Mechanized Proofs of Adversarial Complexity and Application to Universal Composability

Annual Meeting of the WG "Formal Methods for Security", Fréjus

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

23 March, 2022
Cryptographic Reduction $\leq_{\text{red}}$.

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

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:

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

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:

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

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

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

(†) implies that:

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

but this statement is **useless**, since \( B \) is not resource-limited: its advantage is often 1.
Hence adversaries \textit{constructed} in reductions are kept \textit{explicit}:

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

\textbf{Limitations}

\begin{itemize}
\item \textbf{Not fully verified}: \(C[A]\)’s complexity is checked manually.
\item \textbf{Less composable}, as composition is done manually (inlining).
\end{itemize}

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

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

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

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

- Implemented in EasyCrypt, embedded in its ambient higher-order logic.
  ⇒ meaningful ∀∃ 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**

```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**:

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

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

**RO**

\[
\text{RO.o(r)}:
\]

**Memory**:

Adv must not access the log in Log

---

Inverter
**Example: Bellare-Rogaway, 93**

**Concrete**

```
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**

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

**Adv**

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

**RO**

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

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

Log
\[
\text{proc o}(r:rand): \text{ctxt} = \{ \\
\quad \text{log } \leftarrow r :: \text{log}; \\
\quad \text{return RO.o(r);} \\
\}\]

RO
\[
\text{proc o}(r:rand): \text{ctxt}
\]
**Inverter**

```plaintext
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:**
- $\text{conc} : (5 + t_f) \cdot (k_c + k_g) + 4$
- $\text{Adv.choose} : 1$
- $\text{Adv.guess} : 1$
- $\text{RO.o} : k_c + k_g$

**Proc choose**

```plaintext
proc choose(p:pkey) : unit
```

$\leq k_c$

**Proc guess**

```plaintext
proc guess(c:ctxt) : unit
```

$\leq k_g$

**Proc o**

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

**Adv**

**Log**

**RO**
Example: Bellare-Rogaway, 93

**Concrete**

```plaintext
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**

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

**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:** \text{Adv} must not access the log in \text{Log}
Support programs mixing **concrete** and **abstract** code.

Example: \( \text{Adv}(\log(\text{RO})) \)

