SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq

Carmine Abate[1] **Philipp G. Haselwarter[2]** Exequiel Rivas[3] Antoine Van Muylder[4] Théo Winterhalter[1] Cătălin Hrițcu[1] Kenji Maillard[5] Bas Spitters[2]

[1]MPI-SP [2]Aarhus University [3]Inria Paris [4]Vrije Universiteit Brussel [5]Inria Rennes

Logic and Semantics seminar,
Aarhus, 22.03.2021

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).

Why SSProve?

Motivation:

- In 2004: Shoup, Bellare and Rogaway: “crisis of rigor” in cryptography. Proposal: game-playing proofs.
- Monolithic game-based proofs can become intractable
- *state-separating proofs* from high-level structure of miTLS paper proofs (Brzuska, Delignat-Lavaud, Fournet, Kohbrok, Kohlweiss; 2018)

SSProve contributions:

- Formalise SSP in Coq
- Modular language, logic & semantics
- bridge between high-level SSP arguments and low-level program logic (Theorem 1)

Plan

- Informal State-Separating Proofs (SSP)
- Formal SSP in SSProve
- Probabilistic Relational Hoare Logic
- Future work

Informal SSP/Example 0: Pseudorandom functions

Given $\texttt{prf} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$

Informal SSP/Example 0: Pseudorandom functions

Given $\texttt{prf} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ consider

```
package: PRF^0
mem: k : option KEY

EVAL(x):
 if k = ⊥ then
   k <$ uniform {0,1}^n ;;
 return prf(k, x)
```

```
package: PRF^1
mem: T : map [KEY -> KEY]

EVAL(x):
 if T[x] = ⊥ then
   T[x] <$ uniform{0,1}^n ;;
 return T[x]
```

Informal SSP/Example 0: Pseudorandom functions

Given prf : $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ consider

```
package: PRF^0
mem: k : option KEY

EVAL(x):
 if k = ⊥ then
   k <$ uniform {0,1}^n ;;
 return prf(k, x)
```

```
package: PRF^1
mem: T : map [KEY -> KEY]

EVAL(x):
 if T[x] = ⊥ then
   T[x] <$ uniform{0,1}^n ;;
 return T[x]
```

Security notion: prf indistinguishable from random sampling. For any adversary $\mathcal{A}$,

$$\mathcal{A} \circ \mathrm{PRF}^0 \text{ ``}{=}\text{''} \mathcal{A} \circ \mathrm{PRF}^1$$

Informal SSP/Example 0: Pseudorandom functions

Given $\texttt{prf} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ consider

```
package:  PRF^0
mem:  k :  option KEY

EVAL(x):
 if k = ⊥ then
   k <$ uniform {0,1}^n ;;
 return prf(k, x)
```

```
package:  PRF^1
mem:  T : map [KEY -> KEY]

EVAL(x):
 if T[x] = ⊥ then
   T[x] <$ uniform{0,1}^n ;;
 return T[x]
```

Security notion: prf indistinguishable from random sampling. For any adversary $\mathcal{A}$,

$$\mathcal{A} \circ \mathrm{PRF}^0 \text{ ``$=$'' } \mathcal{A} \circ \mathrm{PRF}^1$$

What's an adversary? What does $\mathcal{A} \circ P$ do? What does indistinguishability mean? What is k <$ uniform $\{0,1\}^n$?

Informal SSP/Packages, Games, Adversaries

- Package: collection of *exported* procedure implementations
  - stateful, probabilistic
  - can call *imported* procedures

Informal SSP/Packages, Games, Adversaries

- Package: collection of *exported* procedure implementations
  - stateful, probabilistic
  - can call *imported* procedures
- Package operations:
  - seq. composition $P_1 \circ P_2$ if $\texttt{import}(P_1) \subseteq \texttt{export}(P_2)$
  - par. comp. $P_1 \parallel P_2$ if $\texttt{export}(P_1) \cap \texttt{export}(P_2) = \emptyset$

