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 $S \leq_{\text{red}} H$

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

$$\forall A. \exists B. \ \text{adv}_S(A) \leq \text{adv}_H(B) + \epsilon \land \text{cost}(B) \leq \text{cost}(A) + \delta$$

where $\epsilon$ and $\delta$ are small.

Advantage of an unbounded adversary is often 1.

$\Rightarrow$ bounding $B$’s resources is critical
EasyCrypt is a proof assistant to verify cryptographic proofs. In the proof, the adversary against $\mathcal{H}$ is explicitly constructed:

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

But EasyCrypt lacked support for complexity upper-bounds.
**EasyCrypt** is a **proof assistant** to verify cryptographic proofs.

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

$$\forall A. \text{adv}_S(A) \leq \text{adv}_\mathcal{H}(C[A]) + \epsilon$$  \hspace{1cm} (†)

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

**Getting a $\forall \exists$ statement**

(†) implies that:

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

but this statement is **useless**, since $B$ is not resource-limited: its advantage is often 1.
Hence adversaries *constructed* in reductions are kept *explicit*:

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

**Limitations**

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

If \( \forall A. \text{adv}_S(A) \leq \text{adv}_{\mathcal{H}_1}(C[A]) + \epsilon_1 \)
and \( \forall D. \text{adv}_{\mathcal{H}_1}(D) \leq \text{adv}_{\mathcal{H}_2}(F[D]) + \epsilon_2 \)
then \( \forall A. \text{adv}_S(A) \leq \text{adv}_{\mathcal{H}_2}(F[C[A]]) + \epsilon_1 + \epsilon_2 \)
Our Contributions

- A Hoare logic to prove worst-case complexity upper-bounds of probabilistic programs.
  ⇒ fully mechanized cryptographic reductions.

- Implemented in \texttt{EASYCRYPT}, embedded in its ambient higher-order logic.
  ⇒ meaningful ∀∃ statements: better \textit{composability}.

- Application: \texttt{UC} formalization in \texttt{EASYCRYPT}.

- First formalization of \texttt{EASYCRYPT} module system.
  (of independent interest)
Hoare Logic for Complexity
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
```

**Property**: $|\log| ≤ k_c + k_g$

**Complexity**:
- Concrete: $(5 + t_f) \cdot (k_c + k_g) + 4$
- Abstract: $1$
- RO: $k_c + k_g$

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

**Concrete**

```prolog
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**

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

Adv
```

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

RO
```

**Property:** \[ |log| \leq k_c + k_g \]

**Complexity:**
- **Concrete:** \[ (5 + t) \cdot (k_c + k_g) + 4 \]
- **Abstract:** \[ 1 \]
- **RO:** \[ k_c + k_g \]

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

**Concrete**

```
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;
    }
}
```

**Abstract**

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

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

**Memory:**
Adv must not access the log in Log

**Complexity:**
- \(\text{conc} : (5 + tf) \cdot (kc + kg) + 4\)
- \(\text{Adv} . \text{choose} : 1\)
- \(\text{Adv} . \text{guess} : 1\)
- \(\text{RO} . \text{o} : kc + kg\)
**Example: Bellare-Rogaway, 93**

**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;
    }
}
```

**Property:** \(|\log| \leq k_c + k_g\)

**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\]

---

**Adv**

```
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**

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

**Memory:** Adv must not access the log in Log

---

**Concrete** | **Abstract**
Example: Bellare-Rogaway, 93

**Concrete**

```prolog
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;
    }
}
```

**Property:** $|\log| \leq k_c + k_g$

**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]

**Abstract**

```prolog
proc choose(p:pkey) : unit $\leq k_c$
proc guess(c:ctxt) : unit $\leq k_g$
```

**Adv**

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

**Log**

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

**RO**

**Memory:** Adv must not access the log in Log
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: $\text{Adv} (\text{Log}(\text{RO}))$

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

**Abstract** procedures must be restricted:

- **Complexity**: restrict intrinsic cost/number of calls to oracles.
  Example: choose can call $o \leq k_c$ times.

