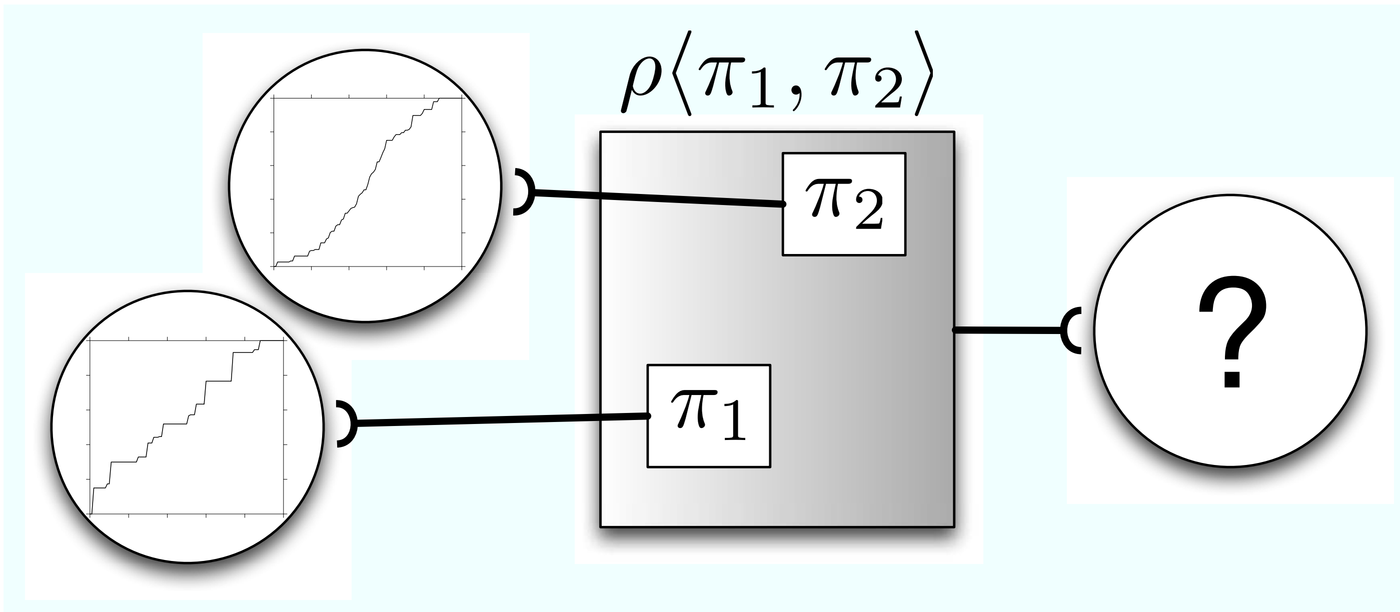
Is Cryptography Going to Be an Engineering Discipline?

Sven Laur
University of Tartu

swen@math.ut.ee

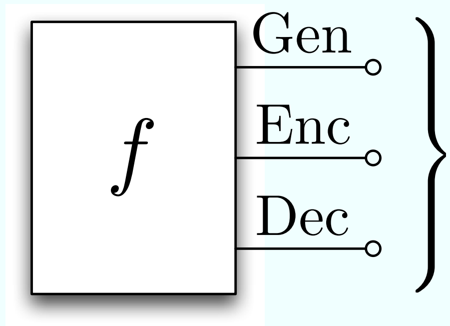
What is a cryptographic proof?

Cryptographic proof manipulates objects with abstract properties



- ▷ Does the proof provide an optimal upper bounds?
- ▷ Is the construction itself optimal?
- ▷ Are there any alternative solutions with different primitives?

What is a cryptographic primitive?