Informal SSP/Packages, Games, Adversaries

- Package: collection of *exported* procedure implementations
  - stateful, probabilistic
  - can call *imported* procedures
- Package operations:
  - seq. composition $P_1 \circ P_2$ if $\texttt{import}(P_1) \subseteq \texttt{export}(P_2)$
  - par. comp. $P_1 \parallel P_2$ if $\texttt{export}(P_1) \cap \texttt{export}(P_2) = \emptyset$
- Game: package with no imports
- Game pair: two games with the same exports

Informal SSP/Packages, Games, Adversaries

- Package: collection of *exported* procedure implementations
  - stateful, probabilistic
  - can call *imported* procedures
- Package operations:
  - seq. composition $P_1 \circ P_2$ if $\texttt{import}(P_1) \subseteq \texttt{export}(P_2)$
  - par. comp. $P_1 \parallel P_2$ if $\texttt{export}(P_1) \cap \texttt{export}(P_2) = \emptyset$
- Game: package with no imports
- Game pair: two games with the same exports
- Adversary $\mathcal{A}$ for game G: package compatible with G exporting one procedure $\texttt{run : unit} \rightarrow \texttt{bool}$
- Advantage of $\mathcal{A}$ against (G_0, G_1): $\alpha_{(G_0,G_1)}(\mathcal{A}) =$

  $$|\Pr[\text{true} \leftarrow (\mathcal{A} \circ G_0).run()] - \Pr[\text{true} \leftarrow (\mathcal{A} \circ G_1).run()]|$$

- Perfect indistinguishability $G_0 \overset{0}{\approx} G_1$: $\forall \mathcal{A}\,.\ \alpha_{(G_0,G_1)}(\mathcal{A}) = 0$

Informal SSP/IND-CPA security for PRF-based encryption

Symmetric encryption from $\texttt{prf} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$

```
enc(k, m):
  r <$ uniform {0,1}^n ;;
  pad ← prf(k, r) ;;
  c ← m xor pad ;;
  return (r, c)

kgen():
  k <$ uniform {0,1}^n ;;
  return k

dec(k, (r, c)):
  pad ← prf(k, r) ;;
  m ← c xor pad ;;
  return m
```

Goal: prove that

```
package: IND-CPA^0
mem: k : option KEY

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

$\overset{\varepsilon}{\approx}$

```
package: IND-CPA^1
mem: k : option KEY

ENC(m):
 if k = ⊥ then
   k <$ uniform {0,1}^n ;;
 m' <$ uniform {0,1}^n ;;
 (r,c) ← enc(k, m') ;;
 return (r,c)
```

Informal SSP/PRF IND-CPA Game-hopping (1)

```
package: IND-CPA⁰
mem: k : option KEY

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

=

```
package: IND-CPA⁰
mem: k : option KEY

ENC(m):
 if k = ⊥ then
   k <$ uniform {0,1}^n
 r <$ uniform {0,1}^n
 pad ← prf(k, r)
 c ← m xor pad
 return (r, c)
```

(def. of enc)

$\overset{0}{\approx}$

```
package: MOD-CPA⁰
mem:

ENC(m):
 r <$ uniform {0,1}^n
 pad ← EVAL(r)
 c ← m xor pad
 return (r, c)
```

$\circ$

```
package: PRF⁰
mem: k : option KEY

EVAL(x):
 if k = ⊥ then
   k <$ uniform {0,1}^n
 return prf(k, x)
```

(Theorem 1 + swap rule)

Informal SSP/PRF IND-CPA Game-hopping (2)

$\texttt{IND-CPA}^0$

$\overset{0}{\approx}$

```
package: MOD-CPA^0
mem:

ENC(m):
 r <$ uniform {0,1}^n
 pad ← EVAL(r)
 c ← m xor pad
 return (r, c)
```

$\circ$

```
package: PRF^0
mem: k : option KEY

EVAL(x):
 if k = ⊥ then
   k <$ uniform {0,1}^n
 return prf(k, x)
```

$\overset{\varepsilon^0_{\mathtt{PRF}}}{\approx}$

```
package: MOD-CPA^0
mem:

ENC(m):
 r <$ uniform {0,1}^n
 pad ← EVAL(r)
 c ← m xor pad
 return (r, c)
```

