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:

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

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**:

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

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

**Getting a $∀∃$ statement**

(†) implies that:

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

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

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

**Limitations**

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

If

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

and

\[
\forall D. \ \text{adv}_H(D) \leq \text{adv}_H(F[D]) + \epsilon_2
\]

then

\[
\forall A. \ \text{adv}_S(A) \leq \text{adv}_H(F[C[A]]) + \epsilon_1 + \epsilon_2
\]
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
\begin{align*}
\textbf{proc} & \text{ invert}(pk:pkey,y:rand): \text{ rand } = \{ \\
& \quad \text{ log } \leftarrow []; \\
& \quad \text{ Adv.choose}(pk); \\
& \quad h \leftarrow\text{dptxt}; \\
& \quad \text{ Adv.guess}(y \| h); \\
& \quad \textbf{while} (\text{log} \neq []) \{ \\
& \quad \quad \text{ r } \leftarrow \text{head log}; \\
& \quad \quad \text{ if } (f \text{ pk } r = y) \textbf{return } r; \\
& \quad \quad \text{ log } \leftarrow \text{tail log}; \\
& \quad \}\}
\end{align*}

\begin{align*}
\textbf{proc} & \text{ choose}(p:pkey) : \text{ unit} \\
\textbf{proc} & \text{ guess}(c:ctxt) : \text{ unit}
\end{align*}
**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;
  }
}
```

Inverter

Abstract

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

Adv

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

RO
Example: Bellare-Rogaway, 93

**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;
  }
}
```

**Adv**

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

**Log**

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

**RO**

```plaintext
proc o(r:rand): ptxt
```
Example: Bellare-Rogaway, 93

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

\text{Inverter}

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**

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

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

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

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

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

- **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: \( \text{choose} \) 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)

```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.

```plaintext
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.

```haskell
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 $c$. 

$\mathcal{E} \vdash \{ \phi \} \mathcal{E} \uplus \{ \psi \mid c \}$
Complexity Judgements

Assuming $\phi$, evaluating $s$ guarantees $\psi$, and takes time at most $c$.

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

Concrete cost

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

Abstract procedures

Integers

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

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

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

Rules handling abstract code are the most interesting.

- Hoare logic for cost + typing rules for module restrictions.

Figure 6: Abstract call rule for cost judgment.

Figure 22: Basic rules for cost judgment.

Figure 23: Instanlation rule for cost judgment.

Figure 13: Core typing rules.
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.
  ⇒ 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
UC is a **general framework** providing strong security guarantees.

**Fundamentals properties:** transitivity and composability.  
⇒ allow for modular and composable proofs.
∃S ∈ Sim, ∀Z ∈ Env,

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

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

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

\[ \pi_1 \quad \mathcal{Z} \quad \pi_2 \quad S \]
Universal Composability: Transitivity

∃S_{12} ∈ Sim
∀Z ∈ Env

∃S_{23} ∈ Sim
∀Z ∈ Env
Universal Composability: Transitivity

∃S_{12} ∈ Sim
∀Z ∈ Env

∃S_{23} ∈ Sim
∀Z ∈ Env

∃S ∈ Sim
∀Z ∈ Env

precise complexity bounds are crucial here.
Universal Composability: Transitivity

∃S_{12} ∈ Sim
∀Z ∈ Env

∃S_{23} ∈ Sim
∀Z ∈ Env

∃S ∈ Sim
∀Z ∈ Env

precise complexity bounds are crucial here.
Universal Composability: Transitivity

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

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

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

precise complexity bounds are crucial here.

[Diagram showing relationships between systems and environments, with symbols \( \pi \), \( S \), and \( Z \).]
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

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

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

⇒ 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

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

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

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

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

- UC formalization in \texttt{EASYCRYPT}, with fully mechanized general UC theorems (transitivity, composability).
- Our formalization exploits \texttt{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.