# Concrete security of cryptographic primitives

Benjamin Grégoire

# Formally verified cryptography

**Algorithms:** **EasyCrypt**

Primitives

Protocols

$\longleftrightarrow$ Adversary

Provable security: $Pr[A \text{ breaks } P] \leq Pr[B(A) \text{ breaks assumption}] + \epsilon$

# Formally verified cryptography

**Algorithms:** **EasyCrypt**

Primitives

Protocols

$\longleftrightarrow$ Adversary

Provable security: $Pr[A \text{ breaks } P] \leq Pr[B(A) \text{ breaks assumption}] + \epsilon$

**Source code:** **Jasmin**

Code

$\longleftrightarrow$ Adversary

Provable security: Algorithms + Functional correctness + Safety

# Formally verified cryptography

**Algorithms:** **EasyCrypt**

Primitives

Protocols

⟷ Adversary

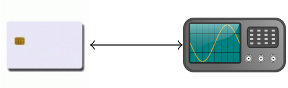
Provable security: $Pr[A \text{ breaks } P] \leq Pr[B(A) \text{ breaks assumption}] + \epsilon$

**Source code:** **Jasmin**

Code ⟷ Adversary

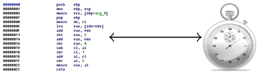
Provable security: Algorithms + Functional correctness + Safety

**Hardware:** **MaskComp** **MaskVerif**



Security: Source + Countermeasure

**Assembly:** **Jasmin**



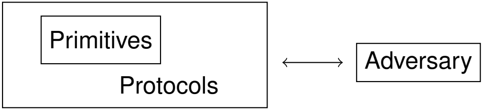
Security: Source + CT + Compiler

# This talk

- Motivate provable security notions
- Proof by reduction
- Probabilistic Relational Hoare Logic (pRHL)

Next talk: Jasmin, functional correctness and Constant time.

**Algorithms:** **EasyCrypt**




Provable security: $Pr[A \text{ breaks } P] \leq Pr[B(A) \text{ breaks assumption}] + \epsilon$

- How to formally define $P$ and $A$ ?
- How to formally define $A$ breaks $P$ ?
- How to formally define $B(A)$ ?
- How to perform proof by reduction ?

# What is provable security?

Precisely define a security model:

- What functionality must the system provide?
- What qualifies as a break for the system?
- What class of attackers should it protect against?

To claim that a system is secure one must:

- State the assumptions upfront:
  - security properties of low-level components (hypothesis)
  - these should be widely used and well studied
- Prove that
  $\forall$ attackers, assumptions $\Rightarrow$ no break
- Or, equivalently,
  $\forall$ attackers, break $\Rightarrow$ assumption false

# Which security model?

All security models are abstractions
They result from a compromise:

- More detail $\rightarrow$ less likely to ignore relevant attacks
- Less detail $\rightarrow$ proofs become feasible

Cryptographers have been developing security models for crypto primitives for a long time

# Elgamal encryption Scheme

Let $G$ be a cyclic group of order $q$ and $g$ a generator of $G$.
Elgamal encryption scheme is defined by

$$\begin{array}{lcl} \mathrm{kg}() & = & \mathrm{sk} \xleftarrow{\$} [0, q); (g^{\mathrm{sk}}, \mathrm{sk}) \\ \mathrm{enc}(\mathrm{pk}, \mathrm{m}) & = & y \xleftarrow{\$} [0, q); (g^{y}, \mathrm{pk}^{y} * \mathrm{m}) \\ \mathrm{dec}(\mathrm{sk}, \mathrm{c}) & = & (\mathrm{gy}, \mathrm{gm}) \leftarrow \mathrm{c}; \mathrm{Some}\ (\mathrm{gm} * \mathrm{gy}^{-\mathrm{sk}}) \end{array}$$

Correctness of the encryption Scheme:
$\forall$ m, (pk, sk) $\leftarrow$ kg(); dec(pk, enc(sk, m)) = Some c

Remarks:

- This is a probabilistic property
- Already an abstraction (plaintext and ciphertext are not bitstring)

# Elgamal encryption Scheme in EasyCrypt

```
type pkey = group.
type skey = F.t.
type ptxt = group.
type ctxt = group * group.

(* Concrete Construction: ElGammal *)
module ElGamal : Scheme = {
 proc kg(): pkey * skey = {
  var sk;

  sk <$ F.dt;
  return (g ^ sk, sk);
 }

 proc enc(pk:pkey, m:ptxt): ctxt = {
  var y;

  y <$ F.dt;
  return (g ^ y, pk ^ y * m);
 }

 proc dec(sk:skey, c:ctxt): ptxt option = {
  var gy, gm;

  (gy, gm) <- c;
  return Some (gm * gy^(-sk));
 }
}.
```

# Correctness of an encryption scheme

```
theory Correctness.

type pkey, skey, ptxt, ctxt.

module type Scheme = {
 proc kg() : pkey * skey
 proc enc(pk : pkey, m : ptxt) : ctxt
 proc dec(sk : skey, c : ctxt) : ptxt option
}.

module Correct(S:Scheme) = {
 proc main(m:ptxt) : bool = {
  var m';
  (pk, sk) ← S.kg();
  c        ← S.enc(pk, m);
  m'       ← S.dec(sk, c);
  return (m' = Some m);
}.

end Correctness.
```

```
clone import Correctness as C with
  type pkey ← pkey, (* i.e. group *)
  type skey ← skey, (* i.e. F.t   *)
  type ptxt ← ptxt, (* i.e. group *)
  type ctxt ← ctxt. (* i.e. group * group *)

lemma Elgamal_correct : hoare [Correct(Elgamal).main : true ⇒  res].
proof. · · · qed.
```