$\circ$

```
package: PRF^1
mem: T : map [KEY -> KEY]

EVAL(x):
 if T[x] = ⊥ then
   T[x] <$ uniform{0,1}^n
 return T[x]
```

(assumption + Reduction Lemma)

Informal SSP/PRF IND-CPA Game-hopping (3)

$$\texttt{IND-CPA}^0 \overset{0}{\approx} \texttt{MOD-CPA}^0 \circ \texttt{PRF}^0$$

$\overset{\varepsilon^0_{\mathrm{PRF}}}{\approx}$

```
package: MOD-CPA^0
mem:

ENC(m):
 r <$ uniform {0,1}^n
 pad ← EVAL(r)
 c ← m xor pad
 return (r, c)
```

$\circ$

```
package: PRF^1
mem: T : map [KEY -> KEY]

EVAL(x):
 if T[x] = ⊥ then
   T[x] <$ uniform{0,1}^n
 return T[x]
```

$\overset{\varepsilon_{stat.\ gap}}{\approx}$

```
package: MOD-CPA^1
mem:

ENC(m):
 m' <$ uniform {0,1}^n
 r <$ uniform {0,1}^n
 pad ← EVAL(r)
 c ← m' xor pad
 return (r, c)
```

$\circ$

```
package: PRF^1
mem: T : map [KEY -> KEY]

EVAL(x):
 if T[x] = ⊥ then
   T[x] <$ uniform{0,1}^n
 return T[x]
```

(bday bound)

Informal SSP/PRF IND-CPA Game-hopping (4)

$$\texttt{IND-CPA}^0 \overset{0}{\approx} \texttt{MOD-CPA}^0 \circ \texttt{PRF}^0 \overset{\varepsilon^0_{\mathrm{PRF}}}{\approx} \texttt{MOD-CPA}^0 \circ \texttt{PRF}^1$$

$\overset{\varepsilon_{stat.\ gap}}{\approx}$

```
package: MOD-CPA¹
mem:

ENC(m):
 m' <$ uniform {0,1}^n
 r <$ uniform {0,1}^n
 pad ← EVAL(r)
 c ← m' xor pad
 return (r, c)
```

$\circ$

```
package: PRF¹
mem: T : map [KEY -> KEY]

EVAL(x):
 if T[x] = ⊥ then
   T[x] <$ uniform{0,1}^n
 return T[x]
```

$\overset{\varepsilon^1_{\mathrm{PRF}}}{\approx}$

```
package: MOD-CPA¹
mem:

ENC(m):
 m' <$ uniform {0,1}^n
 r <$ uniform {0,1}^n
 pad ← EVAL(r)
 c ← m' xor pad
 return (r, c)
```

$\circ$

```
package: PRF⁰
mem: k : option KEY

EVAL(x):
 if k = ⊥ then
   k <$ uniform {0,1}^n
 return prf(k, x)
```

(assumption + Reduction Lemma)

Informal SSP/PRF IND-CPA Game-hopping (5)

$$\texttt{IND-CPA}^0 \overset{0}{\approx} \texttt{MOD-CPA}^0 \circ \texttt{PRF}^0 \overset{\varepsilon^0_{\texttt{PRF}}}{\approx} \texttt{MOD-CPA}^0 \circ \texttt{PRF}^1 \overset{\varepsilon_{stat.}}{\approx} \texttt{MOD-CPA}^1 \circ \texttt{PRF}^1$$

$\overset{\varepsilon^1_{\texttt{PRF}}}{\approx}$

```
package: MOD-CPA^1
mem:

ENC(m):
 m' <$ uniform {0,1}^n
 r <$ uniform {0,1}^n
 pad ← EVAL(r)
 c ← m' xor pad
 return (r, c)
```

$\circ$

```
package: PRF^0
mem: k : option KEY

EVAL(x):
 if k = ⊥ then
   k <$ uniform {0,1}^n
 return prf(k, x)
```