Abstract encryption primitive

$$\forall(\mathsf{pk},\mathsf{sk}) \leftarrow \mathrm{Gen}, \forall m \in \mathcal{M}: \quad \mathrm{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{pk}}(\mathrm{m})) = m$$

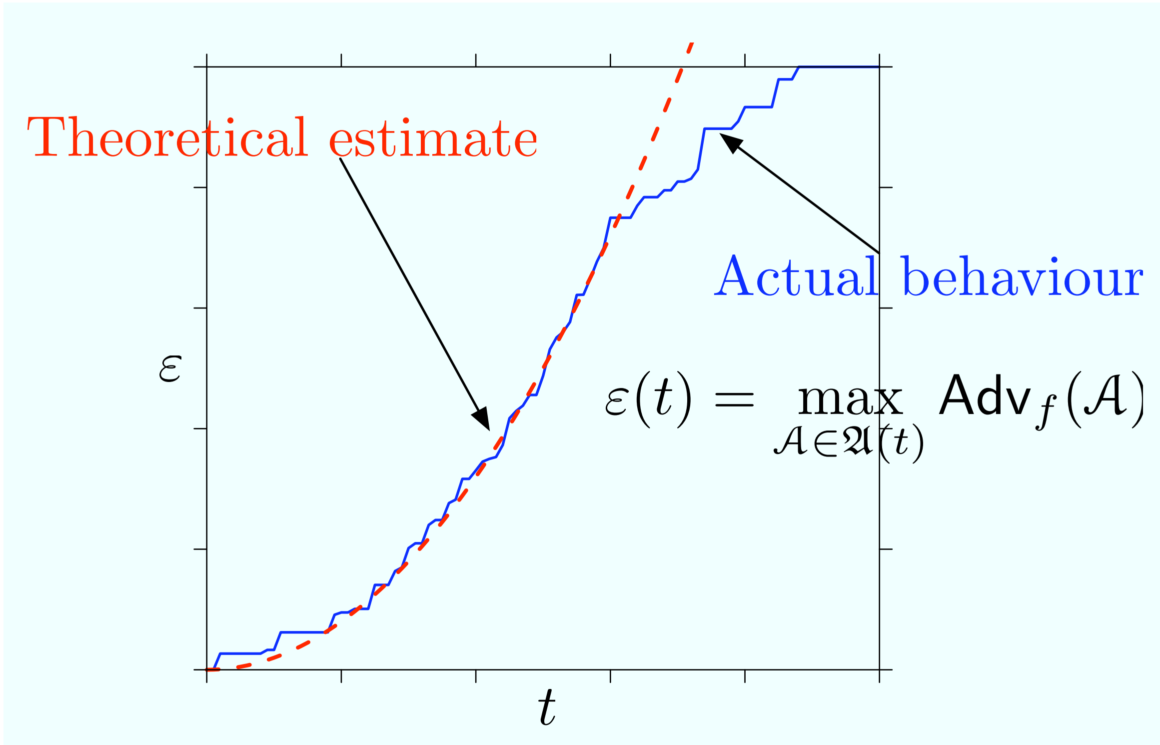
- ▷ A primitive is a black-box object that provides certain services.
- ▷ Objects returned by the primitive are from an abstract (algebraic) domain.
- ▷ Only way to convert outputs to something useful is to use the functions of the primitive to convert inputs from one domain to the other.
- ▷ **These restrictions do not apply to potential adversaries.**

A security game

$$
\begin{aligned}
&(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathrm{Gen} \\
&(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathrm{Gen} \\
&(m_0, m_1) \leftarrow \mathcal{A} \\
&b_0, b_1 \leftarrow \{0,1\} \\
&c_0 \leftarrow \mathrm{Enc}_{\mathsf{pk}_0}(m_{b_0}) \\
&c_1 \leftarrow \mathrm{Enc}_{\mathsf{pk}_1}(m_{b_1}) \\
&q \leftarrow \mathcal{A}(c_0, c_1) \\
&\mathrm{iseq} \leftarrow \mathcal{A}(\mathsf{sk}_q) \\
&\mathsf{return}\ [(b_0 \stackrel{?}{=} b_1) \stackrel{?}{=} \mathrm{iseq}]
\end{aligned}
$$

$$\mathsf{Adv}_{\mathcal{G}_0}(\mathcal{A}) = \Pr\left[\mathcal{G}_0^{\mathcal{A}} = 1\right]$$