# Hoare Logic

- Judgments $c : P \Rrightarrow Q$ (usually $\{P\}\ c\ \{Q\}$)
  ($P$ and $Q$ are f.o. formulae over program variables)
- A judgment $c : P \Rrightarrow Q$ is valid iff (deterministic setting)

$$\forall m, m \models P \Rightarrow [\![c]\!]_m = m' \Rightarrow m' \models Q$$

- A judgment $c : P \Rrightarrow Q$ is valid iff (probabilistic setting)

$$\forall m, m \models P \Rightarrow [\![c]\!]_m = d \Rightarrow d \models Q$$

  where : $d \models Q$ means

$$\forall m', m' \in d \Rightarrow m' \models Q$$

# Selected rules

$$\frac{}{x \leftarrow e : Q[e/x] \Rrightarrow Q} \qquad \frac{}{x \xleftarrow{\$} d : \forall v \in d, Q[v/x] \Rrightarrow Q}$$

$$\frac{c_1 : P \Rrightarrow Q \quad c_2 : Q \Rrightarrow R}{c_1; c_2 : P \Rrightarrow R}$$

$$\frac{c_1 : P \wedge e \Rrightarrow Q \quad c_2 : P \wedge \neg e \Rrightarrow Q}{\text{if } e \text{ then } c_1 \text{ else } c_2 : P \Rrightarrow Q} \qquad \frac{c : I \wedge e \Rrightarrow I}{\text{while } e \text{ do } c : I \Rrightarrow I \wedge \neg e}$$

$$\frac{c : P \Rrightarrow Q \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{c : P' \Rrightarrow Q'}$$

Going back to the security model

# SM public key encryption security: IND-CPA

$(pk, sk) \xleftarrow{\$} \text{Gen}();$

# SM public key encryption security: IND-CPA

$(pk, sk) \xleftarrow{\$} Gen();$

pk

# SM public key encryption security: IND-CPA

$(m_0, m_1)$

# SM public key encryption security: IND-CPA

$b \xleftarrow{\$} \{0,1\};$

$c \leftarrow \text{Enc}(\text{pk}, m_b);$

# SM public key encryption security: IND-CPA

$b \xleftarrow{\$} \{0,1\};$

$c \leftarrow \mathrm{Enc}(pk, m_b);$

c

# SM public key encryption security: IND-CPA

$b \xleftarrow{\$} \{0,1\}$;

$c \leftarrow \mathrm{Enc}(pk,\ m_b)$;

$b'$

# SM public key encryption security: IND-CPA

$b \xleftarrow{\$} \{0,1\};$

$c \leftarrow \mathrm{Enc}(\mathrm{pk}, m_b);$

$b'$

Break : $b' \stackrel{?}{=} b$

# IND-CPA in EasyCrypt

```
theory Cpa.
type pkey, skey, ptxt, ctxt.

module type Scheme = {
  proc kg() : pkey * skey
  proc enc(pk:pkey, m:ptxt) : ctxt
  proc dec(sk:skey, c:ctxt) : ptxt option
}.

module type Adversary = {
  proc choose(pk:pkey) : ptxt * ptxt
  proc guess(c:ctxt)   : bool
}.

module CPA (S:Scheme) (A:Adversary) = {
  proc main() : bool = {
    var pk, sk, m0, m1, c, b, b';

    (pk, sk)   ← S.kg();
    (m0, m1) ← A.choose(pk);
    b          ← ${0,1};
    c          ← S.enc(pk, b ? m1 : m0);
    b'         ← A.guess(c);
    return (b' = b);
  }
}.
end Cpa.
```

```
clone import Cpa as Cpa0 with
  type pkey ← pkey,
  type skey ← skey,
  type ptxt ← ptxt,
  type ctxt ← ctxt.

lemma Elgamal_cpa &m (A<:Adversary):
 `| Pr[CPA(Elgamal, A).main() @ &m] − 1%r / 2%r | ≤ · · ·
```

# SM public key encryption security: IND-CCA

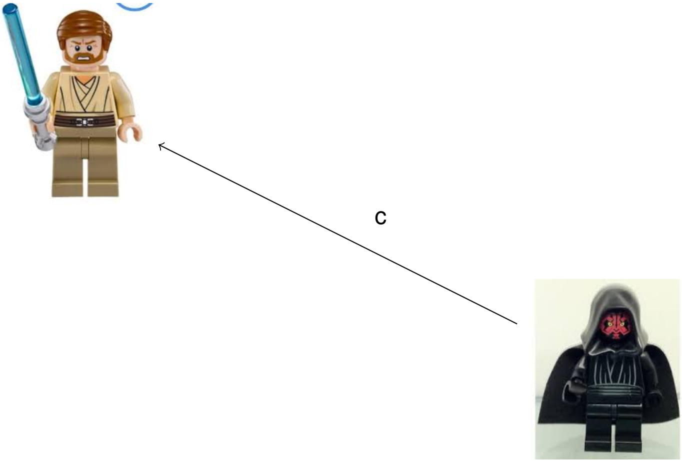
$(pk, sk) \xleftarrow{\$} Gen();$

# SM public key encryption security: IND-CCA

$(pk, sk) \xleftarrow{\$} Gen();$

pk

# SM public key encryption security: IND-CCA



# SM public key encryption security: IND-CCA

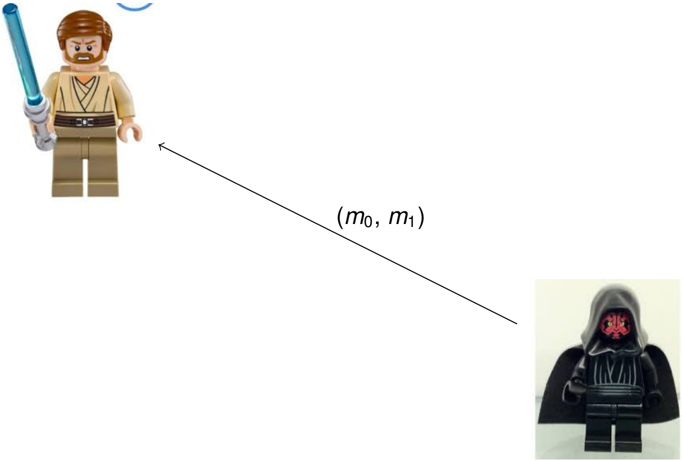
m ← Dec(sk, c)

