Mechanized Proofs of Adversarial Complexity and Application to Universal Composability

Manuel Barbosa *University of Porto (FCUP) & INESC TEC*
Gilles Barthe *MPI-SP & IMDEA Software Institute*
Benjamin Grégoire *Inria*
Adrien Koutsos *Inria*
Pierre-Yves Strub *Institut Polytechnique de Paris*

Cryptographic Reduction

Cryptographic Reduction $\mathcal{S} \leq_{\textsf{red}} \mathcal{H}$

$\mathcal{S}$ reduces to a hardness hypothesis $\mathcal{H}$ (e.g. DLog, DDH) if:

$$\forall \mathcal{A}. \exists \mathcal{B}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}}(\mathcal{B}) + \epsilon \wedge \mathsf{cost}(\mathcal{B}) \leq \mathsf{cost}(\mathcal{A}) + \delta$$

where ϵ and δ are small.

Advantage of an unbounded adversary is often 1.

$\Rightarrow$ **bounding** $\mathcal{B}$'s resources is **critical**

Mechanizing Cryptographic Reduction

EasyCrypt is a **proof assistant** to verify cryptographic proofs.

In the proof, the adversary against $\mathcal{H}$ is **explicitly constructed**:

$$\forall \mathcal{A}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}}(\mathcal{C}[\mathcal{A}]) + \epsilon \qquad (\dagger)$$

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

Mechanizing Cryptographic Reduction

EasyCrypt is a **proof assistant** to verify cryptographic proofs.

In the proof, the adversary against $\mathcal{H}$ is **explicitly constructed**:

$$\forall \mathcal{A}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}}(\mathcal{C}[\mathcal{A}]) + \epsilon \tag{†}$$

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

Getting a ∀∃ statement

(†) implies that:

$$\forall \mathcal{A}. \exists \mathcal{B}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}}(\mathcal{B}) + \epsilon$$

but this statement is **useless**, since $\mathcal{B}$ is not resource-limited: its advantage is often 1.

Mechanizing Cryptographic Reduction

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

$$\forall \mathcal{A}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}}(\mathcal{C}[\mathcal{A}]) + \epsilon$$

Limitations

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

If $\forall \mathcal{A}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}_1}(\mathcal{C}[\mathcal{A}]) + \epsilon_1$

and $\forall \mathcal{D}.\ \mathsf{adv}_{\mathcal{H}_1}(\mathcal{D}) \leq \mathsf{adv}_{\mathcal{H}_2}(\mathcal{F}[\mathcal{D}]) + \epsilon_2$

then $\forall \mathcal{A}.\ \mathsf{adv}_{\mathcal{S}}(\mathcal{A}) \leq \mathsf{adv}_{\mathcal{H}_2}(\mathcal{F}[\mathcal{C}[\mathcal{A}]]) + \epsilon_1 + \epsilon_2$

Our Contributions

- A **Hoare logic** to prove **worst-case complexity** upper-bounds of **probabilistic** programs.
  $\Rightarrow$ **fully mechanized** cryptographic reductions.
- Implemented in EasyCrypt, embedded in its ambient higher-order logic.
  $\Rightarrow$ meaningful $\forall\exists$ statements: better **composability**.
- Application: **UC** formalization in EasyCrypt.
- First **formalization** of EasyCrypt **module system**.
  (of independent interest)

Hoare Logic for Complexity

Example: Bellare-Rogaway, 93

Concrete Abstract

```
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;
  }
}
```

Inverter

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

Adv

Example: Bellare-Rogaway, 93

Concrete   Abstract

```
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;
  }
}
```

Inverter

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

Adv

```
proc o(r:rand): ptxt
```

RO

Example: Bellare-Rogaway, 93

Concrete   Abstract

Inverter

```
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;
  }
}
```

Adv

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

Log

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

RO

```
proc o(r:rand): ptxt
```

Example: Bellare-Rogaway, 93

Concrete — Abstract

```
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: $|\text{log}| \leq k_c + k_g$

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

Example: Bellare-Rogaway, 93

Concrete Abstract

