SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq

Carmine Abate\textsuperscript{1} Philipp G. Haselwarter\textsuperscript{2} Exequiel Rivas\textsuperscript{3} Antoine Van Muylder\textsuperscript{4} Théo Winterhalter\textsuperscript{1} Cătălin Hrițcu\textsuperscript{1} Kenji Maillard\textsuperscript{5} Bas Spitters\textsuperscript{2}

34\textsuperscript{th} Computer Security Foundation Symposium, 24.06.2021

This work was in part supported by the European Research Council under ERC Starting Grant SECOMP (715753), by AFOSR grant Homotopy type theory and probabilistic computation (12595060), and by the Concordium Blockchain Research Center at Aarhus University. Antoine Van Muylder holds a PhD Fellowship from the Research Foundation – Flanders (FWO).

\textsuperscript{1}MPI-SP \textsuperscript{2}Aarhus University \textsuperscript{3}Inria Paris \textsuperscript{4}Vrije Universiteit Brussel \textsuperscript{5}Inria Rennes
Motivation:

- Monolithic game-based proofs can become intractable
- State-Separating Proofs (SSP) from high-level structure of miTLS paper proofs (Brzuska, Delignat-Lavaud, Fournet, Kohbrok, Kohlweiss; 2018)
Why SSProve?

Motivation:

- Monolithic game-based proofs can become intractable
- State-Separating Proofs (SSP) from high-level structure of miTLS paper proofs (Brzuska, Delignat-Lavaud, Fournet, Kohbrok, Kohlweiss; 2018)

Contributions:

- Give precise meaning to SSP and formalise it in Coq prover
- Modular language, logic & semantics
- Theorem connecting high-level SSP arguments and low-level program logic
- Approach validated by formalising several examples
  - This paper: PRF, ElGamal. Github: KEM-DEM, Σ-protocols (with N. Sidorenco).
Why SSProve?

Motivation:

  Proposal: Game-playing proofs
- Monolithic game-based proofs can become intractable
- *State-Separating Proofs* (SSP) from high-level structure of miTLS paper proofs
  (Brzuska, Delignat-Lavaud, Fournet, Kohbrok, Kohlweiss; 2018)

Contributions:

- Give precise meaning to SSP and formalise it in Coq prover
- Modular language, logic & semantics
- Theorem connecting high-level SSP arguments and low-level program logic
- Approach validated by formalising several examples
  This paper: PRF, ElGamal. Github: KEM-DEM, Σ-protocols (with N. Sidorenco).
Requirements: IND-CPA security for PRF based encryption

To prove...

<table>
<thead>
<tr>
<th>Package: IND-CPA⁰</th>
<th>Package: IND-CPA¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem: key : option KEY</td>
<td>Mem: key : option KEY</td>
</tr>
<tr>
<td>ENC(msg):</td>
<td>ENC(msg):</td>
</tr>
<tr>
<td>if key = ⊥ then</td>
<td>if key = ⊥ then</td>
</tr>
<tr>
<td>key &lt;$&gt; uniform {0,1}ⁿ</td>
<td>key &lt;$&gt; uniform {0,1}ⁿ</td>
</tr>
<tr>
<td>(r,c) ← enc(key, msg)</td>
<td>msg_rnd &lt;$&gt; uniform {0,1}ⁿ</td>
</tr>
<tr>
<td>return (r,c)</td>
<td>(r,c) ← enc(key, msg_rnd)</td>
</tr>
<tr>
<td>return (r,c)</td>
<td></td>
</tr>
</tbody>
</table>

We need to...

1. pick a proof assistant
2. define a core language (syntax, semantics)
3. prove code-level reasoning principles (pRHL)
4. define packages, package composition
5. define games, adversaries, and security
6. prove high-level reasoning principles (SSP)
Requirements: IND-CPA security for PRF based encryption

To prove...

\[
\text{package: IND-CPA}^0 \\
\text{mem: key : option KEY}
\]

\[
\begin{align*}
\text{ENC}(\text{msg}) & : \\
\text{if key } = \bot \text{ then} \\
\text{key } \sim \text{uniform } \{0,1\}^n \\
(r,c) & \leftarrow \text{enc}(\text{key}, \text{msg}) \\
\text{return } (r,c)
\end{align*}
\]

\[
\text{package: IND-CPA}^1 \\
\text{mem: key : option KEY}
\]

\[
\begin{align*}
\text{ENC}(\text{msg}) & : \\
\text{if key } = \bot \text{ then} \\
\text{key } \sim \text{uniform } \{0,1\}^n \\
\text{msg_rnd} & \sim \text{uniform } \{0,1\}^n \\
(r,c) & \leftarrow \text{enc}(\text{key}, \text{msg_rnd}) \\
\text{return } (r,c)
\end{align*}
\]

We need to...

