Mechanized Proofs of Adversarial Complexity and Application to Universal Composability

SCOT seminar

Manuel Barbosa  
University of Porto (FCUP) & INESC TEC

Gilles Barthe  
MPI-SP & IMDEA Software Institute

Benjamin Grégoire  
Inria Sofia

Adrien Koutsos  
Inria Paris

Pierre-Yves Strub  
PQShield

December 15, 2023
Cryptographic systems provide security to many applications.

- Critical + pervasive: high-level of confidence needed.
- Formal methods:
  - precise and rigorous formulation of security properties.
  - security proofs.
- Security proofs are complicated and error-prone.
  ⇒ proof mechanization: highest level of confidence.
Formalizing the security of an asymmetric encryption.

Encryption: $\text{enc}(m, pk)$
Decryption: $\text{dec}(m, sk)$

Asymmetric encryption scheme is secure if:

No adversary can distinguish between the encryptions of two plaintexts even if it chooses them.

Example: $\text{enc}(0, sk) \sim \text{enc}(1, sk)$
No cat can distinguish between the encryptions of two plaintexts even if it chooses them.

\[
\text{adv}_S(\mathcal{A}) = \left| \Pr \left[ b' \leftarrow \mathcal{A}(O) : b' = b \right] - \frac{1}{2} \right|
\]
Cryptographic Games

- **Security properties for \( S \):**
  - *game* between an adversary \( \mathcal{A} \) and a challenger.

```plaintext
Challenger for \( S \)
\( \mathcal{O}_1, \ldots, \mathcal{O}_n \) (oracles)
```

- **The advantage** \( \text{adv}_S(\mathcal{A}) \) is \( \Pr[\mathcal{A}(\mathcal{O}_1, \ldots, \mathcal{O}_n) \text{ wins}] \).

\( \Rightarrow \) \text{Advantage of an unbounded adversary is often 1.}

\( S \) secure \( \iff \) \( \text{adv}_S(\mathcal{A}) \) is small for any efficient \( \mathcal{A} \).
Cryptographic Games

- **Security properties for $S$:**
  
  A game between an adversary $\mathcal{A}$ and a challenger.

  - The advantage $\text{adv}_S(\mathcal{A})$ is $\Pr[\mathcal{A}(O_1, \ldots, O_n) \text{ wins}]$.
  - Advantage of an unbounded adversary is often 1.
    - $\Rightarrow$ $\mathcal{A}$’s resources must be limited.
  - $S$ secure $\iff$ $\text{adv}_S(\mathcal{A})$ is small for any efficient $\mathcal{A}$.
Crypto. systems are **combined** to provide more **involved properties**.

- **Diffie-Hellman**
  - hardness assumption

- **Signature**
  - authentication

- **Key-Exchange**
  - secret shared key

- **Encryption**
  - secrecy

- **Secure Channel**
  - secrecy + authentication

\[ S \text{ denotes cryptographic reduction.} \]

If an efficient adversary can break \( S \) then there exists an efficient adversary breaking \( H \).
Crypto. systems are **combined** to provide more **involved properties**.

- **Diffie-Hellman**
  - hardness assumption

- **Signature**
  - authentication

- **Key-Exchange**
  - secret shared key

- **Encryption**
  - secrecy

- **Secure Channel**
  - secrecy + authentication

- $S \Rightarrow H$ denotes **cryptographic reduction**.

  *If an efficient adversary 🦈 can break $S$ then there exists an efficient adversary 🦈 breaking $H$.***
**Cryptographic Reduction**  \( S \rightarrow \mathcal{H} \)

\( S \) reduces to a hardness hypothesis \( \mathcal{H} \) if:

\[
\forall A. \exists B. \ \text{adv}_S(A) \leq \text{adv}_\mathcal{H}(B) + \epsilon \land \text{cost}(B) \leq \text{cost}(A) + \delta
\]

where \( \epsilon \) and \( \delta \) are small.
A proof assistant to verify cryptographic proofs. It relies on:
- general purpose higher-order ambient logic.
- probabilistic relational Hoare logic (pRHL).
- powerful module system.