```
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
```

$\leq k_c$

$\leq k_g$

Adv

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

Log

```
proc o(r:rand): ptxt
```

RO

Property: $|\mathsf{log}| \leq k_c + k_g$

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

Memory: Adv must not access the log in Log

Key Ingredients

- Support programs mixing **concrete** and **abstract** code.
  Example: $\mathsf{Adv}(\mathsf{Log}(\mathsf{RO}))$
- **Complexity** upper-bound requires some program **invariants**.
  Example: $|\mathsf{log}| \leq k_c + k_g$

Key Ingredients

- Support programs mixing **concrete** and **abstract** code.
  Example: Adv(Log(RO))
- **Complexity** upper-bound requires some program **invariants**.
  Example: $|\mathrm{log}| \leq k_c + k_g$

Abstract procedures must be **restricted**:

- **Complexity**: restrict intrinsic cost/number of calls to oracles.
  Example: choose can call $\mathrm{o} \leq k_c$ times.
- **Memory footprint**: some memory areas are off-limit.
  Example: Adv cannot access the log in Log's memory

Module Restrictions

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
}.
```

Module Restrictions

Abstract code modeled as any program implementing some **module signature** (à la ML), with some **restrictions**:

- Module **memory footprint** can be restricted.

```
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
}.
```

Module Restrictions

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.

```
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]
}.
```

Complexity Judgements

Pre-condition Post-condition

$$\mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid c\}$$

Environment Program statement Cost vector

Assuming ϕ, evaluating s guarantees ψ, and takes time at most c.

Complexity Judgements

Pre-condition

Post-condition

$$\mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid c\}$$

Environment

Program statement

Cost vector

Assuming ϕ, evaluating s guarantees ψ, and takes time at most c.

Example: $\mathcal{E} \vdash \{\top\}$ Inverter(Adv,RO).invert $\{|\mathsf{log}| \leq k_c + k_g \mid c\}$

Cost Vectors

Concrete cost

Abstract procedures

$$c ::= [\,\mathsf{conc} : k,\ \mathsf{O}_1.f_1 : k_1, \ldots, \mathsf{O}_l.f_l : k_l\,]$$

Integers

Example:
$$\left[\begin{array}{lll}\mathsf{conc} & : & (5 + t_f)\cdot(k_c + k_g) + 4, \\ \mathsf{Adv.choose} & : & 1, \\ \mathsf{Adv.guess} & : & 1, \\ \mathsf{RO.o} & : & k_c + k_g \end{array}\right.\ ]$$

Hoare Logic for Cost: If Statements

IF

$$\dfrac{\vdash \{\phi\}\ e \leq t_e \qquad \mathcal{E} \vdash \{\phi \wedge e\}\ \mathsf{s}_1\ \{\psi \mid t\} \qquad \mathcal{E} \vdash \{\phi \wedge \neg e\}\ \mathsf{s}_2\ \{\psi \mid t\}}{\mathcal{E} \vdash \{\phi\}\ \textbf{if}\ e\ \textbf{then}\ \mathsf{s}_1\ \textbf{else}\ \mathsf{s}_2\ \{\psi \mid t + t_e\}}$$

Whenever:

- e takes time $\leq t_e$;
- s_1, assuming $\phi \wedge e$, guarantees ψ in time $\leq t$;
- s_2, assuming $\phi \wedge \neg e$, guarantees ψ in time $\leq t$;

then the conditional, assuming ϕ, guarantees ψ in time $\leq t + t_e$.

Hoare Logic for Cost

Skip

$$\frac{}{\mathcal{E} \vdash \{\phi\}\ \mathbf{skip}\ \{\phi \mid 0\}}$$

Weak

$$\frac{\mathcal{E} \vdash \{\phi'\}\ \mathsf{s}\ \{\psi' \mid t'\} \qquad \phi \Rightarrow \phi' \qquad \psi' \Rightarrow \psi \qquad t' \leq t}{\mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t\}}$$

