CS-550

EzRSA

Using formal verification to prove RSA’s hardness

Mathias Marty, Artur Papp, Zacharie Tevaearai

Background

Necessity of complexity

- Shannon defined perfect secrecy in 1949
- Modern cryptography assumes adversaries with limited computational resources
- Probabilistic guarantees (upper bound of the probability of an adversary breaking the system)
- Proofs are typically done by reduction to known “hard” problems

RSA (quick overview)

1. We generate 2 primes, p and q and fix $n = p^* q$ (p and q are secret)
2. We fix e such that e is coprime with $(p-1)*(q-1) = \varphi(n)$, e is the part of the public key
3. We fix $d = e^{-1} \bmod (\varphi(n))$, d is the secret key
4. To encrypt, we take $x^e = y \bmod n$, and send y
5. To decrypt, the recipient takes $y^d \bmod n = x$

Game and *Adversary*

- In 1984, Goldwasser and Micali used the term *game* and *adversary* in a cryptographic context
- The game models a cryptographic primitive or computational problem, which is considered hard/secure if an adversary with polynomial computational power has only a negligible chance of winning the game
- EasyCrypt uses a game-based approach to constructing proofs

RSAFP
(RSA Factoring Problem)

1. RSA generate -> (n, e, d)
2. Adv(n) -> (p, q)
3. Win if: $p*q = n$, $1 < p,q < n$

It is believed to be hard!

RSAGOP (RSA Group Ordering Problem)

1. RSA generate -> (n, e, d)
2. Adv(n) -> (z)
3. Win if: z = φ(n)

RSAEMP (RSA Exponent Multiple Problem)

1. RSA generate -> (n, e, d)
2. Adv(n) -> (z)
3. Win if: z divides λ(n) and z not 0

RSAKRP (RSA Key Recovery Problem)

1. RSA generate -> (n, e, d)
2. Adv(n, e) -> (z)
3. Win if: z = d

RSADP (RSA Decryption Problem)

1. RSA generate -> (n, e, d)
2. Pick x in Z_n
3. $y = x^e \bmod n$
4. Adv(n, e, y) -> (z)
5. Win if: z = x

Reductions

$$
\begin{tikzcd}
\mathrm{RSAFP} & \mathrm{EMP} \arrow[l, Rightarrow] \\
\mathrm{GOP} \arrow[u, Leftrightarrow] \arrow[r, Rightarrow] & \mathrm{RSAKRP} \arrow[u, Rightarrow] \arrow[r, Rightarrow] & \mathrm{RSADP}
\end{tikzcd}
$$

Game as program

- These games can be described mathematically
- In 2006, Bellare and Rogaway modelled them as probabilistic programs
- This opened the path for formal verification on these games
- EasyCrypt facilitates this transition by using already existent SMT solvers and automated theorem provers

RSA in EasyCrypt

```
(** RSA Types **)
type pkey = int * int.
type skey = int * int * int.
type plaintext = int.
type ciphertext = int.
```

```
module RSA: Scheme = {
  proc key_gen(): pkey * skey = {...}
  proc enc(pk: pkey, m: plaintext): ciphertext = {...}
  proc dec(sk: skey, c: ciphertext): plaintext = {...}
}
```

How to generate keys ?

```
proc key_gen(): pkey * skey = {
  var pk: pkey;
  var sk: skey;

  (pk, sk) <$ (* where do we sample ?? *);
}
```

Do it more abstractly

```
type pkey.
op p_n: pkey -> int.
op p_e: pkey -> int.

type skey.
op s_d: skey -> int.
op s_p: skey -> int.
op s_q: skey -> int.
op s_n: skey -> int = fun sk,
    s_p sk * s_q sk.
```

Abstract distributions

```
type keypair = pkey * skey.
op keypairs: { keypair distr | is_lossless keypairs }
    as dkp_ll.

axiom valid_keypairs pk sk:
support keypairs (pk,sk) =>
  p_n pk = s_n sk /\
  (p_e pk)*(s_d sk) %% (s_p sk-1)*(s_q sk-1) = 1 /\
  2^(k - 1) <= p_n pk < 2^k.
```

```
axiom primality_p sk:
prime (s_p sk).

axiom primality_q sk:
prime (s_q sk).
```

How to define RSA.Gen() ?

```
const SK: skey.
const PK: pkey.
axiom valid_global : support keypairs (PK, SK).
```

Game as program (formally)

$$
\begin{array}{rll}
\mathcal{C} ::= & \mathsf{skip} & \text{nop} \\
\mid & \mathcal{V} \leftarrow \mathcal{E} & \text{deterministic assignment} \\
\mid & \mathcal{V} \xleftarrow{\$} \mathcal{D}\mathcal{E} & \text{probabilistic assignment} \\
\mid & \mathsf{if}\ \mathcal{E}\ \mathsf{then}\ \mathcal{C}\ \mathsf{else}\ \mathcal{C} & \text{conditional} \\
\mid & \mathsf{while}\ \mathcal{E}\ \mathsf{do}\ \mathcal{C} & \text{loop} \\
\mid & \mathcal{V} \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E}) & \text{procedure call} \\
\mid & \mathcal{C};\ \mathcal{C} & \text{sequence}
\end{array}
$$