Many advanced existing case studies: AWS KMS, SHA3, ...
In \texttt{EasyCrypt} proof, the adversary against $\mathcal{H}$ is explicitly constructed:

$$\forall \mathcal{A}. \text{adv}_S(\mathcal{A}) \leq \text{adv}_\mathcal{H}(C[\mathcal{A}]) + \epsilon \quad (\dagger)$$

But \texttt{EasyCrypt} lacked support for \textit{complexity upper-bounds}. 
In **EasyCrypt** proof, the adversary against \( \mathcal{H} \) is explicitly constructed:

\[
\forall A. \text{adv}_S(A) \leq \text{adv}_H(C[A]) + \epsilon \tag{†}
\]

But **EasyCrypt** lacked support for **complexity upper-bounds**.

**Getting a \( \forall \exists \) statement**

(†) implies that:

\[
\forall A. \exists B. \text{adv}_S(A) \leq \text{adv}_H(B) + \epsilon
\]

but this statement is **useless**, since \( B \) is not resource-limited: its advantage is often 1.
Mechanizing Cryptographic Reduction

Hence adversaries **constructed** in reductions are kept **explicit**:

\[ \forall \mathcal{A}. \text{adv}_S(\mathcal{A}) \leq \text{adv}_H(C[\mathcal{A}]) + \epsilon \]

**Limitations**

- **Not fully verified**: $C[\mathcal{A}]$’s complexity is checked manually.
- **Less composable**, as composition is done manually (inlining).

If

\[ \forall \mathcal{A}. \text{adv}_S(\mathcal{A}) \leq \text{adv}_{H_1}(C[\mathcal{A}]) + \epsilon_1 \]

and

\[ \forall \mathcal{D}. \text{adv}_{H_1}(\mathcal{D}) \leq \text{adv}_{H_2}(F[\mathcal{D}]) + \epsilon_2 \]

then

\[ \forall \mathcal{A}. \text{adv}_S(\mathcal{A}) \leq \text{adv}_{H_2}(F[C[\mathcal{A}]])) + \epsilon_1 + \epsilon_2 \]
Our Contributions

- A Hoare logic to prove **worst-case complexity** upper-bounds of probabilistic programs.
  \[\Rightarrow\text{fully mechanized cryptographic reductions.}\]

- Implemented in **EasyCrypt**, embedded in its ambient higher-order logic.
  \[\Rightarrow\text{meaningful } \forall \exists \text{ statements: better composability.}\]

- Application: **UC** formalization in **EasyCrypt**.

- First formalization of **EasyCrypt** module system.
Hoare Logic for Complexity
The Bellare-Rogaway scheme builds a public-key encryption from:

- a trapdoor permutation
- and a random oracle (modeling a hash function).
The Bellare-Rogaway scheme builds a public-key encryption from:

- a trapdoor permutation
- and a random oracle (modeling a hash function).
**Example: Bellare-Rogaway, 93**

---

**Concrete**

```plaintext
proc invert(pk:pkey,y:rand): rand = {
    log ← [];
    Adv.choose(pk);
    h ← dptxt;
    Adv.guess(y || h);
    while (log ≠ []) {
        r ← head log;
        if (f pk r = y) return r;
        log ← tail log;
    }
}
```

**Abstract**

```plaintext
proc choose(p:pkey) : unit
proc guess(c:ctxt) : unit
```

**Adv**

Property:

\[ |log| \leq k_c + k_g \]

Complexity:

\[
\text{Concrete: (5+tf) \cdot (k_c+k_g) + 4,}\]

\[
\text{Abstract: 1,}\]

\[
\text{RO: } k_c+k_g\]

Memory:

Adv must not access the log in Log
Example: Bellare-Rogaway, 93

Concrete

\[
\text{proc} \ \text{invert}(pk:\text{pkey}, y:\text{rand}): \text{rand} = \{
\text{log} \leftarrow []; \\
\text{Adv}.\text{choose}(pk); \\
h \leftarrow dptxt; \\
\text{Adv}.\text{guess}(y \parallel h); \\
\text{while} (\text{log} \neq []) \{ \\
\quad r \leftarrow \text{head log}; \\
\quad \text{if} (f pk r = y) \text{return} r; \\
\quad \text{log} \leftarrow \text{tail log}; \\
\}\}
\]

Abstract

\[
\text{proc} \ \text{choose}(p:\text{pkey}) : \text{unit} \\
\text{proc} \ \text{guess}(c:\text{ctxt}) : \text{unit}
\]

\[
\text{proc} \ \text{o}(r:\text{rand}): \text{ptxt}
\]

\[
\text{Property: } |\text{log}| \leq k_c + k_g
\]

\[
\text{Complexity: } [\text{conc}: (5 + t + f) \cdot (k_c + k_g) + 4, \\
\text{Adv}.\text{choose}: 1, \\
\text{Adv}.\text{guess}: 1, \\
\text{RO}.\text{o}: k_c + k_g]
\]

Memory: \text{Adv} must not access the log in \text{Log}
proc invert(pk:pkey,y:rand): rand = {
    log ← []; 
    Adv(Log(RO)).choose(pk); 
    h ← dptxt; 
    Adv(Log(RO)).guess(y || h); 
    while (log ≠ []) {
        r ← head log; 
        if (f pk r = y) return r; 
        log ← tail log; 
    } 
}

Inverter

proc choose(p:pkey) : unit
proc guess(c:ctxt) : unit

Adv

proc o(r:rand): ptxt = {
    log ← r :: log; 
    return RO.o(r); 
}

Log

proc o(r:rand): ptxt

RO
proc invert(pk:pkey,y:rand): rand = {
  log ← []; 
  Adv(Log(RO)).choose(pk);
  h ← dptxt;
  Adv(Log(RO)).guess(y || h);
  while (log ≠ []) {
    r ← head log;
    if (f pk r = y) return r;
    log ← tail log;
  }
}
Inverter

proc choose(p:pkey) : unit ≤ k_c
proc guess(c:ctxt) : unit ≤ k_g
Adv

proc o(r:rand): ptxt = {
  log ← r :: log;
  return RO.o(r);
}
Log

proc o(r:rand): ptxt
RO

Property: |log| ≤ k_c + k_g
Complexity: [conc : (5 + t_f) · (k_c + k_g) + 4,
Adv.choose : 1,
Adv.guess : 1,
RO.o : k_c + k_g]
Example: Bellare-Rogaway, 93

proc invert(pk:pkey, y:rand): rand = {
    log ← [];
    Adv(Log(RO)).choose(pk);
    h ← \$ dptxt;
    Adv(Log(RO)).guess(y || h);
    while (log \neq []) {
        r ← head log;
        if (f pk r = y) return r;
        log ← tail log;
    }
}

proc choose(p:pkey): unit = \leq k_c
proc guess(c:ctxt): unit = \leq k_g

Adv

proc o(r:rand): ptxt = {
    log ← r :: log;
    return RO.o(r);
}

Log

RO

Property: \| log \| \leq k_c + k_g
Complexity: [conc : (5 + t_f) \cdot (k_c + k_g) + 4,
Adv.choose : 1,
Adv.guess : 1,
RO.o : k_c + k_g]

Memory: Adv must not access the log in Log

Concrete

Abstract
Key Ingredients

- Support programs mixing **concrete** and **abstract** code.
  Example: $\text{Adv}(\text{Log}(\text{RO}))$

- **Complexity** upper-bound requires some program **invariants**.
  Example: $|\log| \leq k_c + k_g$
Key Ingredients

- Support programs mixing **concrete** and **abstract** code.
  Example: \(\text{Adv}(\log(\text{RO}))\)

- **Complexity** upper-bound requires some program **invariants**.
  Example: \(|\log| \leq \kappa_c + \kappa_g\)

**Abstract** procedures must be **restricted**:
- **Complexity**: restrict intrinsic cost/number of calls to oracles.
  Example: \(\text{choose}\) can call \(\circ \leq \kappa_c\) times.