# SM public key encryption security: IND-CCA

$m \leftarrow \text{Dec}(sk, c)$

m

# SM public key encryption security: IND-CCA



# SM public key encryption security: IND-CCA

$b \xleftarrow{\$} \{0,1\};$

$c^* \leftarrow \mathrm{Enc}(\mathrm{pk}, m_b);$

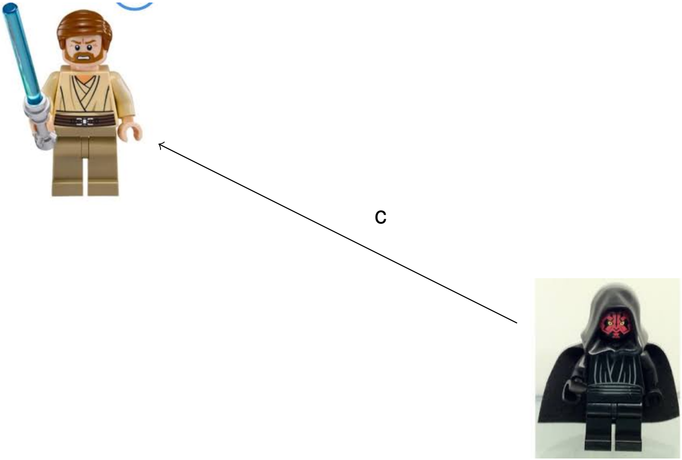
# SM public key encryption security: IND-CCA

$b \xleftarrow{\$} \{0,1\}$;

$c^* \leftarrow \text{Enc}(\text{pk}, m_b)$;

$c^*$

# SM public key encryption security: IND-CCA



# SM public key encryption security: IND-CCA

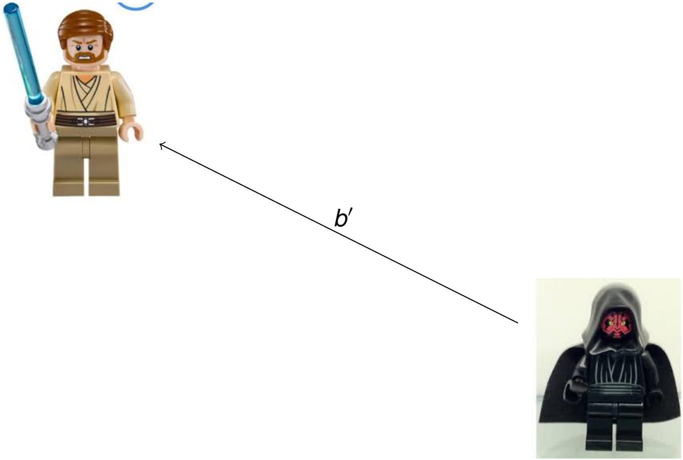
m ← Dec(sk, c)

# SM public key encryption security: IND-CCA

$m \leftarrow \text{Dec}(sk, c)$

m

# SM public key encryption security: IND-CCA



# SM public key encryption security: IND-CCA

$b'$

Break : $b' \stackrel{?}{=} b$

# We need more restrictions

- Can the adversary makes a query to the decryption oracle on $c^*$ ?
- Can the adversary makes queries to the decryption oracle in the “guess” stage ? (CCA1/CCA2)
- Does the number of queries to the decryption oracle is limited/unlimited ? (Can be a problem for exact security)

How to encode this in EasyCrypt?

# IND-CCA in EasyCrypt: first attempt

```
module type CCA_ORC = {
  proc dec(c:ctxt) : ptxt option
}.

module type CCA_ADV (O:CCA_ORC) = {
  proc choose(pk:pkey) : ptxt * ptxt
  proc guess(c:ctxt) : bool
}.
```

```
module CCA (S:Scheme, A:CCA_ADV) = {

  var sk : skey

  module O = {

    proc dec(c:ctxt) : ptxt option = {
      var m;
      m ← S.dec(sk, c);
      return m;
    }
  }

  proc main() : bool = {
    var pk, m0, m1, cstar, b, b';

    (pk, sk)   ← S.kg();
    (m0, m1)   ← A(O).choose(pk);
    b          ←$ {0,1};
    cstar      ← S.enc(pk, b ? m1 : m0);
    b'         ← A(O).guess(cstar);
    return (b' = b);
  }
}.
```

# Problems

- The adversary A can call the decryption oracle on cstar
- The number of calls to the decryption oracle is unlimited

# IND-CCA in EasyCrypt: second attempt

```
const qD : int.

axiom qD_pos : 0 < qD.

module CCA (S:Scheme, A:CCA_ADV) = {
 var log : ctxt list
 var sk : skey

 module O = {
  proc dec(c:ctxt) : ptxt option = {
   var m;
   log ← c :: log;
   m   ← S.dec(sk, c);
   return m;
  }
 }
```

```
 proc main() : bool = {
  var pk, m0, m1, cstar, b, b';
  log        ← [];
  (pk, sk)   ← S.kg();
  (m0, m1)   ← A(O).choose(pk);
  b          ←$ {0,1};
  cstar      ← S.enc(pk, b ? m1 : m0);
  b'         ← A(O).guess(c);
  return (b' = b ∧ ¬ cstar ∈ log ∧ size log ≤ qD);
 }
}.
```