1. pick a proof assistant
2. define a core language (syntax, semantics)
3. prove code-level reasoning principles (pRHL)
4. define packages, package composition
5. define games, adversaries, and security
6. prove high-level reasoning principles (SSP)
Coq – a mature formal proof management system

Provides a formal language for
• mathematical definitions & theorems
• executable algorithms (pure, i.e. no state/probabilities etc)

Example libraries
• computer science: CompCert (C compiler), Verified Software Toolchain (verification of C programs), Fiat-Crypto (fast cryptographic primitives)
• mathematics: 4 colour theorem, Feit-Thompson theorem, real analysis

Architecture
• trusted code base = clearly delimited kernel
• tactic language for programming automation
• easy installation via package manager
SSProve/Core language

package: IND-CPA⁰
mem: key : option KEY

ENC(msg):
  if !key == ⊥ then
    k <$> uniform {0,1}ⁿ
    key := k
  (r,c) ← enc(key, msg)
  return (r,c)

!ℓ read from memory location ℓ
ℓ := v write v to memory location ℓ
x <$> D sample from (sub-) distribution D
x ← p(a) call imported procedure p on value a
c₁ ; c₂ sequencing (omitted at end of line)
return v embed v from Coq’s ambient algorithm. language
package: IND-CPA

mem: key : option KEY

ENC(msg):
if !key == ⊥ then
    k <$> uniform {0,1}^n
    key := k
(r,c) ← enc(key, msg)
return (r,c)

!ℓ read from memory location ℓ
ℓ := v write v to memory location ℓ
x <$> D sample from (sub-) distribution D
x ← p(a) call imported procedure p on value a
c_1 ; c_2 sequencing (omitted at end of line)
return v embed v from Coq’s ambient algorithm language

Under the hood, in Coq:

Inductive code A = ret (x : A) | call (p : op) (x : src p) (κ : tgt p → code A) | ...
SSProve/Core language

package: IND-CPA^0
mem: key : option KEY

ENC(msg):
  if !key == ⊥ then
    k <$ uniform {0,1}^n
    key := k
  (r,c) ← enc(key, msg)
  return (r,c)

!ℓ read from memory location ℓ
ℓ := v write v to memory location ℓ
x <$ D sample from (sub-) distribution D
x ← p(a) call imported procedure p on value a
c_1 ; c_2 sequencing (omitted at end of line)
return v embed v from Coq’s ambient algorithm. language

Under the hood, in Coq:

\[ \text{Inductive code A} = \text{ret} (x : A) \mid \text{call} (p : \text{op}) (x : \text{src} p) (κ : \text{tgt} p \rightarrow \text{code} A) \mid \ldots \]

Derived definitions:

\[
\begin{align*}
\text{assert (b): } & \quad \text{if } b == \text{false} \text{ then } \\
& \quad \text{BOOM <$ null distr } \{0,1\} \\
& \quad \text{return } \text{BOOM}
\end{align*}
\]

\[
\begin{align*}
\text{for (n, c): } & \quad \text{if } n > 0 \text{ then } \\
& \quad \text{c n} \\
& \quad \text{for (n-1, c)}
\end{align*}
\]
Each rule is a theorem in Coq.

Details on semantics: Antoine Van Muylder’s video presentation on SSProve at TYPES 2021.
SSProve/Packages

<table>
<thead>
<tr>
<th>Package: IND-CPA⁰</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem: key : option KEY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ENC(msg):</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ! key == ⊥ then k &lt;$&gt; uniform {0,1}ⁿ</td>
</tr>
<tr>
<td>key := k</td>
</tr>
<tr>
<td>(r,c) ← enc(key, msg)</td>
</tr>
<tr>
<td>return (r,c)</td>
</tr>
</tbody>
</table>

package | a collection of typed procedure implementations with shared state |
interface | set of (typed) locations, 2 collections of (typed) procedure names: imports & exports |
seq. comp. | $P_1 \circ P_2$ |
prerequisites | inlining: replace call to imported procedure $x \leftarrow f(a)$ in $P_1$ with $x \leftarrow P_2.f(a)$ |
par. comp. | $P_1 \parallel P_2$ |
prerequisites | union of implementations |
prerequisites | no clashing procedure names |
package IND-CPA

mem: key : option KEY

ENC(msg):
if !key == ⊥ then
  k <$ uniform {0,1}^n
  key := k
  (r,c) ← enc(key, msg)
return (r,c)

package a collection of typed
procedure implementations with shared state

interface set of (typed) locations, 2 collections of
(typed) procedure names: imports & exports

seq. comp. \( P_1 \circ P_2 \)
inlining: replace call to imported procedure
\( x \leftarrow f(a) \) in \( P_1 \) with \( x \leftarrow P_2.f(a) \)

prerequisites provide all imports. No requirement about state!