- **Memory footprint**: some memory areas are off-limit.
  Example: \(\text{Adv}\) cannot access the log in \(\log\)'s memory
Abstract code modeled as any program implementing some module signature (à la ML)

```
module type RO = {
  proc o (r:rand) : ptxt
}.

module type Adv (H: RO) = {
  proc choose(p:pkey) : unit
  proc guess(c:ctxt) : unit
}.
```
Abstract code modeled as any program implementing some module signature (à la ML), with some restrictions:

- Module memory footprint can be restricted.

```ml
module type RO = {
    proc o (r:rand) : ptxt
}.

module type Adv (H: RO) {+all mem, -Log, -RO, -Inverter} = {
    proc choose(p:pkey) : unit
    proc guess(c:ctxt) : unit
}.
```
Abstract code modeled as any program implementing some module signature (à la ML), with some restrictions:

- Module memory footprint can be restricted.
- Procedure complexity can be upper-bounded.

```plaintext
module type RO = {
    proc o (r:rand) : ptxt [intr : t_o]
}.

module type Adv (H: RO) {+all mem, -Log, -RO, -Inverter} = {
    proc choose(p:pkey) : unit [intr : t_c, H.o : k_c]
    proc guess(c:ctxt) : unit [intr : t_g, H.o : k_g]
}.
```
Assuming $\phi$, evaluating $s$ guarantees $\psi$, and takes time at most $c$. 

$E \vdash \{ \phi \} s \{ \psi \mid c \}$
Assuming $\phi$, evaluating $s$ guarantees $\psi$, and takes time at most $c$.

Example: $\mathcal{E} \vdash \{T\} \text{ Inverter(Adv,RO).invert} \{ |\log| \leq k_c + k_g | c\}$
Cost Vectors

**Concrete cost**

\[ c ::= [\text{conc} : k, O_1.f_1 : k_1, \ldots, O_l.f_l : k_l] \]

**Abstract procedures**

**Integers**

**Example:**

\[ [\text{conc} : (5 + t_f) \cdot (k_c + k_g) + 4, \]
\[ \text{Adv.\,choose} : 1, \]
\[ \text{Adv.\,guess} : 1, \]
\[ \text{RO.o} : k_c + k_g ] \]
Concrete and Abstract Cost: Example

\[ \vdash \{ \top \} \ A(B, C).a \ \{ \top \ | \ [\text{conc} \leftrightarrow t_{\text{conc}}, B.b \leftrightarrow 1] \} \]

where \( B = \text{abs}(T_B) \) is abstract.
Concrete and Abstract Cost: Example

\[
\Gamma \{ \top \} \ A(B, C).a \ \{ \top \ | \ [\text{conc} \mapsto t_{\text{conc}}, B.b \mapsto 1] \} \\
\text{where } B = \text{abs}(T_B) \text{ is abstract.}
\]
Denotational semantics of programs:

Valuation

\[
[s]^\rho_\nu \in D(\mathcal{M} \times \mathbb{N})
\]

Memory

- \(D(\mathcal{M} \times \mathbb{N})\): discrete distributions over memories and cost.
- Valuation \(\rho\) of abstract modules.
  - Must respect restrictions in \(\mathcal{E}\).
Denotational semantics of programs:

Valuation \[ [s]^{\rho \nu} \in D(M \times N) \]

Memory

- \( D(M \times N) \): discrete distributions over memories and cost.
- Valuation \( \rho \) of abstract modules. Must respect restrictions in \( \mathcal{E} \).