# Problem

```
proc main() : bool = {
  var pk, m0, m1, cstar, b, b';
  log        ← [];
  (pk, sk)   ← S.kg();
  (m0, m1)   ← A(O).choose(pk);
  b          <$ {0,1};
  cstar      ← S.enc(pk, b ? m1 : m0);
  b'         ← A(O).guess(c);
  return (b' = b ∧ ¬ cstar ∈ log ∧ size log ≤ qD);
 }
}.
```

This is not really CCA

- The restriction on the decryption oracle should be only for the *guess* stage

# IND-CCA in EasyCrypt

```
module CCA (S:Scheme, A:CCA_ADV) = {
 var log : ctxt list
 var cstar : ctxt option
 var sk : skey

 module O = {
  proc dec(c:ctxt) : ptxt option = {
   var m : ptxt option;

   if (size log < qD && (Some c ≠ cstar)) {
    log ← c :: log;
    m   ← S.dec(sk, c);
   }
   else m ← None;
   return m;
  }
 }
```

```
 proc main() : bool = {
  var pk, m0, m1, c, b, b';
  log        ← [];
  cstar      ← None;
  (pk, sk)   ← S.kg();
  (m0, m1)   ← A(O).choose(pk);
  b          ←$ {0,1};
  c          ← S.enc(pk, b ? m1 : m0);
  cstar      ← Some c;
  b'         ← A(O).guess(c);
  return (b' = b);
 }
}.
```

# IND-CCA1 versus IND-CCA2

The previous security game corresponds to the IND-CCA2 notion:

- The adversary can call the decryption oracle in both stages *choose* and *guess*

In the IND-CCA1 notion, the adversary can only call the decryption oracle in the *choose* stage

# IND-CCA1 in EasyCrypt

```
module CCA1 (S:Scheme, A:CCA_ADV) = {
 var log : ctxt list
 var sk : skey

 module Oc = {
  proc dec(c:ctxt) : ptxt option = {
   var m : ptxt option;

   if (size log < qD) {
    log ← c :: log;
    m   ← S.dec(sk, c);
   }
   else m ← None;
   return m;
  }
 }

 module Og = {
  proc dec(c:ctxt) : ptxt option = {
    return None
  }
 }
```

```
 proc main() : bool = {
  var pk, m0, m1, cstar, b, b';
  log        ← [];
  (pk, sk)   ← S.kg();
  (m0, m1)   ← A(Oc).choose(pk);
  b          ←$ {0,1};
  cstar      ← S.enc(pk, b ? m1 : m0);
  b'         ← A(Og).guess(cstar);
  return (b' = b);
 }
}.
```

# IND-CCA1 in EasyCrypt

```
module type CCA_ADV (O:CCA_ORC) = {
  proc choose(pk:pkey) : ptxt * ptxt {O.dec}
  proc guess(c:ctxt) : bool          {}
}.

module CCA1 (S:Scheme, A:CCA_ADV) = {
  var log : ctxt list
  var sk : skey

  module O = {
    proc dec(c:ctxt) : ptxt option = {
      var m : ptxt option;

      if (size log < qD) {
        log ← c :: log;
        m   ← S.dec(sk, c);
      }
      else m ← None;
      return m;
    }
  }
```

```
  proc main() : bool = {
    var pk, m0, m1, cstar, b, b';
    log        ← [];
    (pk, sk)   ← S.kg();
    (m0, m1)   ← A(O).choose(pk);
    b          <$ {0,1};
    cstar      ← S.enc(pk, b ? m1 : m0);
    b'         ← A(O).guess(cstar);
    return (b' = b);
  }
}.
```

# Quantification over adversaries

Quantification of adversaries are done by quantification over modules:

```
lemma foo: ∀ (A <: CCA_ADV), ···
```

# Warning: module are not generative

```
module type T = · · ·.

module F(A:T) = {
  var x: int
}

module F1 = F(A1)
module F2 = F(A2)
```

# Warning: module are not generative

```
module type T = ....

module F(A:T) = {
  var x: int
}

module F1 = F(A1)
module F2 = F(A2)
```

In language like ocaml:

- The variable `F.x` does not exists need to instantiate the functor
- Variable `F1.x` and `F2.x` are disjoints

# Warning: module are not generative

```
module type T = · · ·.

module F(A:T) = {
  var x: int
}

module F1 = F(A1)
module F2 = F(A2)
```

In EasyCrypt:

- The variable `F.x` exists
- Variable `F1.x` and `F2.x` are equal to `F.x`

# Need to add more restrictions on adversary

```
module Adv (O:CCA_ORC) = {
 · · ·
 proc guess(c:ctxt) = {
  · · · CCA.sk · · ·
 }
```

- Is a valid adversary (CCA_ADV)
- And it can trivially break the CCA game
- We need to add restrictions

```
 lemma MySchemeCCA: ∀ (A<:CCA_ADV {CCA}),
   Pr[CCA(MyScheme, A)] ≤ · · ·
```

# Where we are?

- We know how to represent schemes
- We know how to represent security notions

We need to understand how to perform proofs.