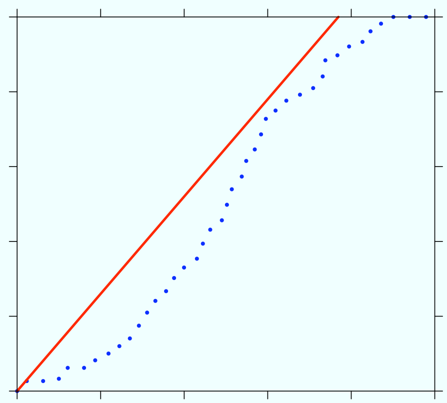
Security of a primitive



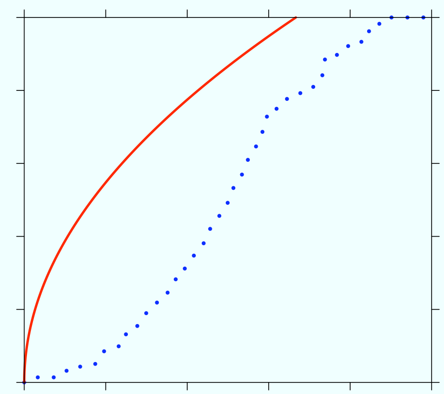
A cryptographic primitive is characterised by a time-success profile $\varepsilon(t)$ that is quantified as a maximal success probability in a certain game.

Proofs by reductions

A classical way to prove security of a derived primitive is to transform a successful adversary $\mathcal{A}$ against the primitive to a new adversary $\mathcal{B}$ against one of the primary primitives.



$$\mathcal{A} \mapsto \mathcal{B} : \begin{cases} \varepsilon_1 \geq \rho(\varepsilon_2) \\ t_1 \leq \tau(t_2) \end{cases}$$



Usually, we need to do a lengthy and detailed probability calculations in order to find the quantitative properties of a reduction.

Drawbacks of direct reductions

Direct probability computations

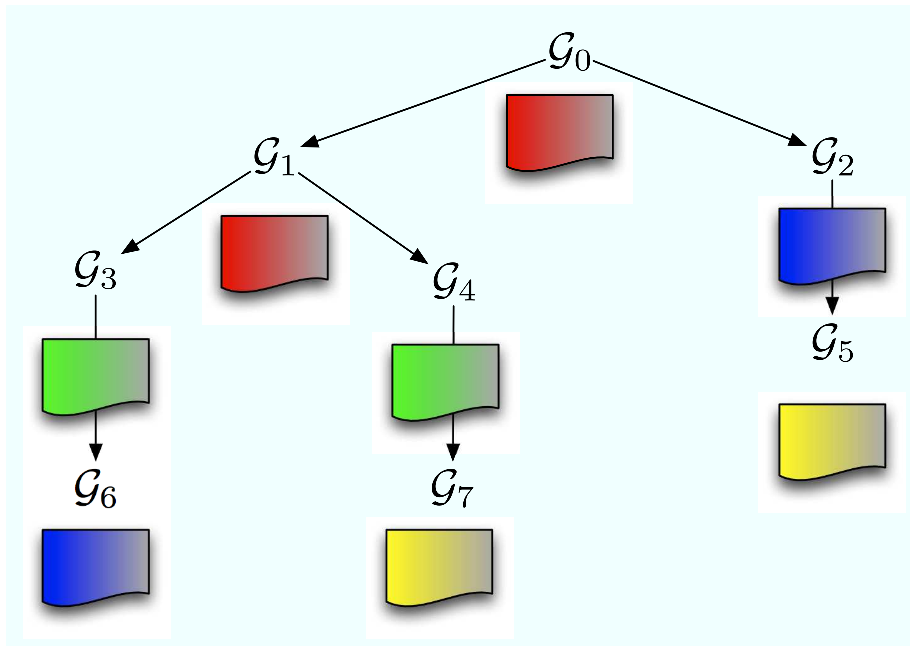
- ▷ Analysis of randomised algorithms is technical.
- ▷ Most of us cannot correctly operate with probabilities.
- ▷ Verification of these calculations is equivalent to the derivation of them.

Proofs are unstructured

- ▷ To verify a proof, one must debug a complex algorithm.
- ▷ Proofs are several pages long even for simple problems.
- ▷ Analysis of a full-blown system could be hundreds of pages long.

Game-playing proofs ≡ Structured proofs

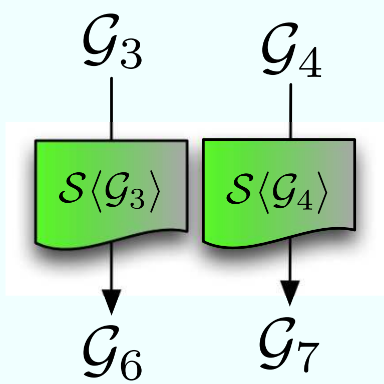
Complex proofs can be represented by game trees.



