CV2EC: Getting the Best of Both Worlds

Bruno Blanchet\textsuperscript{1}  Pierre Boutry\textsuperscript{2,1}  Christian Doczkal\textsuperscript{3}
Benjamin Grégoire\textsuperscript{2,1}  Pierre-Yves Strub\textsuperscript{4}

\textsuperscript{1}Inria  \textsuperscript{2}Université Côte d'Azur  \textsuperscript{3}MPI-SP  \textsuperscript{4}PQShield

July 2024
CSF’24, Enschede, The Netherlands
Overview

CryptoVerif and EasyCrypt both mechanize game-based cryptographic proofs:

- proofs in the computational model,
- like the manual proofs of cryptographers.

Game 0
Protocol to prove
\[ \Leftrightarrow \]
\[ p_1 \text{ negligible} \]

Game 1
\[ \Leftrightarrow \]
\[ p_2 \text{ negligible} \]
\[ \Leftrightarrow \]
\[ \cdots \]
\[ \Leftrightarrow \]
\[ p_n \text{ negligible} \]

Game \( n \)
Property obvious
<table>
<thead>
<tr>
<th>CryptoVerif</th>
<th>EasyCrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ highly automated</td>
<td>− less automated</td>
</tr>
<tr>
<td>+ works well for protocols</td>
<td>− too verbose for large protocols</td>
</tr>
<tr>
<td>− “non-standard” formulations of assumptions</td>
<td>+ any assumptions</td>
</tr>
<tr>
<td>− no complex reductions (e.g., hybrid proofs)</td>
<td>+ arbitrary reductions</td>
</tr>
</tbody>
</table>

⇒ use CryptoVerif for proving protocols from assumptions on primitives and EasyCrypt for proving those assumptions on primitives.

Solution: CV2EC

Automatically translate the “non-standard” assumption of CV to EC, and (manually) reduce them to “standard” security assumptions in EC.
Overview

CryptoVerif (CV) and EasyCrypt (EC) make different tradeoffs:

<table>
<thead>
<tr>
<th>CryptoVerif</th>
<th>EasyCrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ highly automated</td>
<td>− less automated</td>
</tr>
<tr>
<td>+ works well for protocols</td>
<td>− too verbose for large protocols</td>
</tr>
<tr>
<td>− “non-standard” formulations of assumptions</td>
<td>+ any assumptions</td>
</tr>
<tr>
<td>− no complex reductions</td>
<td>+ arbitrary reductions</td>
</tr>
<tr>
<td>(e.g. hybrid proofs)</td>
<td></td>
</tr>
</tbody>
</table>

⇒ use CryptoVerif for proving protocols from assumptions on primitives and EasyCrypt for proving those assumptions on primitives
Overview

CryptoVerif (CV) and EasyCrypt (EC) make different tradeoffs:

<table>
<thead>
<tr>
<th>CryptoVerif</th>
<th>EasyCrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ highly automated</td>
<td>− less automated</td>
</tr>
<tr>
<td>+ works well for protocols</td>
<td>− too verbose for large protocols</td>
</tr>
<tr>
<td>− “non-standard” formulations of assumptions</td>
<td>+ any assumptions</td>
</tr>
<tr>
<td>− no complex reductions (e.g. hybrid proofs)</td>
<td>+ arbitrary reductions</td>
</tr>
</tbody>
</table>

⇒ use CryptoVerif for proving protocols from assumptions on primitives and EasyCrypt for proving those assumptions on primitives

Solution: CV2EC

*Automatically translate* the “non-standard” assumption of CV to EC, and (manually) reduce them to “standard” security assumptions in EC.
## CryptoVerif vs EasyCrypt

<table>
<thead>
<tr>
<th>CryptoVerif</th>
<th>EasyCrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based on oracles</td>
<td>Based on pWHILE + Hoare logic</td>
</tr>
<tr>
<td>Single-use oracles + replication</td>
<td>Multi-call oracle procedures</td>
</tr>
<tr>
<td>All variables are global (arrays indexed by replication indices)</td>
<td>Global memory + local variables</td>
</tr>
<tr>
<td>Games in “Real/Ideal” style</td>
<td>Can express arbitrary games</td>
</tr>
<tr>
<td>Implicit adversary</td>
<td>Explicit adversary</td>
</tr>
</tbody>
</table>
### CryptoVerif vs EasyCrypt