False

$$\frac{}{\mathcal{E} \vdash \{\bot\}\ \mathsf{s}\ \{\psi \mid t\}}$$

Assign

$$\frac{\vdash \{\phi\}\ e \leq t_e}{\mathcal{E} \vdash \{\phi \wedge \psi[x \leftarrow e]\}\ x \leftarrow e\ \{\psi \mid t_e\}}$$

Rand

$$\frac{\vdash \{\phi_0\}\ d \leq t \qquad \phi = (\phi_0 \wedge \forall v \in \mathrm{dom}(d).\psi[x \leftarrow v])}{\mathcal{E} \vdash \{\phi\}\ x \xleftarrow{\$} d\ \{\psi \mid t\}}$$

Seq

$$\frac{\mathcal{E} \vdash \{\phi\}\ \mathsf{s}_1\ \{\phi' \mid t_1\} \qquad \mathcal{E} \vdash \{\phi'\}\ \mathsf{s}_2\ \{\psi \mid t_2\}}{\mathcal{E} \vdash \{\phi\}\ \mathsf{s}_1; \mathsf{s}_2\ \{\psi \mid t_1 + t_2\}}$$

If

$$\frac{\mathcal{E} \vdash \{\phi \wedge e\}\ \mathsf{s}_1\ \{\psi \mid t\} \qquad \mathcal{E} \vdash \{\phi \wedge \neg e\}\ \mathsf{s}_2\ \{\psi \mid t\} \qquad \vdash \{\phi\}\ e \leq t_e}{\mathcal{E} \vdash \{\phi\}\ \mathbf{if}\ e\ \mathbf{then}\ \mathsf{s}_1\ \mathbf{else}\ \mathsf{s}_2\ \{\psi \mid t + t_e\}}$$

While

$$\frac{I \wedge e \Rightarrow c \leq N \qquad \forall k, \mathcal{E} \vdash \{I \wedge e \wedge c = k\}\ \mathsf{s}\ \{I \wedge k < c \mid t(k)\} \qquad \forall k \leq N, \vdash \{I \wedge e \wedge c = k\}\ e \leq t_e(k) \qquad \vdash \{I \wedge \neg e\}\ e \leq t_e(N+1)}{\mathcal{E} \vdash \{I \wedge 0 \leq c\}\ \mathbf{while}\ e\ \mathbf{do}\ \mathsf{s}\ \{I \wedge \neg e \mid \sum_{i=0}^{N} t(i) + \sum_{i=0}^{N+1} t_e(i)\}}$$

Call

$$\frac{\mathsf{args}_{\mathcal{E}}(\mathsf{F}) = \vec{v} \qquad \vdash \{\phi[\vec{v} \leftarrow \vec{e}\,]\}\ \vec{e} \leq t_e \qquad \mathcal{E} \vdash \{\phi\}\ \mathsf{F}\ \{\psi[x \leftarrow \mathsf{ret}] \mid t\}}{\mathcal{E} \vdash \{\phi[\vec{v} \leftarrow \vec{e}\,]\}\ x \leftarrow \mathbf{call}\ \mathsf{F}(\vec{e}\,)\ \{\psi \mid t_e + t\}}$$

Conc

$$\frac{\mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{proc}\ f(\vec{v} : \vec{\tau}) \to \tau_r = \{\_;\ \mathsf{s};\ \mathsf{return}\ r\ \}) \qquad \mathcal{E} \vdash \{\phi\}\ \mathsf{s}\ \{\psi[\mathsf{ret} \leftarrow r] \mid t\} \qquad \vdash \{\psi\}\ r \leq t_{\mathrm{ret}}}{\mathcal{E} \vdash \{\phi\}\ \mathsf{F}\ \{\psi \mid t + t_{\mathrm{ret}}\}}$$

Convention: ret cannot appear in programs (i.e. ret $\notin \mathcal{V}$).

Figure 22: Basic rules for cost judgment.

