corruption to bound the probability of disting and conclude.

guess: guess the tested session

Guess a replication index: transform

$$!^{i \leq n}\, \mathbf{in}(c, x : T); P$$

into

$$!^{i \leq n}\, \mathbf{in}(c, x : T); \mathbf{if}\ i = i_{\text{tested}}\ \mathbf{then}\ P'\ \mathbf{else}\ P''$$

1. P' is the tested session. P' is obtained from P by
   - replacing **event** $e(M)$ with **event** $e(M)$; **event** $e'(M)$.
   - replacing **let** $x = M$ **in** with **let** $x = M$ **in let** $x' = x$ **in** when x is used in (one-session) secrecy queries.
2. P'' represents the other sessions. P'' is obtained from P by
   - replacing **let** $x = M$ **in** with **let** $x = M$ **in let** $x'' = x$ **in** when x is used in secrecy queries.

The same event e (resp. definition of the same variable x) cannot occur both under the modified replication $!^{i \leq n}$ and elsewhere in the game.

guess: guess the tested session

Update of queries: prove queries in the tested session.

secret x [one_session]	$\rightsquigarrow$	**secret** x' [one_session]
secret x	$\rightsquigarrow$	**secret** x' **public_vars** x''
event$(e(M)) \wedge \psi \Rightarrow \phi$	$\rightsquigarrow$	**event**$(e'(M)) \wedge \psi \Rightarrow \phi$

Does not work for injective correspondences! (see next)

Probabilities multiplied by n for modified queries.

guess: injective correspondences

We cannot modify correspondence queries with injective events:

- Counter-example:

  $\forall i : [1, n], x : T'; \mathbf{event}(e_1(i, x)) \wedge \mathbf{inj\text{-}event}(e_2(x)) \Rightarrow \mathbf{inj\text{-}event}(e_3())$

  with events

  $$e_3 \quad e_1(i_1, x_1) \quad e_1(i_2, x_2) \quad e_2(x_1) \quad e_2(x_2)$$

  The query is false, but it is true if we restrict ourselves to one value of i (the index of the tested session), because we consider
  - $e_1(i_1, x_1)$, $e_2(x_1)$ and e_3 for $i = i_1$ and
  - $e_1(i_2, x_2)$, $e_2(x_2)$ and e_3 for $i = i_2$.

Solution:

1. Show that the non-injective version of the correspondence implies the injective version, in the current game.
2. Continue with the non-injective version of the correspondence.

guess: example

$$B \to A: \quad \{na\}_{pkA}$$
$$A \to B: \quad na$$

Role of B:

$$!^{i_B \leq n_B}\, \mathbf{in}(c3[i_B], ()); \mathbf{new}\ na : nonce; \mathbf{out}(c4[i_B], \mathsf{enc}(\mathsf{pad}(na), pkA));$$
$$\mathbf{in}(c5[i_B], = na); \mathbf{event}\ e_B(na)$$

Show the correspondence

$$\forall x : nonce; \mathbf{event}(e_B(x)) \Rightarrow \mathbf{event}(e_A(x))$$

guess: example (continued)

Apply the IND-CCA2 assumption on encryption

- replaces the encryption of na with the encryption of a 0 block Zb,
- adapts the decryption accordingly in A.

Role of B:

$$!^{i_B \leq n_B}\, \mathbf{in}(c3[i_B], ()); \mathbf{new}\ na : nonce; \mathbf{out}(c4[i_B], \mathrm{enc}(Zb, pkA));$$
$$\mathbf{in}(c5[i_B], = na); \mathbf{event}\ e_B(na)$$

guess: example (continued)

guess i_B.

Role of B:

$$
\begin{array}{l}
!^{i_B \leq n_B}\, \mathbf{in}(c3[i_B], ()); \\
\mathbf{if}\ i_B = i_{B\mathrm{tested}}\ \mathbf{then} \\
\quad \mathbf{new}\ na : nonce; \mathbf{out}(c4[i_B], \mathrm{enc}(Zb, pkA)); \\
\quad \mathbf{in}(c5[i_B], = na); \mathbf{event}\ e_B(na); \mathbf{event}\ e'_B(na) \\
\mathbf{else} \\
\quad \mathbf{new}\ na : nonce; \mathbf{out}(c4[i_B], \mathrm{enc}(Zb, pkA)); \\
\quad \mathbf{in}(c5[i_B], = na); \mathbf{event}\ e_B(na)
\end{array}
$$

Show the correspondence $\forall x : nonce; \mathbf{event}(e'_B(x)) \Rightarrow \mathbf{event}(e_A(x))$

guess: example (continued)

SArename *na*: distinguish whether the nonce *na* has been generated in the tested session or not.

Role of B:

$$!^{i_B \leq n_B}\ \mathbf{in}(c3[i_B], ());$$
$$\mathbf{if}\ i_B = i_{B\mathrm{tested}}\ \mathbf{then}$$
$$\quad \mathbf{new}\ na_3 : nonce; \mathbf{out}(c4[i_B], \mathrm{enc}(Zb, pkA));$$
$$\quad \mathbf{in}(c5[i_B], = na_3); \mathbf{event}\ e'_B(na_3)$$
$$\mathbf{else}$$
$$\quad \mathbf{new}\ na_2 : nonce; \mathbf{out}(c4[i_B], \mathrm{enc}(Zb, pkA));$$
$$\quad \mathbf{in}(c5[i_B], = na_2); \mathbf{event}\ e_B(na_2)$$

guess: example (continued)

Insert a **find** just before e'_B that tests whether $e_A(na_3)$ has been executed.

Role of B:

$!^{i_B \leq n_B}$ **in**$(c3[i_B], ())$;
if $i_B = i_{B\text{tested}}$ **then**
  new $na_3 : nonce$; **out**$(c4[i_B], \text{enc}(Zb, pkA))$; **in**$(c5[i_B], = na_3)$;
  find $j \leq n_A$ **suchthat defined**$(eAex[j])$ **then event** $e'_B(na_3)$
    else event_abort bad
else
  new $na_2 : nonce$; **out**$(c4[i_B], \text{enc}(Zb, pkA))$;
  in$(c5[i_B], = na_2)$; **event** $e_B(na_2)$

$\forall x : nonce$; **event**$(e'_B(x)) \Rightarrow$ **event**$(e_A(x))$ is proved.
event(bad) $\Rightarrow$ **false** remains to be proved.

guess: example (continued)

success simplify removes the output of na_3 in A. (When na_3 is sent, $e_A(na_3)$ has been executed, so bad will not be executed.)

A dependency analysis on na_3 shows that the adversary has no information on na_3:

- the input $\mathbf{in}(c5[i_B], = na_3)$ has little probability of succeeding;
- the code that follows it can be removed;
- that removes event bad and concludes the proof.

guess: other variants

- Extension to guessing a sequence of replication indices
- Guess the value of a variable
  - when its type is not too large;
  - loses a factor equal to the cardinal of the type.
- Guess the branch taken in a test.

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.
- Grouping compromise scenarios in WireGuard, by guessing which branch is taken.

Conclusion

We implemented several extensions of CryptoVerif:

1. Improvement of the proof of secrecy.
2. New commands: **focus**, **success simplify**, **guess**.

useful for dealing with the compromise of keys, but that have more general applications.

Work in progress and future work

1. CV2EC
2. CV2F*
3. papers on
   - collecting information in games,
   - **crypto** transformation.