# A trivial security proof

We want to prove the security of Elgamal encryption scheme:

$$\forall A, |\Pr[CPA(Elgamal, A)] - \frac{1}{2}| = \mathsf{Adv}_{\mathsf{DDH}}(B(A))$$

where

$$\mathsf{Adv}_{\mathsf{DDH}}(D) = |\Pr[DDH_0(D)] - \Pr[DDH_1(D)]|$$

$$DDH_0(D) = x \xleftarrow{\$}; y \xleftarrow{\$}; \text{return } D(g^x, g^y, g^{xy});$$

$$DDH_1(D) = x \xleftarrow{\$}; y \xleftarrow{\$}; z \xleftarrow{\$}; \text{return } D(g^x, g^y, g^z);$$

We reduce the security of Elgamal to the hardness of the decisional Diffie Hellman problem.

# Decisional Diffie Hellman

```
module type DistDDH = {
  proc guess(gx gy gz:group): bool
}.
```

```
module DDH0 (D:DistDDH) = {
  proc main() : bool = {
    var b, x, y;
    x <$ F.dt;
    y <$ F.dt;
    b <- D.guess(g ^ x, g ^ y, g ^ (x*y));
    return b;
  }
}.
```

```
module DDH1 (D:DistDDH) = {
  proc main() : bool = {
    var b, x, y, z;
    x <$ F.dt;
    y <$ F.dt;
    z <$ F.dt;
    b <- D.guess(g ^ x, g ^ y, g ^ z);
    return b;
  }
}.
```

# High level view of the proof

CPA(Elgamal, A)

```
(pk, sk)   ← Elgamal.kg();
(m0, m1)   ← A.choose(pk);
b          ← ${0,1};
c          ← Elgamal.enc(pk, b ? m1 : m0);
b'         ← A.guess(c);
return (b' = b);
```

# High level view of the proof

CPA(Elgamal, A)

```
(* (pk, sk) ← Elgamal.kg(); *)
sk        ←$ F.dt;
pk        ← g^sk;
(m0, m1) ← A.choose(pk);
b         ← ${0,1};
(* c      ← Elgamal.enc(pk, b ? m1 : m0); *)
y         ←$ F.dt;
c         ← (g ^ y, pk ^ y * m);
b'        ← A.guess(c);
return (b' = b);
```

# High level view of the proof

CPA(Elgamal, A)

```
(* (pk, sk) ← Elgamal.kg(); *)
sk       ←$ F.dt;
pk       ← g^sk;
(m0, m1) ← A.choose(pk);
b        ← ${0,1};
(* c     ← Elgamal.enc(pk, b ? m1 : m0); *)
y        ←$ F.dt;
c        ← (g ^ y, pk ^ y * m);
b'       ← A.guess(c);
return (b' = b);
```

```
x        ←$ F.dt;
y        ←$ F.dt;
gx       ← g^x;
gy       ← g^y;
gz       ← g^(x*y);
(m0, m1) ← A.choose(pk);
b        ← ${0,1};
c        ← (gy, gz * m);
b'       ← A.guess(c);
return (b' = b);
```

# High level view of the proof

## CPA(Elgamal, A)

```
(* (pk, sk) ← Elgamal.kg(); *)
sk        ←$ F.dt;
pk        ← g^sk;
(m0, m1)  ← A.choose(pk);
b         ← ${0,1};
(* c      ← Elgamal.enc(pk, b ? m1 : m0); *)
y         ←$ F.dt;
c         ← (g ^ y, pk ^ y * m);
b'        ← A.guess(c);
return (b' = b);
```

## DDH0(B(A))

```
module B(A:Adversary) = {
 proc guess (gx, gy, gz) : bool = {
  var m0, m1, b, b';
  (m0, m1) ← A.choose(gx);
  b     ←$ {0,1};
  b'    ← A.guess(gy, gz * (b?m1:m0));
  return b' = b;
 }
}.

module DDH0 (D:DistDDH) = {
 proc main() : bool = {
  var b, x, y;
  x ←$ F.dt;
  y ←$ F.dt;
  b ← D.guess(g ^ x, g ^ y, g ^ (x*y));
  return b;
 }
}.

DDH0(B(A)).main()
```

# High level view of the proof

CPA(Elgamal, A)

```
(* (pk, sk) ← Elgamal.kg(); *)
sk         ←$ F.dt;
pk         ← g^sk;
(m0, m1)   ← A.choose(pk);
b          ← ${0,1};
(* c       ← Elgamal.enc(pk, b ? m1 : m0); *)
y          ←$ F.dt;
c          ← (g ^ y, pk ^ y * m);
b'         ← A.guess(c);
return (b' = b);
```

```
module B(A:Adversary) = {
  proc guess (gx, gy, gz) : bool = {
    var m0, m1, b, b';
    (m0, m1) ← A.choose(gx);
    b     ←$ {0,1};
    b'    ← A.guess(gy, gz * (b?m1:m0));
    return b' = b;
  }
}.

module DDH0 (D:DistDDH) = {
  proc main() : bool = {
    var b, x, y;
    x ←$ F.dt;
    y ←$ F.dt;
    b ← D.guess(g ^ x, g ^ y, g ^ (x*y));
    return b;
  }
}.

DDH0(B(A)).main()
```

We will prove Pr[CPA(Elgamal, A)] = Pr[DDH0(B(A))]

# High level view of the proof

## DDH1(B(A))

```
module B(A:Adversary) = {
  proc guess (gx, gy, gz) : bool = {
    var m0, m1, b, b';
    (m0, m1) ← A.choose(gx);
    b    ←$ {0,1};
    b'   ← A.guess(gy, gz * (b?m1:m0));
    return b' = b;
  }
}.

module DDH1 (D:DistDDH) = {
  proc main() : bool = {
    var b, x, y;
    x ←$ F.dt;
    y ←$ F.dt;
    z ←$ F.dt;
    b ← D.guess(g ^ x, g ^ y, g ^ z);
    return b;
  }
}.

DDH1(B(A)).main();
```