Abs

$$\frac{\begin{array}{c}\mathsf{f\text{-}res}_{\mathcal{E}}(\mathsf{F}) = (\mathsf{abs}_{\mathsf{open}}\ \mathsf{x})(\vec{\mathsf{p}}).f \\ \mathcal{E}(\mathsf{x}) = \mathsf{abs}_{\mathsf{open}} \mathsf{x} : (\mathsf{func}(\vec{\mathsf{y}} : \_)\ \mathsf{sig}\ \_\ \mathsf{restr}\ \theta\ \mathsf{end}) \\ \theta[f] = \lambda_{\mathrm{m}} \wedge \lambda_{\mathrm{c}} \qquad \lambda_{\mathrm{c}} = \mathsf{compl}[\mathsf{intr} : K, \mathsf{z}_{j_1}.f_1 : K_1, \ldots, \mathsf{z}_{j_l}.f_l : K_l] \\ \mathsf{FV}(I) \cap \lambda_{\mathrm{m}} = \emptyset \qquad \vec{k}\ \text{fresh in}\ I \\ \forall i, \forall \vec{k} \leq (K_1, \ldots, K_l),\ \vec{k}[i] < K_i \to \mathcal{E} \vdash \{I\ \vec{k}\}\ \vec{\mathsf{p}}[j_i].f_i\ \{I\ (\vec{k} + \mathbb{1}_i) \mid t_i\ k\}\end{array}}{\mathcal{E} \vdash \{I\ \vec{0}\}\ \mathsf{F}\ \{\exists \vec{k},\ I\ \vec{k} \wedge \vec{0} \leq \vec{k} \leq (K_1, \ldots, K_l) \mid T_{\mathrm{abs}}\}}$$

$$\textit{where} \quad T_{\mathrm{abs}} = \left\{\mathsf{x}.f \mapsto 1;\ (\mathsf{G} \mapsto \textstyle\sum_{i=1}^{l} \sum_{k=0}^{K_i - 1} (t_i\ k)[\mathsf{G}])_{\mathsf{G} \neq \mathsf{x}.f}\right\}$$

Conventions: $\vec{\mathsf{y}}$ can be empty (this corresponds to the non-functor case).

Figure 6: Abstract call rule for cost judgment.

Instantiation

$$\frac{\begin{array}{c}\mathsf{M_I} = \mathsf{func}(\vec{\mathsf{y}} : \vec{\mathsf{M}})\ \mathsf{sig}\ \mathsf{S_I}\ \mathsf{restr}\ \theta\ \mathsf{end} \\ \mathcal{E} \vdash_{\mathsf{x}} \mathsf{m} : \mathsf{erase}_{\mathsf{compl}}(\mathsf{M_I}) \qquad \vec{\mathsf{z}}\ \text{fresh in}\ \mathcal{E} \\ \forall f \in \mathsf{procs}(\mathsf{S_I}),\ (\mathcal{E}, \mathsf{module}\ \vec{\mathsf{z}} : \mathsf{abs}_{\mathsf{open}}\ \vec{\mathsf{M}} \vdash \{\top\}\ \mathsf{m}(\vec{\mathsf{z}}).f\ \{\top \mid t_f\}) \\ \forall f \in \mathsf{procs}(\mathsf{S_I}),\ t_f \leq_{\mathsf{compl}} \theta[f] \\ \mathcal{E}, \mathsf{module}\ \mathsf{x} = \mathsf{abs}_{\mathsf{open}} : \mathsf{M_I} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid t_s\}\end{array}}{\mathcal{E}, \mathsf{module}\ \mathsf{x} = \mathsf{m} : \mathsf{M_I} \vdash \{\phi\}\ \mathsf{s}\ \{\psi \mid T_{\mathrm{ins}}\}}$$

where:

$$T_{\mathrm{ins}} = \left\{\mathsf{G} \mapsto t_s[\mathsf{G}] + \textstyle\sum_{f \in \mathsf{procs}(\mathsf{S_I})} t_s[\mathsf{x}.f] \cdot t_f[\mathsf{G}]\right\}$$