<table>
<thead>
<tr>
<th>CryptoVerif</th>
<th>EasyCrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based on oracles</td>
<td>Based on pWHILE + Hoare logic</td>
</tr>
<tr>
<td>Single-use oracles + replication</td>
<td>Multi-call oracle procedures</td>
</tr>
<tr>
<td>All variables are global</td>
<td>Global memory + local variables</td>
</tr>
<tr>
<td>(arrays indexed by replication indices)</td>
<td></td>
</tr>
<tr>
<td>Games in “Real/Ideal” style</td>
<td>Can express arbitrary games</td>
</tr>
<tr>
<td>Implicit adversary</td>
<td>Explicit adversary</td>
</tr>
</tbody>
</table>

- Running example: Real/Ideal formulation of IND-CCA2 assumption (Adversary tries to distinguish *honest* encryption oracle from encryption of *constant message*).
IND-CCA2 assumption in CryptoVerif (real game)

\[ s \leftarrow_R \text{keyseed}; \ ( \]
\[ \text{Opk}() := \text{return}(\text{pkgen}(s)) \]
\[ | \ \text{foreach } i \leq N \ \text{do } es \leftarrow_R \text{enc}\_seed; \]
\[ \text{Oenc}(m: \text{cleartext}) := \text{return}(\text{enc}(m, \text{pkgen}(s), es)) \]
\[ | \ \text{foreach } i_2 \leq N_2 \ \text{do } \]
\[ \text{Odec}(c: \text{ciphertext}) := \text{return}(\text{dec}(c, \text{skgen}(s))) \]

- sample secret key seed \( s \)
- provide one copy of the \( \text{Opk}() \) oracle
- provide \( N \) copies of the \( \text{Oenc}(m) \) oracle (each with some \( \text{enc}\_seed \))
- provide \( N_2 \) copies of the \( \text{Odec}(c) \) oracle
- All queries are answered faithfully
IND-CCA2 assumption in CryptoVerif (ideal game)

\[
s \leftarrow \text{R keyseed}; \quad (\text{Opk}() := \text{return}(\text{pkgen}(s)))
\]

| foreach \( i \leq N \) do \( es \leftarrow \text{R enc\_seed}; \)
| \( \text{Oenc}(m:\text{cleartext}) := \)
| \( c\text\_enc \leftarrow \text{enc}(\text{zero}(m), \text{pkgen}(s), es); \)
| \( \text{insert cipher}(m, c\text\_enc); \)
| \( \text{return}(c\text\_enc) \)
| foreach \( i2 \leq N2 \) do
| \( \text{Odec}(c:\text{ciphertext}) := \)
| \( \text{get cipher}(m\_dec, =c) \)
| \( \text{in return}(m\_dec) \)
| \( \text{else return}(\text{dec}(c, \text{skgen}(s))) \) \)

- same replication/oracle signature as real game
- \( \text{Oenc}(m) \) encrypts \( \text{zero}(m) \) (zero message of length \( |m| \)) stores the cleartext/ciphertext mapping in table \( \text{cipher} \)
- \( \text{Odec}(c) \) checks whether a previous call to \( \text{Oenc} \) returned \( c \), by searching table \( \text{cipher} \).
Implicit aspects of the CryptoVerif games

- Variables defined under `foreach` are implicitly arrays.
- Oracles defined under `foreach` implicitly have the index as argument.

```verbatim
definition 0
foreach i <= N do es[i] <-R enc_seed;
oenc[i](m[i]:cleartext) := return(enc(m[i], pkgen(s),es[i]))
```

Random number generations outside oracles can be called by the adversary as if they were separate oracles.