$\overset{0}{\approx}$

```
package: IND-CPA^1
mem: k : option KEY

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

(Theorem 1 + swap rule)

Informal SSP/PRF IND-CPA Game-hopping (6)

We have shown:

$$\begin{aligned}
\mathtt{IND\text{-}CPA}^0 &\overset{0}{\approx} \mathtt{MOD\text{-}CPA}^0 \circ \mathtt{PRF}^0 \\
&\overset{\varepsilon^0_{\mathtt{PRF}}}{\approx} \mathtt{MOD\text{-}CPA}^0 \circ \mathtt{PRF}^1 \\
&\overset{\varepsilon_{\mathit{stat.}}}{\approx} \mathtt{MOD\text{-}CPA}^1 \circ \mathtt{PRF}^1 \\
&\overset{\varepsilon^1_{\mathtt{PRF}}}{\approx} \mathtt{MOD\text{-}CPA}^1 \circ \mathtt{PRF}^0 \\
&\overset{0}{\approx} \mathtt{IND\text{-}CPA}^1
\end{aligned}$$

Thus by triangle inequality,

$$\alpha_{(\mathtt{IND\text{-}CPA}^0,\mathtt{IND\text{-}CPA}^1)}(\mathcal{A}) \leq \varepsilon^0_{\mathtt{PRF}} + \varepsilon_{\mathit{stat.}} + \varepsilon^1_{\mathtt{PRF}}$$

SSProve/Core language

```
Inductive code (A : choiceType) : Type :=
| ret (x : A)
| call (p : opsig) (x : src p) (κ : tgt p → code A)
| get (ℓ : Location) (κ : type ℓ → code A)
| put (ℓ : Location) (v : type ℓ) (κ : code A)
| sample (op : Op) (κ : Arit op → code A).

Def opsig := ident × (chUniverse × chUniverse)
```

SSProve/Core language

```
Inductive code (A : choiceType) : Type :=
| ret (x : A)
| call (p : opsig) (x : src p) (κ : tgt p → code A)
| get (ℓ : Location) (κ : type ℓ → code A)
| put (ℓ : Location) (v : type ℓ) (κ : code A)
| sample (op : Op) (κ : Arit op → code A).
```

```
Def opsig := ident × (chUniverse × chUniverse)
```

Example:

$$\texttt{get}\ \ell\ (\lambda\ x_\ell\ .\ \texttt{put}\ \ell\ (x_\ell + 1)\ (\texttt{ret}\ x_\ell)) \quad (1)$$

$$\texttt{sample}\ (\texttt{uniform}\ \{0,1\}^n)\ (\lambda\ y\ .\ \texttt{call prf}\ (y,\ 101010)\ (\lambda\ z\ .\ \texttt{ret}\ z)) \quad (2)$$

(1) $\sim \ell{+}{+}$ (2) $\sim$ y <\$ $\{0,1\}^n$; ret prf(y, 101010)

SSProve/User language

- **Fix** bind (c : code A) (κ : A $\rightarrow$ code B) : code B
- **Fix** for_loop (c : nat $\rightarrow$ code unit) (n: nat) : code unit
- **Fix** do_while (N : nat) (c : code bool) : code bool
- **Def** assert (b : bool) : code unit :=
  if b **then** ret tt **else** sample null (λ F . ret F)
- x $\leftarrow$ sample U A: uniform distribution on finite type A

SSProve/Packages

```
Inductive code (A : choiceType) : Type :=
| ret (x : A)
| call (p : opsig) (x : src p) (κ : tgt p → code A)
...

Def raw_package := {fmap ident →  Σ(A,B:Type) A → code B}
Def Interface := {fset opsig}
```

SSProve/Packages

```
Inductive code (A : choiceType) : Type :=
| ret (x : A)
| call (p : opsig) (x : src p) (κ : tgt p → code A)
...

Def raw_package := {fmap ident → Σ(A,B:Type) A → code B}
Def Interface := {fset opsig}
```

Sequential & parallel composition: $P_1 \circ P_2$, $P_1 \parallel P_2$.

SSProve/Packages

```
Inductive code (A : choiceType) : Type :=
| ret (x : A)
| call (p : opsig) (x : src p) (κ : tgt p → code A)
...