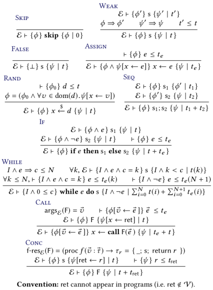
$$t_f \leq_{\mathsf{compl}} \theta[f] = \forall \mathsf{z}_0 \in \vec{\mathsf{z}}, \forall g \in \mathsf{procs}(\vec{\mathsf{M}}[\mathsf{z}_o]),\ t_f[\mathsf{z}_0.g] \leq \theta[f][\mathsf{z}_0.g]\ \wedge$$
$$t_f[\mathsf{conc}] + \sum_{\substack{\mathsf{A} \in \mathsf{abs}(\mathcal{E}) \\ h \in \mathsf{procs}_{\mathcal{E}}(\mathsf{A})}} t_f[\mathsf{A}.h] \cdot \mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h) \leq \theta[f][\mathsf{intr}]$$

Conventions: $\mathsf{intr}_{\mathcal{E}}(\mathsf{A}.h)$ is the intr field in the complexity restriction of the abstract module procedure $\mathsf{A}.h$ in $\mathcal{E}$.

Figure 23: Instantiation rule for cost judgment.

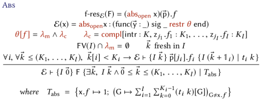
- **Hoare logic** for cost
- **Rules** handling abstract code are the **most interesting**.

Hoare Logic for Cost



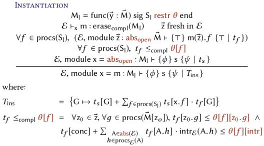
Convention: ret cannot appear in programs (i.e. ret $\notin \mathcal{V}$).

Figure 22: Basic rules for cost judgment.



Conventions: $\vec{y}$ can be empty (this corresponds to the non-functor case).

Figure 6: Abstract call rule for cost judgment.



Conventions: $\mathrm{intr}_{\mathcal{E}}(\mathrm{A}.h)$ is the intr field in the complexity restriction of the abstract module procedure $\mathrm{A}.h$ in $\mathcal{E}$.

Figure 23: Instantiation rule for cost judgment.

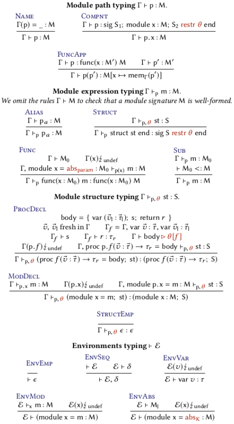


Figure 13: Core typing rules.

- **Hoare logic** for cost + **typing rules** for module restrictions.
- **Rules** handling abstract code are the **most interesting**.

Implementation in EasyCrypt

Mechanizing Cryptographic Reduction

EasyCrypt

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, ...

Implementation in EasyCrypt

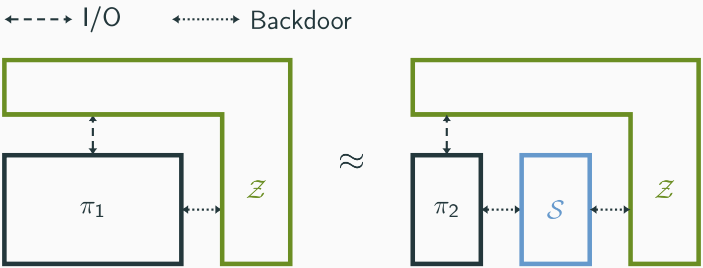
- Hoare logic for cost has been **implemented** in EasyCrypt.
- **Integrated** in EasyCrypt **ambient higher-order logic**.
  $\Rightarrow$ meaningful **existential** quantification over abstract code (e.g. $\forall\exists$ statements).
- Established the **complexity** of **classical examples**:
  BR93, Hashed El-Gamal, Cramer-Shoup.

Application: Universal Composability in EasyCrypt

Universal Composability

- UC is a **general framework** providing strong security guarantees
- **Fundamentals properties**: transitivity and composability.
  $\Rightarrow$ allow for **modular** and **composable** proofs.