- ◇ Structured proof reveals many repeated arguments.
- ◇ Probability calculations can be automated.

Proof compaction $\equiv$ Reduction schemata

We can use a single meta-proof and instantiate for every possible sub-proof.



$\langle\mathcal{G}_{\mathrm{s}}\rangle \xRightarrow{\mathcal{T}} \langle\mathcal{G}_{\mathrm{t}}\rangle$

$\mathcal{P}$ – a precondition on a source game

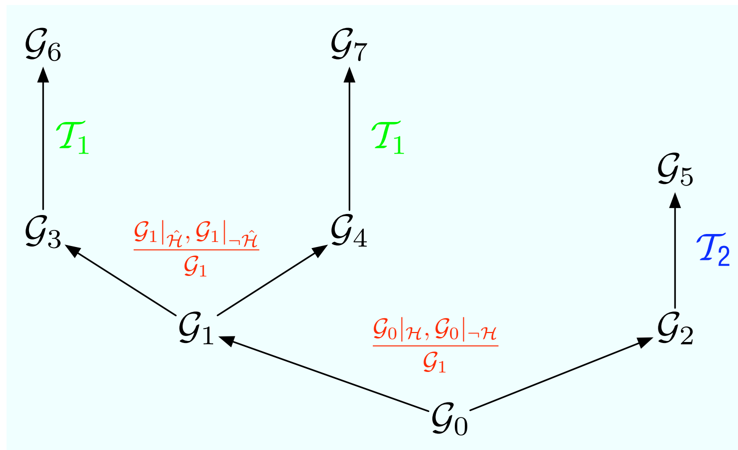
$\mathcal{T}$ – a code transformation rule

$\mathcal{S}$ – the resulting closeness guarantee

- ▷ Construction and analysis of randomised algorithms is abstracted away.
- ▷ It is possible to support parametrised reductions.
- ▷ Application of reduction schemata happens on the syntactical level.

A final compacted proof

The final compacted proof tree can be checked syntactically, except for preconditions of reduction schemata. These must be verified separately.



Proof phases

◇ Primitive elimination phase – few well-documented reduction schemata.

◇ Analysis of combinatorial games – many informal code transformations.

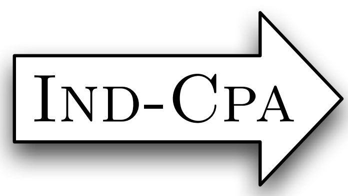
Primitive elimination

It must be possible to eliminate all primitives.

- ▷ For each abstract function there must be an elimination rule.
- ▷ Usually, there are many rules for an abstract function.
- ▷ All preconditions can be formalised through reachability and dependencies

Example. $\textsc{Ind-Cpa}$ **reduction schema**

$$
\begin{array}{|l}
(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen} \\
\ldots \\
m_0 \leftarrow \ldots \\
\ldots \\
m_1 \leftarrow \ldots \\
c \leftarrow \mathrm{Enc}_{\mathsf{pk}}(m_0) \\
\ldots
\end{array}
$$



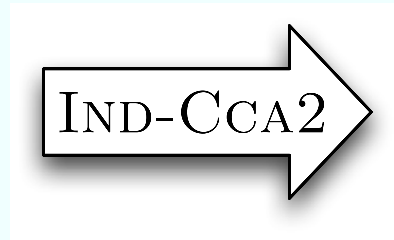
$$
\begin{array}{|l}
(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen} \\
\ldots \\
m_0 \leftarrow \ldots \\
\ldots \\
m_1 \leftarrow \ldots \\
c \leftarrow \mathrm{Enc}_{\mathsf{pk}}(m_1) \\
\ldots
\end{array}
$$

Reduction is applicable when:

- ▷ No variables accessible by the adversary $\mathcal{A}$ depend on sk.
- ▷ No $\mathrm{Dec}_{\mathsf{sk}}(\cdot)$ calls are made during the game.