Def raw_package := {fmap ident → Σ(A,B:Type) A → code B}
Def Interface := {fset opsig}
```

Sequential & parallel composition: $P_1 \circ P_2$, $P_1 \parallel P_2$.

Laws:

$$
\begin{aligned}
P_1 \circ (P_2 \circ P_3) &= (P_1 \circ P_2) \circ P_3 \\
P_1 \parallel P_2 &= P_2 \parallel P_1 \\
P_1 \parallel (P_2 \parallel P_3) &= (P_1 \parallel P_2) \parallel P_3 \\
(P_1 \circ P_3) \parallel (P_2 \circ P_4) &= (P_1 \parallel P_2) \circ (P_3 \parallel P_4)
\end{aligned}
$$

SSProve/Cryptographic notions

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

SSProve/Cryptographic notions

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

Reduction lemma: $\alpha_{(M \circ G^0, M \circ G^1)}(\mathcal{A}) = \alpha_{(G^0, G^1)}(\mathcal{A} \circ M)$.

Theorem

Let (G_0, G_1) be a game pair with respect to export interface $\mathcal{E} = \texttt{export}(G_i)$. Moreover, assume that ψ is a stable invariant that relates the memories of G_0 and G_1, and that it holds on the initial memories.
If for each provided procedure $f : A \rightarrow B \in \mathcal{E}$ and for all $a \in A$,

$$\vDash \{\psi\}\ G_0.f(a) \sim G_1.f(a)\ \{(b_0, b_1).\ b_0 = b_1 \wedge \psi\},$$

then we can conclude that $G_0 \overset{0}{\approx} G_1$.

SSProve/probabilistic Relational Hoare Logic

$\texttt{let } \mathtt{c}_{xy} := \quad x <\$ \ (\texttt{uni } \{0,1\}^n) \ ; \ y <\$ \ (\texttt{uni } \{0,1\}^n); \ \texttt{ret } (x, y)$

$\texttt{let } \mathtt{c}_{yx} := \quad y <\$ \ (\texttt{uni } \{0,1\}^n) \ ; \ x <\$ \ (\texttt{uni } \{0,1\}^n); \ \texttt{ret } (x, y)$

Prove $\mathtt{c}_{xy}$ "=" $\mathtt{c}_{yx}$? (†)

SSProve/probabilistic Relational Hoare Logic

$\texttt{let}\ \mathtt{c}_{xy} := \ x \mathrel{<\$} (\texttt{uni}\ \{0,1\}^n)\ ;\ y \mathrel{<\$} (\texttt{uni}\ \{0,1\}^n);\ \texttt{ret}\ (x,y)$

$\texttt{let}\ \mathtt{c}_{yx} := \ y \mathrel{<\$} (\texttt{uni}\ \{0,1\}^n)\ ;\ x \mathrel{<\$} (\texttt{uni}\ \{0,1\}^n);\ \texttt{ret}\ (x,y)$

Prove $\mathtt{c}_{xy}$ "=" $\mathtt{c}_{yx}$? (†)

$$\vDash \{pre\}\ c_0 \sim c_1\ \{(r_0, r_1).post\}$$

where $\mathtt{c}_i$: $\texttt{code}\ \mathtt{A}_i$ (no procedure calls) and m_0, m_1 are bound in *pre*, *post*

pre / *post* are predicates on memory / memory and results

Valid if $\exists$ coupling d with $\pi_i\, d = [\![c_i]\!]$ s.t. *pre* and *post* hold for d with probability > 0.

SSProve/probabilistic Relational Hoare Logic

$\texttt{let}\ \mathtt{c}_{xy} := \quad x \,\texttt{<\$}\, (\texttt{uni}\ \{0,1\}^n)\ ;\ y \,\texttt{<\$}\, (\texttt{uni}\ \{0,1\}^n);\ \texttt{ret}\ (x, y)$

$\texttt{let}\ \mathtt{c}_{yx} := \quad y \,\texttt{<\$}\, (\texttt{uni}\ \{0,1\}^n)\ ;\ x \,\texttt{<\$}\, (\texttt{uni}\ \{0,1\}^n);\ \texttt{ret}\ (x, y)$