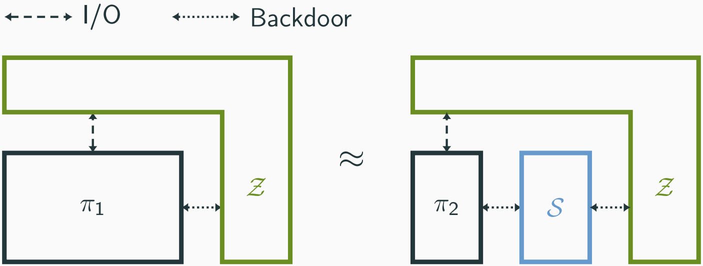
Universal Composability



$$\exists \mathcal{S} \in \mathsf{Sim}, \forall \mathcal{Z} \in \mathsf{Env},$$

$$\left| \mathbf{Pr}[\, \mathcal{Z}(\pi_1) \ : \ \mathsf{true} \,] - \mathbf{Pr}[\, \mathcal{Z}(\langle \pi_2 \circ \mathcal{S} \rangle) \ : \ \mathsf{true} \,] \right| \leq \epsilon$$

Universal Composability



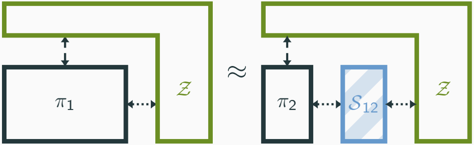
$$\exists \mathcal{S} \in \mathsf{Sim}[c_{\mathsf{sim}}], \forall \mathcal{Z} \in \mathsf{Env}[c_{\mathsf{env}}],$$

$$\left| \mathbf{Pr}[\,\mathcal{Z}(\pi_1) \;:\; \mathsf{true}\,] - \mathbf{Pr}[\,\mathcal{Z}(\langle \pi_2 \circ \mathcal{S} \rangle) \;:\; \mathsf{true}\,] \right| \leq \epsilon$$

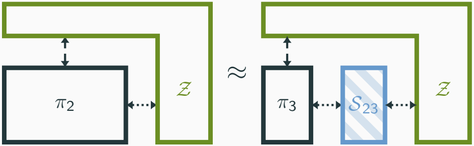
- $\mathcal{Z}$ is the adversary: its complexity must be **bounded**.
- if $\mathcal{S}$'s complexity is unbounded, UC **key theorems** become **useless**.

Universal Composability: Transitivity

$\exists \mathcal{S}_{12} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

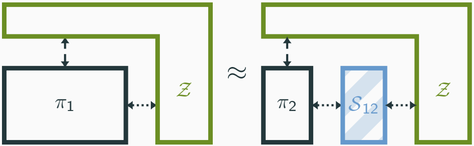


$\exists \mathcal{S}_{23} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

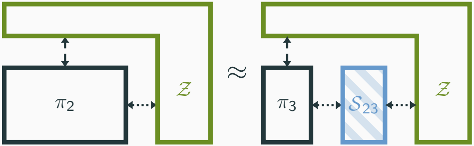


Universal Composability: Transitivity

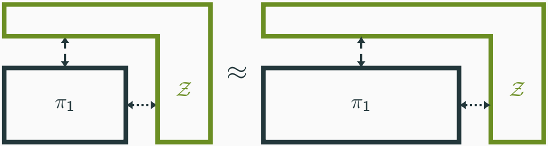
$\exists \mathcal{S}_{12} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$



$\exists \mathcal{S}_{23} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

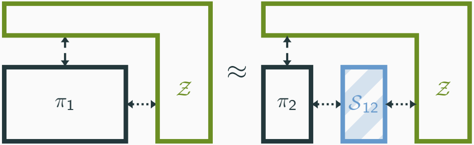


$\exists \mathcal{S} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

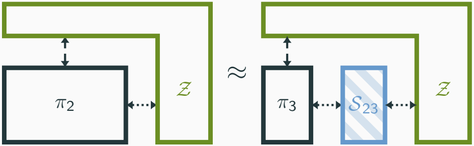


Universal Composability: Transitivity

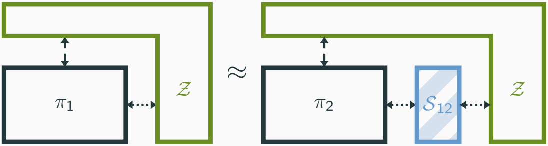
$\exists \mathcal{S}_{12} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$