par. comp. \( P_1 \parallel P_2 \)
union of implementations

prerequisites no clashing procedure names

Laws:

\[
\begin{align*}
    P_1 \circ (P_2 \circ P_3) &= (P_1 \circ P_2) \circ P_3 \\
    P_1 \parallel P_2 &= P_2 \parallel P_1 \\
    P_1 \parallel (P_2 \parallel P_3) &= (P_1 \parallel P_2) \parallel P_3 \\
    (P_1 \circ P_3) \parallel (P_2 \circ P_4) &= (P_1 \parallel P_2) \circ (P_3 \parallel P_4)
\end{align*}
\]
Games, Adversaries, Indistinguishability

- **Game**: a package with no imports
- **Game pair**: two games with the same exports
- **Adversary** $A$ for game $G$: package compatible with $G$ with separate state exporting one procedure
  
  $A$.run : unit $→$ bool

  intuitive meaning: guess which game $A$ is interacting with
Games, Adversaries, Indistinguishability

- **Game**: a package with no imports
- **Game pair**: two games with the same exports
- **Adversary** $\mathcal{A}$ for game $G$: package compatible with $G$ with separate state exporting one procedure
  \[ \mathcal{A}.\text{run} : \text{unit} \rightarrow \text{bool} \]
  Intuitive meaning: guess which game $\mathcal{A}$ is interacting with
- **Advantage** of $\mathcal{A}$ against a game pair $(G_0, G_1)$:
  \[ \alpha_{(G_0, G_1)}(\mathcal{A}) = |\Pr[\text{true} \leftarrow (\mathcal{A} \circ G_0).\text{run()}]) - \Pr[\text{true} \leftarrow (\mathcal{A} \circ G_1).\text{run()}]| \]
- Perfect **indistinguishability** $G_0 \approx G_1$:
  \[ \forall \mathcal{A}. \alpha_{(G_0, G_1)}(\mathcal{A}) = 0 \]
Theorem (Triangle inequality)

$$\alpha_{(F,H)}(A) \leq \alpha_{(F,G)}(A) + \alpha_{(G,H)}(A).$$
### Theorem (Triangle inequality)

\[ \alpha_{(F,H)}(A) \leq \alpha_{(F,G)}(A) + \alpha_{(G,H)}(A). \]

### Theorem (Reduction)

\[ \alpha_{(M \circ G^0, M \circ G^1)}(A) = \alpha_{(G^0, G^1)}(A \circ M). \]
### Theorem (Triangle inequality)

\[ \alpha_{(F,H)}(A) \leq \alpha_{(F,G)}(A) + \alpha_{(G,H)}(A). \]

### Theorem (Reduction)

\[ \alpha_{(M \circ G^0, M \circ G^1)}(A) = \alpha_{(G^0, G^1)}(A \circ M). \]

### Theorem (Relational equivalence \( \Rightarrow \) perf. indistinguishability)

Two games are perfectly indistinguishable if all their procedures are (i) equivalent in the pRHL, and (ii) maintain a stable invariant on the game state.
<table>
<thead>
<tr>
<th>Summary – SSProve:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A foundational built on standard</td>
</tr>
<tr>
<td>mathematical foundations</td>
</tr>
<tr>
<td>framework code, packages, laws, pRHL,</td>
</tr>
<tr>
<td>semantics, tactics</td>
</tr>
<tr>
<td>for modular programs composed from</td>
</tr>
<tr>
<td>packages</td>
</tr>
<tr>
<td>crypto proofs security properties of</td>
</tr>
<tr>
<td>probabilistic, stateful language</td>
</tr>
<tr>
<td>in Coq mature proof assistant with</td>
</tr>
<tr>
<td>clearly delimited TCB</td>
</tr>
<tr>
<td>Docs, code, info</td>
</tr>
<tr>
<td><a href="https://github.com/SSProve">github.com/SSProve</a></td>
</tr>
</tbody>
</table>