Worst-case complexity, \( \mathcal{E} \vdash \{ \phi \} s \{ \psi \mid c \} \) valid if:

\[ \forall \rho : \mathcal{E}. \forall \nu \in \phi. \]

\[ \pi_1([s]^{\rho \nu}) \subseteq \psi \]

\[ \wedge \sup (\pi_2([s]^{\rho \nu})) \leq c[\text{conc}] + \sum_{O.g} c[O.g] \cdot \text{intr}_\rho(O.g) \]
Semantics

- **Denotational semantics** of programs:

  \[ [s]_\rho^\nu \in D(M \times N) \]

  - **Valuation** \( \rho \) of abstract modules.
  - Must respect restrictions in \( \mathcal{E} \).

- **Worst-case complexity**, \( \mathcal{E} \vdash \{ \phi \} \models \{ \psi \mid c \} \) valid if:

  \[
  \forall \rho : \mathcal{E}. \forall \nu \in \phi. \\
  \text{supp}(\pi_1([s]_\rho^\nu)) \subseteq \psi \\
  \wedge \text{sup}(\text{supp}(\pi_2([s]_\rho^\nu))) \leq c[\text{conc}] + \sum_{O.g} c[O.g] \cdot \text{intr}_\rho(O.g)
  \]
We designed a **Hoare logic** for **cost**.

- Many rules are straightforward:
  - **memory** and **cost upper-bound** handled separately.
  
  *Example*: conditional rule.

- More complex rules:
  - simultaneously prove **memory** and **cost upper-bound**.
  
  *Examples*: abstract call and instantiation rules.
Hoare Logic for Cost: If

$$\{\phi\} \quad e \leq t_e \quad \mathcal{E} \vdash \{\phi \land e\} \quad s_1 \quad \{\psi \mid t\} \quad \mathcal{E} \vdash \{\phi \land \neg e\} \quad s_2 \quad \{\psi \mid t\}$$

$$\mathcal{E} \vdash \{\phi\} \quad \text{if } e \text{ then } s_1 \quad \text{else } s_2 \quad \{\psi \mid t + t_e\}$$

Whenever:
- $e$ takes time $\leq t_e$;
- $s_1$, assuming $\phi \land e$, guarantees $\psi$ in time $\leq t$;
- $s_2$, assuming $\phi \land \neg e$, guarantees $\psi$ in time $\leq t$;

then the conditional, assuming $\phi$, guarantees $\psi$ in time $\leq t + t_e$. 
Abstract call rule *without* cost.

(for one oracle $O$ with one procedure $g$)

\[
A : \text{abs}(\text{func}(X). \text{sig proc } f\{\lambda_m\} \text{ end})
\]

\[
\vdash \{\phi\} A(O).f \{\phi\}
\]
**Abstract call rule without cost.**

(for one oracle $O$ with one procedure $g$)

\[
\begin{align*}
A &: \text{abs} (\text{func}(X). \text{sig proc } f \{\lambda_m\} \text{ end}) \\
\text{FV}(\phi) \cap \lambda_m &= \emptyset \\
\downarrow \{\phi\} & A(O).f \{\phi\}
\end{align*}
\]

- **Memory restriction:** $\text{FV}(\phi) \cap \lambda_m = \emptyset$
  
  $\Rightarrow$ ensures that (all pieces of) $A$ preserves $\phi$. 
Abstract call rule without cost.
(for one oracle O with one procedure g)

\[
\begin{align*}
A : \text{abs}(\text{func}(X). \text{sig proc } f\{\lambda_m\} \text{ end}) \\
\text{FV}(\phi) \cap \lambda_m = \emptyset \quad \vdash \{\phi\} \ O.g \ \{\phi\} \\
\vdash \{\phi\} \ A(O).f \ \{\phi\}
\end{align*}
\]

- **Memory restriction:** \( \text{FV}(\phi) \cap \lambda_m = \emptyset \)
  \(\Rightarrow\) ensures that (all pieces of) \(A\) preserves \(\phi\).