Prove $\mathtt{c}_{xy}$ "=" $\mathtt{c}_{yx}$? (†)

$$\vDash \{pre\}\ c_0 \sim c_1\ \{(r_0, r_1).post\}$$

where $\mathtt{c}_i :\ \texttt{code}\ \mathtt{A}_i$ (no procedure calls) and m_0, m_1 are bound in *pre*, *post*

pre / *post* are predicates on memory / memory and results

Valid if $\exists$ coupling d with $\pi_i\, d = [\![c_i]\!]$ s.t. *pre* and *post* hold for d with probability > 0.

ad (†): $\vDash \{True\}\ c_{xy} \sim c_{yx}\ \{(r_{xy}, r_{yx}).\ r_{xy} = r_{yx}\}$

SSProve/Some rules

$$\frac{c : \texttt{code}\ \ L\ A}{\vDash \{m_0 = m_1\}\ c \sim c\ \{(r_0, r_1).\ m_0 = m_1 \wedge r_0 = r_1\}}\ \texttt{reflexivity}$$

$$\frac{\begin{array}{c} c_0 : \texttt{code}\ \ L_0\ A_0 \quad c_1 : \texttt{code}\ \ L_1\ A_1 \\ f_0 : A_0 \rightarrow\ \texttt{code}\ \ L_0\ B_0 \quad f_1 : A_1 \rightarrow\ \texttt{code}\ \ L_1\ B_1 \\ \vDash \{pre\}\ c_0 \sim c_1\ \{\mu\} \\ \forall a_0 a_1.\ \vDash \{(h_0, h_1).\ \mu(a_0, h_0)(a_1, h_1)\}\ (f_0\ a_0) \sim (f_1\ a_1)\ \{post\} \end{array}}{\vDash \{pre\}\ a_0 \leftarrow c_0;;\ f_0\ a_0\ \sim\ a_1 \leftarrow c_1;;\ f_1\ a_1\ \{post\}}\ \texttt{seq}$$

$$\frac{\begin{array}{c} c_0 : \texttt{code}\ \ L\ A_0 \quad c_1 : \texttt{code}\ \ L\ A_1 \\ \vDash \{I\}\ c_0 \sim c_1\ \{(a_0, a_1).\ I \wedge post(a_0, a_1)\} \\ \vDash \{I\}\ c_1 \sim c_0\ \{(a_1, a_0).\ I \wedge post(a_0, a_1)\} \end{array}}{\vDash \{I\}\ c_0;;\ c_1 \sim c_1;;\ c_0\ \{(a_0, a_1).\ I \wedge post(a_0, a_1)\}}\ \texttt{swap}$$

$$\frac{\begin{array}{c} c_0\ c_0' : \texttt{code}\ \ L\ A_0 \quad c_1 : \texttt{code}\ \ J\ A_1 \\ \vDash \{pre\}\ c_0 \sim c_1\ \{post\} \qquad \forall h.\ \theta(c_0\ \ h) = \theta(c_0'\ \ h) \end{array}}{\vDash \{pre\}\ c_0' \sim c_1\ \{post\}}\ \texttt{eqDistrL}$$

$$\frac{\begin{array}{c} c_0 : \texttt{code}\ \ L\ A_0 \quad c_1 : \texttt{code}\ \ L\ A_1 \\ \vDash \{pre\}\ c_0 \sim c_1\ \{post\} \end{array}}{\vDash \{pre^{-1}\}\ c_1 \sim c_0\ \{post^{-1}\}}\ \texttt{symmetry}$$

SSProve/Some more rules

$$\frac{c_0, c_1 : \mathbb{N} \to \texttt{code}\ \ L\ \texttt{unit} \quad N : \mathbb{N} \qquad \forall n.\ \vDash \{I\ n\}\ c_0 \sim c_1\ \{I\ (n+1)\}}{\vDash \{I\ 0\}\ \texttt{for\_loop}\ N\, c_0 \sim \texttt{for\_loop}\ N\, c_1\ \{I\ (N+1)\}}\ \texttt{for-loop}$$

$$\frac{c_0, c_1 : \texttt{code}\ \ L\ \texttt{bool} \quad N : \mathbb{N} \qquad \vDash \{I(\texttt{true}, \texttt{true})\}\ c_0 \sim c_1\ \{(b_0, b_1).\ b_0 = b_1 \land I(b_0, b_1)\}}{\vDash \{I(\texttt{true}, \texttt{true})\}\ \texttt{do\_while}\ N\, c_0 \sim \texttt{do\_while}\ N\, c_1\ \{(b_0, b_1).\ b_0 = b_1 = \texttt{false} \lor I(\texttt{false}, \texttt{false})\}}\ \texttt{do-while}$$