Example. $\textsc{Ind-Cca2}$ **reduction schema**

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen}$
$\ldots$
$m_0 \leftarrow \ldots$
$\ldots$
$m_1 \leftarrow \ldots$
$c \leftarrow \mathrm{Enc}_{\mathsf{pk}}(m_0)$
$\ldots$



$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathrm{Gen}$
$\ldots$
$m_0 \leftarrow \ldots$
$\ldots$
$m_1 \leftarrow \ldots$
$c \leftarrow \mathrm{Enc}_{\mathsf{pk}}(m_1)$
$\ldots$

<u>Reduction is applicable when:</u>

- ▷ No variables accessible by the adversary $\mathcal{A}$ depend on sk.
- ▷ No $\mathrm{Dec}_{\mathsf{sk}}(c)$ calls are made after reaching line $c \leftarrow \mathrm{Enc}_{\mathsf{pk}}(m_0)$.

Why branching is unavoidable

$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathrm{Gen}$
$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathrm{Gen}$
$(m_0, m_1) \leftarrow \mathcal{A}$
$b_0, b_1 \leftarrow \{0,1\}$
$c_0 \leftarrow \mathrm{Enc}_{\mathsf{pk}_0}(m_{b_0})$
$c_1 \leftarrow \mathrm{Enc}_{\mathsf{pk}_1}(m_{b_1})$
$q \leftarrow \mathcal{A}(c_0, c_1)$
$\mathrm{iseq} \leftarrow \mathcal{A}(\mathsf{sk}_q)$

$\underline{q = 0}$
$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathrm{Gen}$
$(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathrm{Gen}$
$(m_0, m_1) \leftarrow \mathcal{A}$
$b_0, b_1 \leftarrow \{0,1\}$
$c_0 \leftarrow \mathrm{Enc}_{\mathsf{pk}_0}(m_{b_0})$
$c_1 \leftarrow \mathrm{Enc}_{\mathsf{pk}_1}(m_{b_1})$
$\mathrm{iseq} \leftarrow \mathcal{A}(\mathsf{sk}_0, c_0, c_1)$

$\underline{q = 1}$
$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathrm{Gen}$
$(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathrm{Gen}$
$(m_0, m_1) \leftarrow \mathcal{A}$
$b_0, b_1 \leftarrow \{0,1\}$
$c_0 \leftarrow \mathrm{Enc}_{\mathsf{pk}_0}(m_{b_0})$
$c_1 \leftarrow \mathrm{Enc}_{\mathsf{pk}_0}(m_{b_1})$
$\mathrm{iseq} \leftarrow \mathcal{A}(\mathsf{sk}_1, c_0, c_1)$

$\underline{q = 0}$
$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathrm{Gen}$
$(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathrm{Gen}$
$(m_0, m_1) \leftarrow \mathcal{A}$
$b_0, b_1 \leftarrow \{0,1\}$
$c_0 \leftarrow \mathrm{Enc}_{\mathsf{pk}_0}(m_{b_0})$
$c_1 \leftarrow \mathrm{Enc}_{\mathsf{pk}_1}(m_{b_0})$
$\mathrm{iseq} \leftarrow \mathcal{A}(\mathsf{sk}_0, c_0, c_1)$

$\underline{q = 1}$
$(\mathsf{pk}_0, \mathsf{sk}_0) \leftarrow \mathrm{Gen}$
$(\mathsf{pk}_1, \mathsf{sk}_1) \leftarrow \mathrm{Gen}$
$(m_0, m_1) \leftarrow \mathcal{A}$
$b_0, b_1 \leftarrow \{0,1\}$
$c_0 \leftarrow \mathrm{Enc}_{\mathsf{pk}_0}(m_{b_1})$
$c_1 \leftarrow \mathrm{Enc}_{\mathsf{pk}_0}(m_{b_1})$
$\mathrm{iseq} \leftarrow \mathcal{A}(\mathsf{sk}_1, c_0, c_1)$

Benefits and hurdles

What does such a proof system give?

- ▷ Eliminates need for probability calculations.
- ▷ Eliminates need for creative steps.
- ▷ Makes error-free analysis of asynchronous systems tractable.

Why do not we have such a proof system?

- ▷ Exact implementation details matter a lot.
- ▷ Most current solutions do not preserve high-level description of games.
- ▷ Most of the reduction schemata belong to combinatorial phase.
- ▷ Formal proofs for reachability and independence are tedious.

Help needed!

Questions and answers are welcome!