- **Premise:** \(\vdash \{\phi\} \ O.g \ \{\phi\}\)
  \(\Rightarrow\) ensures that the oracle preserves \(\phi\).
Abstract call rule with cost.

\[
\begin{align*}
A & : \text{abs} \left( \text{func}(X). \text{sig proc } f \{\lambda_m : \lambda_c \text{ end} \right) \\
FV(\phi) \cap \lambda_m & = \emptyset \\
\lambda_c & = \text{compl}[\text{intr} : K, O.g : K_o] \\
\forall k < K_o, & \vdash \{\phi \ k\} O.g \ \{\phi \ (k + 1) \mid c_o \ k\} \\
\vdash \{\phi \ 0\} A(O).f \ \{\exists k, \phi \ k \land 0 \leq k \leq K_o \mid T_{abs}\}
\end{align*}
\]

where \(T_{abs} = [A.f \mapsto 1] + \sum_{k=0}^{K_o-1} c_o \ k\).
Hoare Logic for Cost

\[ E \vdash \phi \text{ skip } \psi \vdash \phi' \phi' \vdash \psi' \vdash \psi \\
\text{while} \phi \text{ do } E \vdash \phi \text{ end} \]

Rules handling abstract code are the most interesting.

Abs

\[ \begin{align*}
& \text{let } \psi = \varnothing \text{ in } \psi \\
& \text{let } \psi = \varnothing \text{ in } \psi
\end{align*} \]

Conventions: $\gamma$ can be empty (this corresponds to the non-case function).

Figure 6: Abstract call rule for cost judgment.

Figure 23: Instantiation rule for cost judgment.
Hoare Logic for Cost

Figure 6: Abstract call rule for cost judgment.

\[ \begin{align*}
\text{INSTANTIATION} & \quad M_h = \text{func}(g : M) \text{ sig } S_j \text{ restr } \theta \text{ end } \\
E & \vdash \tau \text{ m : erase}_\text{comp}(M_h) \text{ } \text{ fresh in } E \\
\forall f \in \text{proc}(S_j), \quad E, \text{ module } z : \text{absfun}_h(M_h) \vdash (T \text{ m}(z) f (T \tau f)) \\
E \vdash \text{func } (x : S_j) f : \sum_{i=0}^{S_j} f_{\text{proc}(S_j)}(t_i) \cdot [G_x G_i f] \\
\text{where:} & \\
T_{\text{typ}} & = \{ G \vdash t \cdot [G] \} \\
t_f \leq \text{comp } \theta[f] & \quad \forall f \in \text{proc}(S_j), \quad t_f \leq \theta[f]_{x : S_j} \land t_f \leq \theta[f]_{\text{proc}(S_j)}(A, h) \\
E & \vdash \text{func } (x : S_j) f : \sum_{i=0}^{S_j} f_{\text{proc}(S_j)}(t_i) \cdot [G_x G_i f] \\
\text{Conventions:} & \\
\text{intr}(A, h) & \text{ is the intr field in the complexity restriction of the abstract module procedure } A, h \text{ in } E. \\
\end{align*} \]

Figure 23: Instantiation rule for cost judgment.

\[ \begin{align*}
\text{Module path typing } & \quad \Gamma \vdash p : M. \\
\text{Module expression typing } & \quad \Gamma \vdash p : \text{func } (M') \text{ M } \\
\text{Module structure typing } & \quad \Gamma \vdash p_\theta : \text{st } S. \\
\text{Environments typing } & \quad E. \\
\end{align*} \]
Formalization and proof of soundness of our logic. This includes:

- Formalization of the semantics and cost of programs.
  - First formalization of \texttt{EASYCRYPT} module system.
- Subject reduction for module resolution.
  \[\Rightarrow\] Complexity and memory footprint restrictions are preserved.
Hoare logic for cost has been implemented in **EasyCrypt**.

Integrated in **EasyCrypt** ambient higher-order logic.