**Complexity** upper-bound requires some program **invariants**.

Example: \(|\log| \leq k_c + k_g\)
Key Ingredients

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

- **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: $\text{choose}$ 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
Abstract code modeled as any program implementing some module signature (à la ML)

```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, -H, -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.

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

module type Adv (H: RO) {+all mem, -Log, -H, -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: $\mathcal{E} \vdash \{ T \} \ Inverter(\text{Adv,RO}).\text{invert} \ \{ \| \log \| \leq k_c + k_g \ | \ c \}$
Cost Vectors

Concrete cost

Abstract procedures

Integers

c ::= [conc : k, O₁.f₁ : k₁, ..., Oᵢ.fᵢ : kᵢ]

Example: [conc : (5 + tᵣ) · (kᶜ + kᵣ) + 4,
Adv.choose : 1,
Adv.guess : 1,
RO.o : kᶜ + kᵣ]
Hoare Logic for Cost: If Statements

\[
\text{IF} \\
\begin{align*}
\vdash \{\phi\} & \quad e \leq t_e \\
\mathcal{E} & \vdash \{\phi \land e\} \; s_1 \; \{\psi \mid t\} & \mathcal{E} & \vdash \{\phi \land \neg e\} \; s_2 \; \{\psi \mid t\} \\
\mathcal{E} & \vdash \{\phi\} \text{ if } e \text{ then } s_1 \text{ else } s_2 \; \{\psi \mid t + t_e\}
\end{align*}
\]

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

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

Abs

\[ \begin{align*}
\text{true} & \vdash e_0 \rightarrow e_0 \\
\text{false} & \vdash \phi \rightarrow \psi \\
\text{skip} & \vdash \phi \rightarrow \phi \\
\text{let} & \vdash \phi \rightarrow \phi' \\
\text{if_then_else} & \vdash \phi \rightarrow \phi' \\
\text{while} & \vdash \phi \rightarrow \psi \\
\text{fun} & \vdash \phi \rightarrow \psi \\
\text{proc} & \vdash \phi \rightarrow \psi \\
\text{struct} & \vdash \phi \rightarrow \psi \\
\text{seq} & \vdash \phi \rightarrow \psi \\
\text{loop} & \vdash \phi \rightarrow \psi \\
\text{call} & \vdash \phi \rightarrow \psi \\
\end{align*} \]

Inst

\[ \begin{align*}
M_0 & = \text{fun}(\emptyset \rightarrow N_0) \quad \text{sig} \quad S_0 \quad \text{restr} \quad \theta \quad \text{end} \\
M_0 & = \text{fun}(\emptyset \rightarrow N_0) \quad \text{sig} \quad S_0 \quad \text{restr} \quad \theta \quad \text{end} \\
\text{by} & \vdash \phi \rightarrow \psi \\
\text{let} & \vdash \phi \rightarrow \psi \\
\text{while} & \vdash \phi \rightarrow \psi \\
\text{fun} & \vdash \phi \rightarrow \psi \\
\text{proc} & \vdash \phi \rightarrow \psi \\
\text{struct} & \vdash \phi \rightarrow \psi \\
\end{align*} \]

Conventions:

They can be empty (this corresponds to the non-functor case).

Figure 6: Abstract call rule for cost judgment.

Mod

\[ \begin{align*}
\Gamma(p) & = \emptyset \quad \text{M} \\
\Gamma & = p : \text{M} \\
\Gamma & = p : \text{M} \\
\Gamma & = p : \text{M} \\
\Gamma & = p : \text{M} \\
\Gamma & = p : \text{M} \\
\end{align*} \]

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**
**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. ⇒ allow for **modular** and **composable** proofs.
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: **Diffie-Hellman+One-Time Pad** UC-emulates a one-shot **Secure Channel** ideal functionality, assuming DDH.
Conclusion
Conclusion

- Designed a **Hoare logic** for **worst-case** complexity upper-bounds.

- Implemented in **EASYCRYPT**, embedded in its ambient higher-order logic.
  \[ \Rightarrow \text{fully mechanized and composable crypto. reductions.} \]

- 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.
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

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

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

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

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

\[ \pi_1 \quad \Rightarrow \quad \pi_2 \quad \approx \quad S_{12} \]

\[ \pi_2 \quad \Rightarrow \quad \pi_3 \quad \approx \quad S_{23} \]

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

precise complexity bounds are crucial here.
Universal Composability: Transitivity

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

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

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

precise complexity bounds are crucial here.
Universal Composability: Transitivity

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

\[ \rho_{1} \xRightarrow{\pi_{1}} \mathcal{Z} \equiv \rho_{2} \xRightarrow{S_{12}} \mathcal{Z} \]

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

\[ \rho_{2} \xRightarrow{\pi_{2}} \mathcal{Z} \equiv \rho_{3} \xRightarrow{S_{23}} \mathcal{Z} \]

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

\[ \rho_{1} \xRightarrow{\pi_{1}} \mathcal{Z} \equiv \rho_{2} \xRightarrow{S_{12}} \mathcal{Z} \Rightarrow \text{precise complexity bounds are crucial here.} \]
Universal Composability: Transitivity

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

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

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

\[ \forall Z \in \text{Env} \]

precise complexity bounds are crucial here.
Universal Composability: Transitivity

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

\[ \pi_1 \quad \approx \quad Z \]

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

\[ \pi_2 \quad \approx \quad S_{12} \quad \approx \quad Z \]

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

\[ \pi_2 \quad \approx \quad Z \]

\[ \pi_3 \quad \approx \quad S_{23} \quad \approx \quad Z \]

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

\[ \pi_1 \quad \approx \quad Z \]

\[ \pi_3 \quad \approx \quad S_{23} \quad S_{12} \quad \approx \quad Z \]

Precise complexity bounds are crucial here.
Universal Composability: Transitivity

$$\exists S_{12} \in \text{Sim}[c_{\text{sim}}^{12}]$$
$$\forall \pi_1 \in \text{Env}$$

$$\exists S_{23} \in \text{Sim}[c_{\text{sim}}^{23}]$$
$$\forall \pi_2 \in \text{Env}$$

$$\exists S \in \text{Sim}[c_{\text{sim}}^{12} + c_{\text{sim}}^{23}]$$
$$\forall \pi_3 \in \text{Env}$$

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

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

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

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

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

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

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

⇒ precise complexity bounds are crucial here.
Application: One-Shot Secure Channel

- **Diffie-Hellman** UC-emulates a **Key-Exchange** ideal functionality, assuming DDH.
- **Key-Exchange+One-Time Pad** UC-emulates a one-shot **Secure Channel** ideal functionality.
Application: One-Shot Secure Channel

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

- **Key-Exchange+One-Time Pad** UC-emulates a one-shot **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*. 