Random number generations must be executed before the oracle(s) under them can be called. Oracles can be called at most once for each index. The adversary trying to distinguish the games is implicit.
Implicit aspects of the CryptoVerif games

- Variables defined under `foreach` are implicitly arrays.
- Oracles defined under `foreach` implicitly have the index as argument.

```plaintext
foreach i <= N do es[i] <-R enc_seed;
Oenc[i](m[i]:cleartext) := return(enc(m[i], pkgen(s),es[i]))
```

- Random number generations outside oracles can be called by the adversary as if they were separate oracles.

```plaintext
O_s() := s <-R keyseed; return(); ( 
  Opk() := return(pkgen(s))
| foreach i <= N do O_es[i]() := es[i] <-R enc_seed; return();
  Oenc[i](m[i]:cleartext) := return(enc(m[i], pkgen(s),es[i]))
```

- Random number generations must be executed before the oracle(s) under them can be called.
Implicit aspects of the CryptoVerif games

- Variables defined under `foreach` are implicitly arrays.
- Oracles defined under `foreach` implicitly have the index as argument.

```plaintext
foreach i <= N do es[i] <-R enc_seed;
Oenc[i](m[i]:cleartext) := return(enc(m[i], pkgen(s), es[i]))
```

- Random number generations outside oracles can be called by the adversary as if they were separate oracles.

```plaintext
O_s() := s <-R keyseed; return();
O_pk() := return(pkgen(s))
| foreach i <= N do O_es[i]() := es[i] <-R enc_seed; return();
| Oenc[i](m[i]:cleartext) := return(enc(m[i], pkgen(s), es[i]))
```

- Random number generations must be executed before the oracle(s) under them can be called.
- Oracles can be called at most once for each index.
Implicit aspects of the CryptoVerif games

- Variables defined under `foreach` are implicitly arrays.
- Oracles defined under `foreach` implicitly have the index as argument.

```plaintext
foreach i <= N do es[i] <-R enc_seed;
Oenc[i](m[i]:cleartext) := return(enc(m[i], pkgen(s),es[i]))
```

- Random number generations outside oracles can be called by the adversary as if they were separate oracles.

```plaintext
O_s() := s <-R keyseed; return();
Opk() := return(pkgen(s))
| foreach i <= N do O_es[i]() := es[i] <-R enc_seed; return();
Oenc[i](m[i]:cleartext) := return(enc(m[i], pkgen(s),es[i]))
```

- Random number generations must be executed before the oracle(s) under them can be called.
- Oracles can be called at most once for each index.
- The adversary trying to distinguish the games is implicit.
<table>
<thead>
<tr>
<th>CryptoVerif</th>
<th>EasyCrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Oracle</strong></td>
<td><strong>Procedure</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Index added as argument</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Map from oracle index to non-index arguments</strong> (records that the oracle has been called)</td>
</tr>
<tr>
<td><strong>Random number generation</strong></td>
<td><strong>Random oracle</strong></td>
</tr>
<tr>
<td></td>
<td>(by cloning the random oracle theory)</td>
</tr>
<tr>
<td><strong>Other variables</strong></td>
<td><strong>Map from index to value</strong></td>
</tr>
<tr>
<td><strong>Table</strong></td>
<td><strong>Mutable list of tuples</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Explicit adversary</strong></td>
</tr>
</tbody>
</table>
IND-CCA2 game in EasyCrypt

module Game (O : Oracle, A : Adversary) = {
    proc main() = {
        O.init();
        r <@ A(O).distinguish();
        return r;
    }
}.

module type Oracle = {
    proc init() : unit
    proc s() : unit
    proc es() : unit
    proc Opk () : pkey
    proc Oenc (_ : cleartext) : ciphertext
    proc Odec (_ : ciphertext) : cleartext option
}. 

module type Adversary (O : Oracle) = {
    proc distinguish () : bool {O.s O.es O.Opk O.Oenc O.Odec}
}.
module LHS_ (O_s : OL_s.RO) (O_es : OL_es.RO) : Oracle = {
  // Maps for oracles
  var m_s : (unit, unit) fmap
  var m_es : (int, unit) fmap
  var m_Opk : (unit, unit) fmap
  var m_Oenc : (int, cleartext) fmap
  var m_Odec : (int, ciphertext) fmap

  proc init() = {
    // Initialization
    // Maps initially empty
    m_s <- empty; m_es <- empty;
    m_Opk <- empty; m_Oenc <- empty; m_Odec <- empty;
    // Initialize the random oracles
    O_s.init(); O_es.init();
  }

  proc s() = {
    // Procedure for s <-R keyseed
    if (() \notin m_s) {
      m_s.[](()) <- ();
      O_s.sample();
    }
  }
}
Real game in EasyCrypt