⇒ meaningful **existential** quantification over abstract code (e.g. $\forall\exists$ statements).

Established the **complexity** of classical examples: BR93, Hashed El-Gamal, Cramer-Shoup.
Application: Universal Composability in EASYCRYPT
Universal Composability

- UC is a **general framework** providing strong security guarantees.

\[ \pi_1 \textbf{UC}-\text{computes} \pi_2 \quad \text{if} \quad \pi_1 \text{ can safely replace } \pi_2 \text{ in any context.} \]

- **Fundamentals properties:** transitivity and composability.
  \[ \Rightarrow \text{ allow for } \textbf{modular} \text{ and } \textbf{composable} \text{ proofs.} \]
Universal Composability

\[ \exists S \in \text{Sim}, \forall Z \in \text{Env}, \]
\[ | \Pr[Z(\pi_1) : \text{true}] - \Pr[Z(\langle \pi_2 \circ S \rangle) : \text{true}] | \leq \epsilon \]
Universal Composability

∃S ∈ Sim[c_{sim}], ∀Z ∈ Env[c_{env}],

\[ |\Pr[Z(\pi_1) : true] - \Pr[Z(\langle\pi_2 \circ S\rangle) : true]| \leq \epsilon \]

- Z is the adversary: its complexity must be bounded.
- if S’s complexity is unbounded, UC key theorems become useless.
Universal Composability: Transitivity

\[ \exists S_{12} \in \text{Sim} \quad \forall \mathcal{Z} \in \text{Env} \]

\[ \exists S_{23} \in \text{Sim} \quad \forall \mathcal{Z} \in \text{Env} \]
Universal Composability: Transitivity

\[ \exists S_{12} \in \text{Sim} \quad \forall Z \in \text{Env} \]

\[ \exists S_{23} \in \text{Sim} \quad \forall Z \in \text{Env} \]

\[ \exists S \in \text{Sim} \quad \forall Z \in \text{Env} \]

precise complexity bounds are crucial here.
Universal Composability: Transitivity

\[ \exists S_{12} \in \text{Sim} \]
\[ \forall Z \in \text{Env} \]
\[ \exists S_{23} \in \text{Sim} \]
\[ \forall Z \in \text{Env} \]
\[ \exists S \in \text{Sim} \]
\[ \forall Z \in \text{Env} \]
Universal Composability: Transitivity

\[ \exists S_{12} \in \text{Sim} \quad \forall Z \in \text{Env} \]
\[ \exists S_{23} \in \text{Sim} \quad \forall Z \in \text{Env} \]
\[ \exists S \in \text{Sim} \quad \forall Z \in \text{Env} \]

\[ \pi_1 \approx \exists S \in \text{Sim} \quad \forall Z \in \text{Env} \]

Precise complexity bounds are crucial here.
Universal Composability: Transitivity

\[ \exists S_{12} \in \text{Sim} \]
\[ \forall Z \in \text{Env} \]
\[ \pi_1 \]
\[ \approx \]
\[ \exists S \in \text{Sim} \]
\[ \forall Z \in \text{Env} \]

\[ \exists S_{23} \in \text{Sim} \]
\[ \forall Z \in \text{Env} \]
\[ \pi_2 \]
\[ \approx \]
\[ \exists S \in \text{Sim} \]
\[ \forall Z \in \text{Env} \]

\[ \pi_3 \]
\[ \approx \]
\[ S_{23} \]
\[ S_{12} \]
Universal Composability: Transitivity

\[ \exists S_{12} \in \text{Sim}[c_{\text{sim}}^{12}] \quad \forall Z \in \text{Env} \]

\[ \exists S_{23} \in \text{Sim}[c_{\text{sim}}^{23}] \quad \forall Z \in \text{Env} \]

\[ \exists S \in \text{Sim}[c_{\text{sim}}^{12} + c_{\text{sim}}^{23}] \quad \forall Z \in \text{Env} \]