$$\frac{|A|, |B| < \omega \quad f : A \to B\ \textit{bijective}}{\vDash \{pre\}\ a <\$\ \mathcal{U}(A) \sim b <\$\ \mathcal{U}(B)\ \{(a, b).\ f(a) = b \land pre\}}\ \texttt{uniform}$$

$$\frac{b_0, b_1 : \texttt{bool}}{\vDash \{b_0 = b_1\}\ \texttt{assert}\ \ b_0 \sim \texttt{assert}\ \ b_1\ \{b_0 = \texttt{true} \land b_1 = \texttt{true}\}}\ \texttt{asrt}$$

$$\frac{b : \texttt{bool}}{\vDash \{b = \texttt{true}\}\ \texttt{assert}\ \ b \sim \texttt{return}\ \ ()\ \{b = \texttt{true}\}}\ \texttt{asrtL}$$

SSProve/The swap rule used in IND-CPA

$$\frac{\begin{array}{c}\vDash \{m_0 = m_1\}\ \texttt{k <\$ uniform}\ \{0,1\}^{\texttt{n}} \sim \texttt{r <\$ uniform}\ \{0,1\}^{\texttt{n}}\ \{m_0 = m_1 \land c_0 = c_1\} \\ \vDash \{m_0 = m_1\}\ \texttt{r <\$ uniform}\ \{0,1\}^{\texttt{n}} \sim \texttt{k <\$ uniform}\ \{0,1\}^{\texttt{n}}\ \{m_0 = m_1 \land c_0 = c_1\}\end{array}}{\begin{array}{c}\vDash \{m_0 = m_1\} \\ \texttt{k <\$ uniform}\ \{0,1\}^{\texttt{n}}\ \texttt{;;}\ \texttt{r <\$ uniform}\ \{0,1\}^{\texttt{n}} \sim \\ \texttt{r <\$ uniform}\ \{0,1\}^{\texttt{n}}\ \texttt{;;}\ \texttt{k <\$ uniform}\ \{0,1\}^{\texttt{n}} \\ \{m_0 = m_1 \land c_0 = c_1\}\end{array}}$$

SSProve/Semantics

Based on Maillard, Hritcu, Rivas, V. Muylder's *The next 700 relational program logics*:

- Associate a *relational specification monad* (RSM) to the computational monad
- Define *effect observation* as a relative monad morphism
- First for probabilities, then state-transform

Future work

- Security verification of real / low level code
  - code generation for testing
  - integration with VST / FiatCrypto
- Extend the language
  - add more effects: non-termination, I/O
- Extend the logic
  - prove more relational rules
  - unary probabilistic Hoare logic
  - specifications for packages
- More scalable semantics: decouple specification from implementation (CoqEAL?)
- Extend cryptographic scope
  - larger case studies (orig. SSP paper, recent SSP)
  - add further reusable security definitions

Future work

Thank you

- Security verification of real / low level code
  - code generation for testing
  - integration with VST / FiatCrypto
- Extend the language
  - add more effects: non-termination, I/O
- Extend the logic
  - prove more relational rules
  - unary probabilistic Hoare logic
  - specifications for packages
- More scalable semantics: decouple specification from implementation (CoqEAL?)
- Extend cryptographic scope
  - larger case studies (orig. SSP paper, recent SSP)
  - add further reusable security definitions

https://www.github.com/SSProve/ssprove