# High level view of the proof

DDH1(B(A))

```
x        <$- F.dt;
y        <$- F.dt;
z        <$- F.dt;
(m0, m1) <- A.choose(g^x);
b        <$- {0,1};
b'       <- A.guess(g^y, g^z * (b?m1:m0));
return b' = b;
```

# High level view of the proof

DDH1(B(A))

```
x        <$- F.dt;
y        <$- F.dt;
z        <$- F.dt;
(m0, m1) <- A.choose(g^x);
b        <$- {0,1};
b'       <- A.guess(g^y, g^z * (b?m1:m0));
return b' = b;
```

```
x        <$- F.dt;
y        <$- F.dt;
z        <$- F.dt;
(m0, m1) <- A.choose(g^x);
b        <$- {0,1};
b'       <- A.guess(g^y, g^z);
return b' = b;
```

# High level view of the proof

DDH1(B(A))

$$
\begin{array}{ll}
x & \xleftarrow{\$} \text{F.dt;} \\
y & \xleftarrow{\$} \text{F.dt;} \\
z & \xleftarrow{\$} \text{F.dt;} \\
(m_0, m_1) & \leftarrow \text{A.choose(g\^{}x);} \\
b & \xleftarrow{\$} \{0,1\}; \\
b' & \leftarrow \text{A.guess(g\^{}y, g\^{}z} * \text{(b?}m_1\text{:}m_0\text{));} \\
\textbf{return } b' = b;
\end{array}
$$

G

$$
\begin{array}{ll}
x & \xleftarrow{\$} \text{F.dt;} \\
y & \xleftarrow{\$} \text{F.dt;} \\
z & \xleftarrow{\$} \text{F.dt;} \\
(m_0, m_1) & \leftarrow \text{A.choose(g\^{}x);} \\
b' & \leftarrow \text{A.guess(g\^{}y, g\^{}z);} \\
b & \xleftarrow{\$} \{0,1\}; \\
\textbf{return } b' = b;
\end{array}
$$

# High level view of the proof

DDH1(B(A))

```
x         <$ F.dt;
y         <$ F.dt;
z         <$ F.dt;
(m0, m1) <- A.choose(g^x);
b         <$ {0,1};
b'        <- A.guess(g^y, g^z * (b?m1:m0));
return b' = b;
```

G

```
x         <$ F.dt;
y         <$ F.dt;
z         <$ F.dt;
(m0, m1) <- A.choose(g^x);
b'        <- A.guess(g^y, g^z);
b         <$ {0,1};
return b' = b;
```

We will prove:

- Pr[DDH1(B(A))] = Pr[G]
- Pr[G] = $\frac{1}{2}$

# High level view of the proof

1. Pr[CPA(Elgamal, A)] = Pr[DDH0(B(A))]
2. Pr[DDH1(B(A))] = Pr[G]
3. Pr[G] = $\frac{1}{2}$

$$
\begin{aligned}
| \Pr[\mathrm{CPA}(\mathrm{Elgamal}, A)] - \tfrac{1}{2} | &= | \Pr[\mathrm{DDH0}(B(A))] - \tfrac{1}{2} | && (1)\\
&= | \Pr[\mathrm{DDH0}(B(A))] - \Pr[G] | && (3)\\
&= | \Pr[\mathrm{DDH0}(B(A))] - \Pr[\mathrm{DDH1}(B(A))] | && (2)
\end{aligned}
$$

## What we need?

- Being able to compute some probability: $\Pr[G] = \frac{1}{2}$
- Being able to relate probabilities:

$$\Pr[CPA(Elgamal, A)] = \Pr[DDH0(B(A))]$$

- More generally: $\Pr[G_1 : E_1] \leq \Pr[G_2 : E_2]$

# Probabilistic Coupling

Dealing with probability is hard, we want to provide some abstraction

Problem:

$$\Pr[D_1 : E_1] \leq \Pr[D_2 : E_2]$$

where $D_1$, $D_2$ are distributions and $E_1$, $E_2$ are events

Probabilistic coupling allows to relate distributions

# Probabilistic Coupling

Probabilistic Coupling $\mathcal{C}(D_1, D_2, D, R)$:

- $D_1 \in \mathrm{Distr}(U),\ D_2 \in \mathrm{Distr}(V),\ D \in \mathrm{Distr}(U \times V)$
- $R$ is a relation over $U \times V$
- $\pi_1(D) = D_1,\ \pi_2(D) = D_2$
- $\forall (u, v) \in \mathrm{supp}(D), u\ R\ v$

Consequence:

If $\forall u\ v,\ u\ R\ v \Rightarrow u \in E_1 = v \in E_2$ then $\Pr[D_1 : E_1] = \Pr[D_2 : E_2]$

If $\forall u\ v,\ u\ R\ v \Rightarrow u \in E_1 \Rightarrow v \in E_2$ then $\Pr[D_1 : E_1] \leq \Pr[D_2 : E_2]$

# Probabilistic Relational Hoare Logic

$P, \ Q$ probabilistic programs

$$c \sim c' : P \Rrightarrow Q$$

Interpretation:

$$\forall m_1 \ m_2, m_1 \ P \ m_2 \Rightarrow \exists D, \mathcal{C}([\![c]\!]_{m_1}, [\![c']\!]_{m_2}, D, Q)$$

Difficulty: rule for random assignment, desynchronized while, adversaries

# probabilistic Relational Hoare Logic

**lemma** l1 : **equiv** [G1.f ˜ G2.g : $x\{1\} = x\{2\} \Rrightarrow$ **res**$\{1\}$ = **res**$\{2\} \wedge$ G2.z$\{1\} = 0$].
**proof**. · · · **qed**.