⇒ precise complexity bounds are crucial here.
Universal Composability: Transitivity

\[ \exists S_{12} \in \text{Sim}[c_{12}^{\text{sim}}] \]
\[ \forall Z \in \text{Env}[c_{\text{env}}] \]

\[ \exists S_{23} \in \text{Sim}[c_{23}^{\text{sim}}] \]
\[ \forall Z \in \text{Env}[c_{\text{env}} + c_{12}^{\text{sim}}] \]

\[ \exists S \in \text{Sim}[c_{12}^{\text{sim}} + c_{23}^{\text{sim}}] \]
\[ \forall Z \in \text{Env}[c_{\text{env}}] \]

\[ \Rightarrow \text{ precise complexity bounds are crucial here.} \]
Universal Composability: Transitivity

\[ \exists S_{12} \in \text{Sim}[c_{sim}^{12}] \quad \forall Z \in \text{Env}[c_{env}] \]

\[ \exists S_{23} \in \text{Sim}[c_{sim}^{23}] \quad \forall Z \in \text{Env}[c_{env} + c_{sim}^{12}] \]

\[ \exists S \in \text{Sim}[c_{sim}^{12} + c_{sim}^{23}] \quad \forall Z \in \text{Env}[c_{env}] \]

\[ \Rightarrow \text{ precise complexity bounds are crucial here.} \]
Universal Composability in \textsc{EasyCrypt}

- UC formalization in \textsc{EasyCrypt}, with fully mechanized general UC theorems (transitivity, composability).
- Our formalization exploits \textsc{EasyCrypt} machinery:
  - module restrictions for complexity/memory footprint constraints;
  - message passing done through procedure calls.
Application: One-Shot Secure Channel

- **Diffie-Hellman** UC-computes a **Key-Exchange** ideal functionality, assuming DDH.

- **One-Time Pad** + **Key-Exchange** UC-computes a one-show **Secure Channel** ideal functionality.

Final security statements with precise probability and complexity bounds.
Application: One-Shot Secure Channel

- **Diffie-Hellman** UC-computes a **Key-Exchange** ideal functionality, assuming DDH.

- **One-Time Pad + Key-Exchange** UC-computes a one-show **Secure Channel** ideal functionality.

- **Diffie-Hellman + One-Time Pad** UC-computes a one-shot **Secure Channel** ideal functionality, assuming DDH.

- Final security statements with **precise probability** and **complexity bounds**.
Conclusion
Conclusion

- Designed a Hoare logic for worst-case complexity upper-bounds.
- Implemented in EASYCRYPT, embedded in its ambient higher-order logic.
  \[\implies\text{fully mechanized and composable crypto. reductions.}\]
- First formalization of EASYCRYPT module system.
- Main application: UC formalization in EASYCRYPT. Key results (transitivity, composability) and examples (DH+OTP) are fully mechanized.
Conclusion

- Designed a Hoare logic for worst-case complexity upper-bounds.
- Implemented in EASYCRYPT, embedded in its ambient higher-order logic.
  ⇒ fully mechanized and composable crypto. reductions.
- First formalization of EASYCRYPT module system.
- Main application: UC formalization in EASYCRYPT.
  Key results (transitivity, composability) and examples (DH+OTP) are fully mechanized.

Thank you for your attention.
Complexity Judgements: Expressions

Assuming $\phi$, evaluating expression $e$ takes time at most $t_e$. 

Pre-condition: \[\{\phi\} \quad e \leq t_e\] 

Expression
Complexity Judgements: Expressions

Assuming $\phi$, evaluating expression $e$ takes time at most $t_e$.

Example: Cost of an addition:

$$(\phi \Rightarrow |a| \leq N) \Rightarrow (\phi \Rightarrow |b| \leq N) \Rightarrow$$

$$\{\phi\} a \leq t_a \Rightarrow \{\phi\} b \leq t_b \Rightarrow$$

$$\{\phi\} a + b \leq (t_a + t_b + \text{cadd } N)$$