// Procedure for Oenc
proc Oenc(i : int, m : cleartext) = {
    var aout : ciphertext <- witness;
    var s : keyseed;
    var es : enc_seed;

    // Check that Oenc can be called
    if (1 <= i <= b_N \ i \in m_es \ i \notin m_Oenc) {
        m_Oenc.[i] <- m;
        // Compute encryption
        s <@ O_s.get();
        es <@ O_es.get(i);
        aout <- enc (m, pkgen (s), es);
    }
    return aout;
}
module RHS_ (O_s : OR_s.RO) (O_es : OR_es.RO) : Oracles = {
...
  var t_cipher : (cleartext * ciphertext) list

proc init() : unit = {
...
  t_cipher <- []; }

proc Oenc(i : int, m : cleartext) = {
...
  if (1 <= i <= b_N \ i \in m_es \ i \notin m_Oenc) {
    m_Oenc.[i] <- m;
    s <@ O_s.get();
    es <@ O_es.get(i);
    c_enc.[i] <- enc (zero (m), pkgen (s), es);
    t_cipher <- (m, oget c_enc.[i]) :: t_cipher;
    aout <- oget c_enc.[i];
  }
  return aout;
}
Ideal game in EasyCrypt

\[
\text{proc } \text{Odec}(\text{i}2 : \text{int}, \text{c} : \text{ciphertext}) = \{ \\
... \\
\text{if } (1 \leq i2 \leq b_N2 \land () \in m_s \land i2 \notin m_{Odec}) \{ \\
\text{m}_{Odec}[i2] \leftarrow c; \\
// \text{get cipher(m}_{dec}. =c) ... \\
\text{r_cipher} \leftarrow \text{List.pmap (fun row: cleartext } * \text{ ciphertext } => \\
\text{if } (\text{row}.\text{'}2 = \text{c}) \text{ then Some row.}\text{'}1 \text{ else None) } \text{t_cipher}; \\
\text{if } (\text{r_cipher} = []) \{ \\
\text{s} \leftarrow \text{O_s.get();} \\
\text{aout} \leftarrow \text{dec (c, skgen (s));} \\
\} \text{ else } \{} \\
\text{m1} \leftarrow \text{drat r_cipher;} \\
\text{m}_{dec}[i2] \leftarrow m1; \\
\text{aout} \leftarrow \text{oget m}_{dec}[i2]; \\
\} \\
\} \\
\text{return aout;}
\}
EasyCrypt proof

1. Simplify the generated games
   ▶ Sample \texttt{keyseed} in initialization
   ▶ Sample \texttt{enc.seed} in \texttt{Oenc}
     ★ Eager/Lazy arguments to move sampling
   ▶ Remove unnecessary maps

2. Reduce real/ideal EC games to standard assumptions (hybrid arguments, etc.)
Extensions

- In CV, **find** looks for indices of arrays that satisfy certain conditions.
  - Encoded in EC by searching the maps corresponding to arrays.
- CV allows to combine indistinguishability with up-to-bad reasoning in one step.
  - “bad” cases represented by events.
  - Encoded in EC by setting a variable when an event occurs.
- Details and further extensions in the paper.
Case studies

- **IND-CCA2:**
  - reduction to single challenge query

- **Outsider-CCA for Authenticated KEMs:**
  - reduction from $n$ users and many encap/decap queries to 2 users and single challenge query.
  - fix errors in manual proof:
    - counting error for encap queries
    - need to eliminate public key collisions

- **Computational Diffie–Hellmann (CDH) and Gap Diffie–Hellmann (GDH) for nominal groups:**
  - random self-reducibility (from many exponents to one)
  - nominal groups include standard prime-order groups as well as Curve25519/448
Conclusion

- We translate the CryptoVerif language for assumptions on primitives to EasyCrypt.
  - Prove protocols in CryptoVerif and primitives in EasyCrypt.
  - We trust the translation (as well as CryptoVerif and EasyCrypt).
  - Not translated yet: pattern-matching in get (not used in examples).

- Future work:
  - Translate several related CryptoVerif assumptions together.
  - Translate probability formulas.
  - Improve the performance of EasyCrypt.
  - Translate the full CryptoVerif language of games.