$\exists \mathcal{S}_{23} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

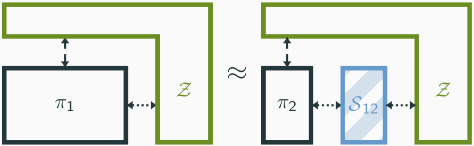


$\exists \mathcal{S} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

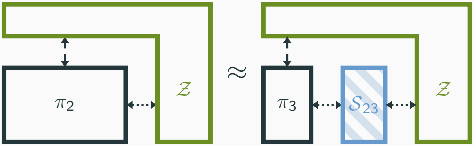


Universal Composability: Transitivity

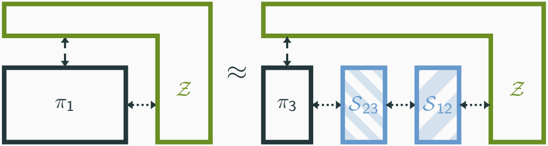
$\exists \mathcal{S}_{12} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$



$\exists \mathcal{S}_{23} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

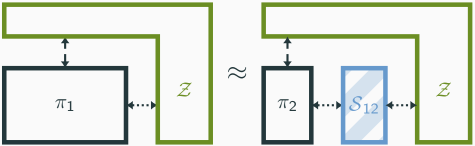


$\exists \mathcal{S} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

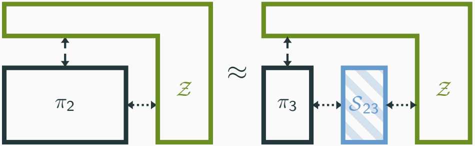


Universal Composability: Transitivity

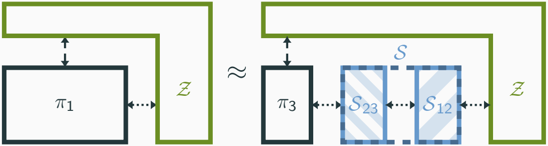
$\exists \mathcal{S}_{12} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$



$\exists \mathcal{S}_{23} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

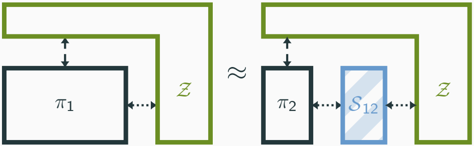


$\exists \mathcal{S} \in \mathsf{Sim}$
$\forall \mathcal{Z} \in \mathsf{Env}$

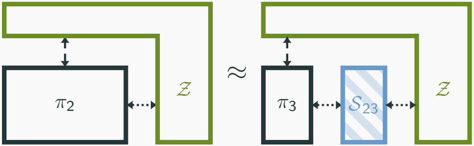


Universal Composability: Transitivity

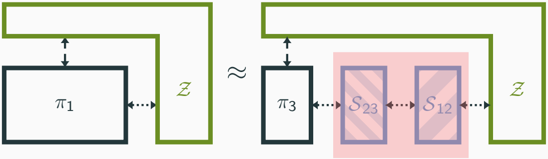
$\exists \mathcal{S}_{12} \in \mathsf{Sim}[c^{12}_{\mathsf{sim}}]$
$\forall \mathcal{Z} \in \mathsf{Env}$



$\exists \mathcal{S}_{23} \in \mathsf{Sim}[c^{23}_{\mathsf{sim}}]$
$\forall \mathcal{Z} \in \mathsf{Env}$



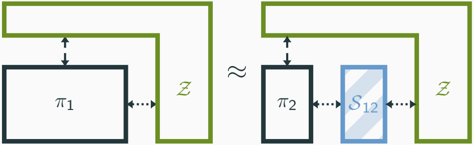
$\exists \mathcal{S} \in \mathsf{Sim}[c^{12}_{\mathsf{sim}} + c^{23}_{\mathsf{sim}}]$
$\forall \mathcal{Z} \in \mathsf{Env}$



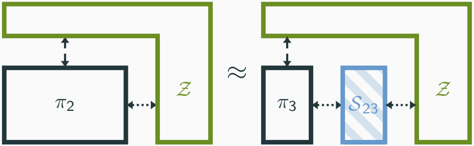
$\Rightarrow$ precise complexity bounds are crucial here.