**lemma** l2 : **equiv** [G1.f ˜ G2.g : $=\{x\} \Rrightarrow$ $=\{$**res**$\}$ $\wedge$ G2.z$\{1\} = 0$].

**equiv** l3 : G1.f ˜ G2.g : $=\{x\} \Rrightarrow$ $=\{$**res**$\}$ $\wedge$ G2.z$\{1\} = 0$.

# *equiv* judgment can be used to deduce fact on probabilities

$$\frac{G_1 \sim G_2 : \text{true} \Rrightarrow Q \qquad Q \Rightarrow E_{\{1\}} = F_{\{2\}}}{\Pr[G_1 : E] = \Pr[G_2 : F]}$$

$$\frac{G_1 \sim G_2 : \text{true} \Rrightarrow Q \qquad Q \Rightarrow E_{\{1\}} \Rightarrow F_{\{2\}}}{\Pr[G_1 : E] \leq \Pr[G_2 : F]}$$

```
lemma pr &m vx: Pr[G1.f(vx) @ &m : res] =  Pr[G2.g(vx) @ &m : res ∧ G2.z = 0].
proof.
 byequiv.
 . . .
qed.
```

# Proof rules: skip and assignments

**Skip**

$$\frac{P \Rrightarrow Q}{\text{skip} \sim \text{skip} : P \Rrightarrow Q}\ \textbf{skip}$$

**Sequence**

$$\frac{c_1 \sim c_2 : P \Rrightarrow R \qquad c_1' \sim c_2' : R \Rrightarrow Q}{c_1; c_1' \sim c_2; c_2' : P \Rrightarrow Q}\ \textbf{seq}$$

**Assignments**

$$\frac{}{x \leftarrow e \sim \text{skip} : Q[_{\{1\}}/x_{\{1\}}] \Rrightarrow Q}\ \textbf{wp}$$

$$\frac{}{x \leftarrow e \sim x' \leftarrow e' : Q[e_{\{1\}}/x_{\{1\}}][e'_{\{2\}}/x'_{\{2\}}] \Rrightarrow Q}\ \textbf{wp}$$

# Proof rules: conditionals

## Conditionals

$$\frac{P \Rightarrow e_{\{1\}} = e'_{\{2\}} \quad c_1 \sim c'_1 : P \wedge e_{\{1\}} \Rrightarrow Q \qquad c_2 \sim c'_2 : P \wedge \neg e_{\{1\}} \Rrightarrow Q}{\text{if } e \text{ then } c_1 \text{ else } c_2 \sim \text{if } e' \text{ then } c'_1 \text{ else } c'_2 : P \Rrightarrow Q} \; \textbf{if}$$

$$\frac{c_1 \sim c : P \wedge e_{\{1\}} \Rrightarrow Q \qquad c_2 \sim c : P \wedge \neg e_{\{1\}} \Rrightarrow Q}{\text{if } e \text{ then } c_1 \text{ else } c_2 \sim c : P \Rrightarrow Q} \; \textbf{if}\{1\}$$

## Case

$$\frac{c \sim c' : P \wedge R \Rrightarrow Q \quad c \sim c' : P \wedge \neg R \Rrightarrow Q}{c \sim c' : P \Rrightarrow Q} \; \textbf{case}\ \mathrm{R}$$

## Reduce Conditionals

$$\frac{c : P \Rrightarrow e \quad c; c_1 \sim c' : P \wedge R \Rrightarrow Q}{c;\ \text{if } e \text{ then } c_1 \text{ else } c_2 \sim c' : P \Rrightarrow Q} \; \textbf{rcondt}$$

Rules for conditionals are a consequence of the **Case** and **Reduce**

# Loops

**Two-sided rule**

$$\frac{\begin{array}{c} I \Rightarrow e_{\{1\}} = e'_{\{2\}} \\ c \sim c' : I \wedge e_{\{1\}} \Rightarrow I \end{array}}{\text{while } e \text{ do } c \sim \text{while } e' \text{ do } c' : I \Rightarrow I \wedge \neg e_{\{1\}}} \ \textbf{while}: \text{I}$$

- rule is incomplete: same number of iterations

**One sided-rules**

- standard rule with losslessness verification condition

# Proof rules: program transformations

EasyCryptprovides rules for program transformations:

- **inline** f : inline the function f
- **inline** * : inline all functions
- **swap**{1} i n : move instruction at position i of p instructions

# Proof rules: random assignment

## Intuition

Let $A$ be a finite set and let $f, g : A \rightarrow B$. Define

- $c = x \xleftarrow{\$} \mu; y \leftarrow f\ x$
- $c' = x \xleftarrow{\$} \mu'; y \leftarrow g\ x$

Then $[\![c]\!] = [\![c']\!]$ (extensionally) iff there exists $h : A \stackrel{1-1}{\rightarrow} A$ st

- $f = g \circ h$
- for all $a$, $\mu(a) = \mu'(h(a))$

$$\frac{h \text{ is 1-1 and } \forall a,\ \mu(a) = \mu'(h(a))}{x \xleftarrow{\$} \mu \sim x \xleftarrow{\$} \mu' \ :\ \forall v, Q[h\ v/x_{\{1\}}][v/x_{\{2\}}] \Rightarrow Q}$$

- Rule captures a special case of lifting
- General rule might lead to untractable arithmetic equalities

# Adversaries: Intuition

- Adversaries can be any sequence of code.
- Given the same inputs, provide the same outputs

$$\frac{}{x \leftarrow A(\vec{y}) \ \sim \ x \leftarrow A(\vec{y}) \ : \ =_{\{\vec{y}\}} \ \Rightarrow \ =_{\{x\}}}$$

But adversaries can also perform oracle calls . . .

