Dealing with Dynamic Key Compromise in CryptoVerif

Bruno Blanchet

Inria, Paris
Bruno.Blanchet@inria.fr

July 2024
The computational model

CryptoVerif relies on the computational model of cryptography, the standard model used by cryptographers:

- Messages are bitstrings.
- Cryptographic primitives are functions on bitstrings.
- The attacker is a probabilistic Turing machine.
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 using a security assumption on 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.)
CrytoVerif is a mechanized prover that:

- generates proofs by sequences of games.
- proves secrecy, authentication, and indistinguishability properties.
- provides a generic method for specifying assumptions on 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 automatic and interactive modes.
Basic treatment of key compromise

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

1. \( k \leftarrow R \text{key}; ( \)
2. \( !_{i \leq n}^O \text{Enc}(x[i] : \text{cleartext}) := \text{return}(\text{enc}(x[i], k)) | \)
3. \( !_{i' \leq n'}^O \text{Dec}(y : \text{ciphertext}) := \text{return}(\text{dec}(y, k)) \)
4. \( ) \)
5. \( \approx \)
6. \( k \leftarrow R \text{key}; ( \)
7. \( !_{i \leq n}^O \text{Enc}(x[i] : \text{cleartext}) := \text{let } z[i] = \text{enc}(x[i], k) \text{ in return}(z[i]) | \)
8. \( !_{i' \leq n'}^O \text{Dec}(y : \text{ciphertext}) := \)
9. \( \)  
10. \( \text{find } j \leq n \text{ suchthat defined}(x[j], z[j]) \land z[j] = y \)
11. \( \text{then return}(x[j]) \text{ else return}(\bot) \)
12. \( ) \)
Basic treatment of key compromise

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

\begin{verbatim}
1  k \xleftarrow{R} \text{key}; \langle
2  \forall i \leq n \text{Oenc}(x[i] : \text{cleartext}) := \text{return}(\text{enc}(x[i], k)) |
3  \forall i' \leq n' \text{Odec}(y : \text{ciphertext}) := \text{return}(\text{dec}(y, k)) |
4  \text{Ocorrupt()} := \text{return}(k)
5  \approx

6  k \xleftarrow{R} \text{key}; \langle
7  \forall i \leq n \text{Oenc}(x[i] : \text{cleartext}) := \text{let } z[i] = \text{enc}(x[i], k) \text{ in return}(z[i]) |
8  \forall i' \leq n' \text{Odec}(y : \text{ciphertext}) :=
9       \text{if defined}(\text{corrupt}) \text{ then return}(\text{dec}(y, k)) \text{ else}
10      \text{find } j \leq n \text{ such that defined}(x[j], z[j]) \land z[j] = y
11      \text{then return}(x[j]) \text{ else return}(\bot) |
12  \text{Ocorrupt()} := \text{let corrupt = true in return}(k)
\end{verbatim}
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.
New in this paper

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

Suppose:

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

Old approach [TDSC’08]:

- Show that $k$ and all variables computed using $k$ are secret, that is, they are not used in tests and output messages.
- Conclude that $s$ is secret.
Suppose:
1. $s$ is defined by an assignment $s[i] = k[M],$
2. we want to prove the secrecy of $s.$

New approach:
- Show that the cells of $k$ that are stored in $s$ cannot be the same as those that are leaked (used in tests and output messages).
Suppose:

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

New approach:

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

Advantages:

- Allows proving secrecy for a part of array $k.$
- Especially useful in the presence of key compromise.
Suppose:

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

Sketch of the procedure:

- Collect
  - facts that hold at the definition of $s,$
  - facts that hold when $k$ leaks, that is, is used in a test or output, possible through assignments to other variables,
  - equality of indices of $k$ in both cases.

- Derive a contradiction (possibly up to elimination of collisions).
Proving secrecy: toy example

\[ !i \leq n \quad \text{O1()} := k[i] \xleftarrow{R} \text{key}; \text{return}(); \]

\[ \text{O2(compr[i] : bool)} := \]

\[ \text{if compr[i] then} \]

\[ \text{return(k[i])} \]

\[ \text{else} \]

\[ \text{let s[i] = k[i] in return()} \]

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 $q_1, \ldots, q_m$ 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

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 $\text{event}(e_i) \Rightarrow \text{false}$ for events $e_i$ executed by $\text{event\_abort} \ e_i$;

- $\mathcal{F}_\mu$ are facts that hold at program point $\mu$;

- $\mu_j$ 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 $\mu$ is executed, then $\mu_j$ cannot be executed, so the adversary loses

- success simplify replaces the code at $\mu$ with $\text{event\_abort}$ 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.

\[
k \xleftarrow{\ramdom} \text{key}; \ \\
\begin{align*}
^{i \leq n} \text{Oenc}(x[i] : \text{cleartext}) & := \text{let } z[i] = \text{enc}(x[i], k) \text{ in return}(z[i]) | \\
^{i' \leq n'} \text{Odec}(y : \text{ciphertext}) & := \\
\text{if defined}(\text{corrupt}) & \text{ then return}(\text{dec}(y, k)) \text{ else } \\
\text{find } j \leq n & \text{ such that defined}(x[j], z[j]) \land z[j] = y \text{ then return}(x[j]) \text{ else } \\
\text{if } \text{dec}(y, k) & \neq \bot \text{ then } \mu \text{event_abort distinguish} \\
\text{else return}(\bot) | \\
\text{Ocorrupt()} & := \text{let corrupt} = \text{true in } \mu^1 \text{return}(k)).
\end{align*}
\]
success simplify: example

\[ k \leftarrow^R \text{key}; (\]
\[ \begin{align*}
!i \leq n \quad & \text{Oenc}(x[i] : \text{cleartext}) := \text{let } z[i] = \text{enc}(x[i], k) \text{ in return}(z[i]) | \\
!i' \leq n' \quad & \text{Odec}(y : \text{ciphertext}) := \\
& \quad \text{if } \text{defined}(\text{corrupt}) \text{ then return}(\text{dec}(y, k)) \text{ else} \\
& \quad \text{find } j \leq n \text{ suchthat } \text{defined}(x[j], z[j]) \land z[j] = y \\
& \quad \text{then return}(x[j]) \text{ else} \\
& \quad \text{if } \text{dec}(y, k) \neq \bot \text{ then } \mu \text{event\_abort distinguish} \\
& \quad \text{else return}(\bot) | \\
\end{align*}
\]
\[ \text{Ocorrupt()} := \text{let } \text{corrupt} = \text{true in } \mu_1 \text{return}(k)). \]

- \( \mathcal{F}_\mu \cup \mathcal{F}_{\mu_1} \) yields a contradiction;
- **success simplify** replaces code at \( \mu_1 \) 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 $\text{event}(e_i) \Rightarrow \text{false}$.
3. **success simplify** removes the compromise of the key.
4. We prove queries $\text{event}(e_i) \Rightarrow \text{false}$.
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.
Conclusion

We implemented several extensions of CryptoVerif:

1. Improvement of the proof of secrecy.
2. New commands: focus, success simplify, guess the tested session, guess the value of a variable, guess the branch taken in a test.

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