Universal Composability: Transitivity

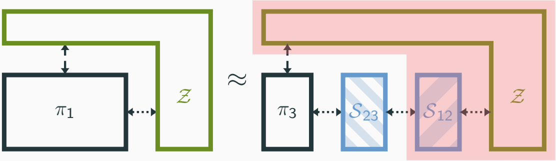
$\exists \mathcal{S}_{12} \in \mathsf{Sim}[c_{\mathsf{sim}}^{12}]$
$\forall \mathcal{Z} \in \mathsf{Env}[c_{\mathsf{env}}]$



$\exists \mathcal{S}_{23} \in \mathsf{Sim}[c_{\mathsf{sim}}^{23}]$
$\forall \mathcal{Z} \in \mathsf{Env}[c_{\mathsf{env}} + c_{\mathsf{sim}}^{12}]$,



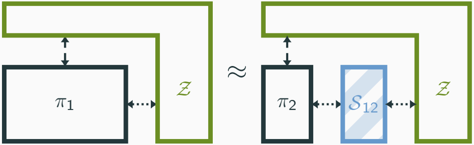
$\exists \mathcal{S} \in \mathsf{Sim}[c_{\mathsf{sim}}^{12} + c_{\mathsf{sim}}^{23}]$
$\forall \mathcal{Z} \in \mathsf{Env}[c_{\mathsf{env}}]$



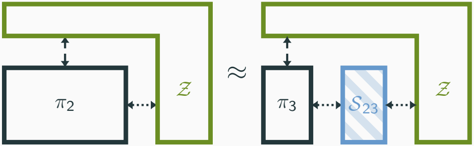
$\Rightarrow$ precise complexity bounds are crucial here.

Universal Composability: Transitivity

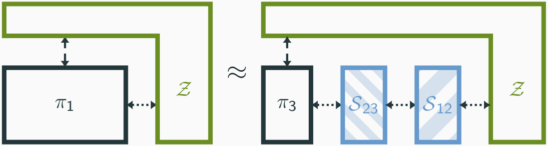
$\exists \mathcal{S}_{12} \in \mathsf{Sim}[c_{\mathsf{sim}}^{12}]$
$\quad \forall \mathcal{Z} \in \mathsf{Env}[c_{\mathsf{env}}]$



$\exists \mathcal{S}_{23} \in \mathsf{Sim}[c_{\mathsf{sim}}^{23}]$
$\quad \forall \mathcal{Z} \in \mathsf{Env}[c_{\mathsf{env}} + c_{\mathsf{sim}}^{12}]$,



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



$\Rightarrow$ precise complexity bounds are crucial here.

Universal Composability in EasyCrypt

- UC **formalization** in EasyCrypt, with **fully mechanized** general UC theorems (transitivity, composability).
- Our formalization **exploits EasyCrypt machinery**:
  - **module restrictions** for complexity/memory footprint constraints;
  - **message passing** done through **procedure calls**.

⇒ **simple** and **usable** formalism.

Application: One-Shot Secure Channel

- Diffie-Hellman UC-emulates a Key-Exchange ideal functionality, assuming DDH.
- One-Time Pad+Key-Exchange UC-emulates a Secure Channel ideal functionality.

Application: One-Shot Secure Channel

- Diffie-Hellman UC-emulates a Key-Exchange ideal functionality, assuming DDH.
- One-Time Pad+Key-Exchange UC-emulates a Secure Channel ideal functionality.
- Diffie-Hellman+One-Time Pad UC-emulates 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.
  $\Rightarrow$ **fully mechanized** and **composable** crypto. reductions.
- First **formalization** of EasyCrypt **module system**.
  (of independent interest)
- Main application: **UC** formalization in EasyCrypt.
  Key results (transitivity, composability) and examples (DH+OTP) are **fully mechanized**.

Thank you for your attention.