# Adversaries with oracle

- Adversaries perform arbitrary sequences of oracle calls (and intermediate computations)
- Oracle are not necessary the same in both sides
- We can view it as a loop

$$\frac{z \leftarrow O(\vec{w}) \;\sim\; z \leftarrow O'(\vec{w}) \;:\; I \wedge =_{\{\vec{w}\}} \;\Rrightarrow\; I \wedge =_{\{z\}}}{x \leftarrow A^{O}(\vec{y}) \;\sim\; x \leftarrow A^{O'}(\vec{y}) \;:\; I \wedge =_{\{\vec{y}\}} \;\Rrightarrow\; I \wedge =_{\{x\}}}$$

Restriction:

- Intermediate computations should not break $I$
- global variables of the adversary should be equals

# Reasoning about Failure Events

Lemma (Fundamental Lemma)

*Let* $A, B, \mathsf{bad}$ *be events and* $G_1, G_2$ *be two games such that*

$$\Pr[G_1 : A \wedge \neg\mathsf{bad}] = \Pr[G_2 : B \wedge \neg\mathsf{bad}]$$

*and*

$$\Pr[G_1 : \mathsf{bad}] = \Pr[G_2 : \mathsf{bad}]$$

*Then*

$$|\Pr[G_1 : A] - \Pr[G_2 : B]| \leq \Pr[G_2 : \mathsf{bad}]$$

# Fundamental Lemma in pRHL

Recall that to prove $\Pr[G_1 : E] = \Pr[G_2 : F]$ it is sufficient to have

$$G_1 \sim G_2 : \text{true} \Rrightarrow Q \text{ and } Q \Rightarrow E_{\{1\}} = F_{\{2\}}$$

Let $A, B, \text{bad}$ be events and $G_1, G_2$ be two games such that

$$G_1 \sim G_2 : \text{true} \Rrightarrow (\text{bad}_{\{1\}} \Leftrightarrow \text{bad}_{\{2\}}) \wedge (\neg\text{bad}_{\{2\}} \Rightarrow (A_{\{1\}} \Leftrightarrow B_{\{2\}}))$$

then

$$\begin{array}{lcl} \Pr[G_1 : A \wedge \neg\text{bad}] & = & \Pr[G_2 : B \wedge \neg\text{bad}] \\ \Pr[G_1 : \text{bad}] & = & \Pr[G_2 : \text{bad}] \end{array}$$

So we can apply the Fundamental Lemma and get:

$$|\Pr[G_1 : A] - \Pr[G_2 : B]| \leq \Pr[G_2 : \text{bad}]$$

# Simpler variant

Let $A$, $B$, bad be events and $G_1$, $G_2$ be two games such that

$$G_1 \sim G_2 : \text{true} \Rrightarrow \neg\text{bad}_{\{2\}} \Rightarrow A_{\{1\}} \Rightarrow B_{\{2\}}$$

Then

$$\Pr[G_1 : A] \leq \Pr[G_2 : B] + \Pr[G_2 : \text{bad}]$$

Proof:
Recall that to prove $\Pr[G_1 : E] \leq \Pr[G_2 : F]$ it is sufficient to have

$$G_1 \sim G_2 : \text{true} \Rrightarrow Q \text{ and } Q \Rightarrow E_{\{1\}} \Rightarrow F_{\{2\}}$$

Since

$$(\neg\text{bad}_{\{2\}} \Rightarrow A_{\{1\}} \Rightarrow B_{\{2\}}) \Rightarrow A_{\{1\}} \Rightarrow B_{\{2\}} \vee \text{bad}_{\{2\}}$$

we have

$$\begin{aligned} \Pr[G_1 : A_{\{1\}}] &\leq \Pr[G_2 : B_{\{1\}} \vee \text{bad}_{\{2\}}] \\ &\leq \Pr[G_2 : B_{\{1\}}] + \Pr[G_2 : \text{bad}_{\{2\}}] \end{aligned}$$

# Fundamental lemma: adversary rule

Assume that:

- bad is monotonic

$$O : \text{bad} \Rrightarrow \text{bad} \qquad\qquad O' : \text{bad} \Rrightarrow \text{bad}$$

- Oracle calls preserve equivalence up to failure

$$\begin{array}{l} y \leftarrow O(x) \ \sim \ y \leftarrow O'(x) \ : \\ \quad \neg\text{bad}_{\{1\}} \wedge \neg\text{bad}_{\{1\}} \wedge Q \wedge =_{\{x\}} \Rrightarrow \\ \quad \text{bad}_{\{1\}} = \text{bad}_{\{2\}} \wedge (\neg\text{bad}_{\{2\}} \Rightarrow Q \wedge =_{\{y\}}) \end{array}$$

Then adversary preserves equivalence up to failure

$$\begin{array}{l} y \leftarrow A^{O}(x) \ \sim \ y \leftarrow A^{O'}(x) \ : \\ \quad \neg\text{bad}_{\{1\}} \wedge \neg\text{bad}_{\{1\}} \wedge Q \wedge =_{\{x\}} \Rrightarrow \\ \quad \text{bad}_{\{1\}} = \text{bad}_{\{2\}} \wedge (\neg\text{bad}_{\{2\}} \Rightarrow Q \wedge =_{\{y\}}) \end{array}$$

# Conclusion

- Solid foundation for cryptographic proofs
- Cryptographic hypothesis and security properties can be expressed using games (programs)
- probabilistic Relational Hoare Logic allows to capture most of the steps used in cryptographic proof:
  - reduction
  - failure event
  - bridging step / program transformation

`http://www.easycrypt.info`