- **Memory footprint**: some memory areas are off-limit.
  Example: $\text{Adv}$ cannot access the log in $\text{Log}$'s memory
Module Restrictions

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

---

```plaintext
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.

---

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

```ml
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 $c$. 
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

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

Abstract procedures

Example:
\[
\begin{array}{l}
\text{conc} : (5 + t_f) \cdot (k_c + k_g) + 4, \\
\text{Adv.choose} : 1, \\
\text{Adv.guess} : 1, \\
\text{RO.o} : k_c + k_g
\end{array}
\]
Hoare Logic for Cost: If Statements

\[
\text{IF} \quad \vdash \{ \phi \} \ e \leq t_e
\]

\[
\begin{array}{c}
\mathcal{E} \vdash \{ \phi \land e \} \ s_1 \ \{ \psi \mid t \} \\
\mathcal{E} \vdash \{ \phi \land \neg e \} \ s_2 \ \{ \psi \mid t \}
\end{array}
\]

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

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 \( \leq t + t_e \).
Hoare Logic for Cost

### Hoare Logic for Cost

#### Rules handling abstract code are the most interesting.
Hoare Logic for Cost

**Module path typing** $\Gamma \vdash p : M$.

**Comprehend** $\Gamma \vdash p : \text{sig } S : M \text{ module } x : M ; S \text{ restr } \emptyset$ end

$\Gamma \vdash p : M$.

**Function expression typing** $\Gamma \vdash p : M$.

We omit the rules $\Gamma \vdash p : M$ to check that a module signature $M$ is well-formed.

**Alias**

$\Gamma \vdash p : M$.

**Struct** $\Gamma \vdash p \text{ struct } \text{ st } : \text{ sig } S \text{ restr } \emptyset$ end

$\Gamma \vdash p : M.$

**Conventions** $\tilde{f}$ can be empty (this corresponds to the non-functor case).

**Figure 6: Abstract call rule for cost judgment.**

**Instantiation**

$M_0 = \text{func}(\tilde{q} ; \tilde{N}) \text{ sig } S_0 \text{ restr } \emptyset$ end

$\Gamma \vdash p : m \vdash \text{erase}(M_0) : \underbrace{\text{fresh} \in E}_{\text{E}}$

$\forall f \in \text{proc}(S_0), (E, \text{module } x \vdash \text{abrupt }, M_0 + \{ T \} \text{ m}(\tilde{F}), f \vdash \{ T \} \{ f \})$

$\exists E, \text{module } x \vdash \text{abrupt }, M_0 + \{ T \} \text{ m}(\tilde{F}) f \vdash \{ T \} \{ f \}$

where:

$\Gamma \vdash p : m \vdash \{ T \} \{ f \}$

Conventions: The $\text{intr } (A, h)$ is the inner field in the complexity restriction of the abstract module procedure $A, h$ in $E$.

**Figure 23: Instantiation rule for cost judgment.**

**Core typing rules.**

- **Hoare logic for cost + typing rules for module restrictions.**
- **Rules handling abstract code are the most interesting.**
Implementation in **EasyCrypt**
**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, ...
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.
- **Fundamentals properties**: transitivity and composability. ⇒ allow for modular and composable proofs.
$\exists S \in \text{Sim}, \forall Z \in \text{Env},$

$$\left| \Pr[Z(\pi_1) : \text{true}] - \Pr[Z(\langle \pi_2 \circ S \rangle) : \text{true}] \right| \leq \epsilon$$
Universal Composability

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

- \( Z \) is the adversary: its complexity must be **bounded**.
- If \( S \)'s complexity is unbounded, UC key theorems become **useless**.
Universal Composability: Transitivity

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

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

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

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

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

precise complexity bounds are crucial here.
$\exists S_{12} \in \text{Sim} \quad \forall Z \in \text{Env}$

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

$\exists S \in \text{Sim} \quad \forall Z \in \text{Env}$
Universal Composability: Transitivity

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

\[ \pi_1 \quad \sim \quad \pi_2 \approx S_{12} \quad \approx \exists S \in \text{Sim} \]

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

\[ \pi_2 \quad \sim \quad \pi_3 \approx S_{23} \quad \approx \exists S \in \text{Sim} \]

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

\[ \pi_1 \quad \sim \quad \pi_3 \approx S_{23} \approx S_{12} \quad \approx \exists S \in \text{Sim} \]

Precise complexity bounds are crucial here.
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

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

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

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

⇒ precise complexity bounds are crucial here.
\[ \exists S_{12} \in \text{Sim}[c_{sim}^{12}] \quad \forall Z \in \text{Env}[c_{env}] \]

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

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

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

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

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

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

⇒ 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.
  \[\Rightarrow\] 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.
  ⇒ 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.