In practice

```
module D4 = {
  var k : bool;
  proc sample () : int = {
    var r : int;
    if (k = true) {
       return 1;
    }

    r <$ [1..4];
    return r;
  }
}.
```

Encoding our games and adversaries 1/3

```
module type RSAFP_adv = {
    proc factorize(n: int): int * int
}.

module RSAFP_game(Adv: RSAFP_adv) = {
    proc main() = {
        var p': int;
        var q': int;

        (p', q') <@ Adv.factorize(p_n PK);

        return ((p'*q') = p_n PK) && 1 < p'
            && p' < q' && q' < p_n PK;
    }
}.
```

Encoding our games and adversaries 2/3

```
module RSAGOP_game(Adv: RSAGOP_adv) = {

 proc main() = {
  var z: int;

  z <@ Adv.compute_GO(p_n PK);

  return z = (s_p SK - 1)*(s_q SK - 1);
 }
}.
```

Encoding a *reduction*

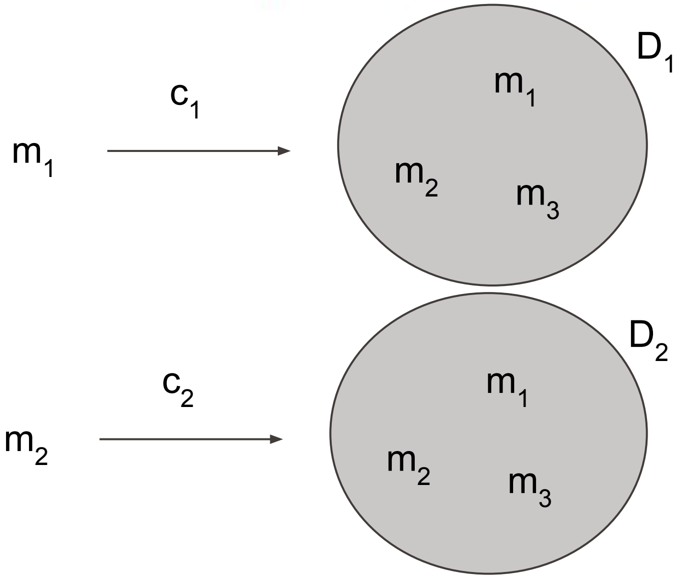
```
module RSAGOP_using_RSAFP(A: RSAFP_adv): RSAGOP_adv = {
  proc compute_GO(n: int): int = {
  var p: int;
  var q: int;
    (p, q) <@ A.factorize(n);

      return (p-1)*(q-1);
  }
}.
```

Encoding a *goal*

```
section.
declare module A <: RSAFP_adv.
  (* TODO prove it*)
lemma RSAFP_to_RSAGOP_red :
    equiv[
        RSAFP_game(A).main ~ RSAGOP_game(RSAGOP_using_RSAFP(A)).main :
        true ==> res{1} => res{2}
        ].
    admit.
  qed.
 end section.
```

pRHL judgment

$$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$$



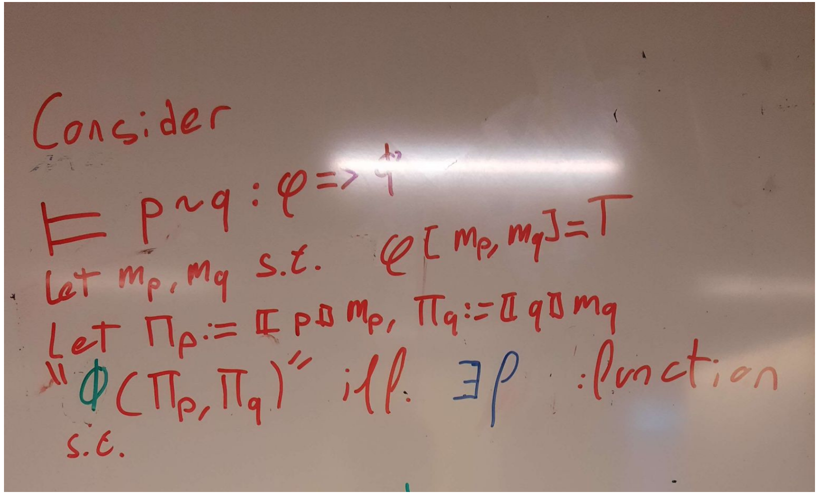
??? $\phi[D_1, D_2]$???

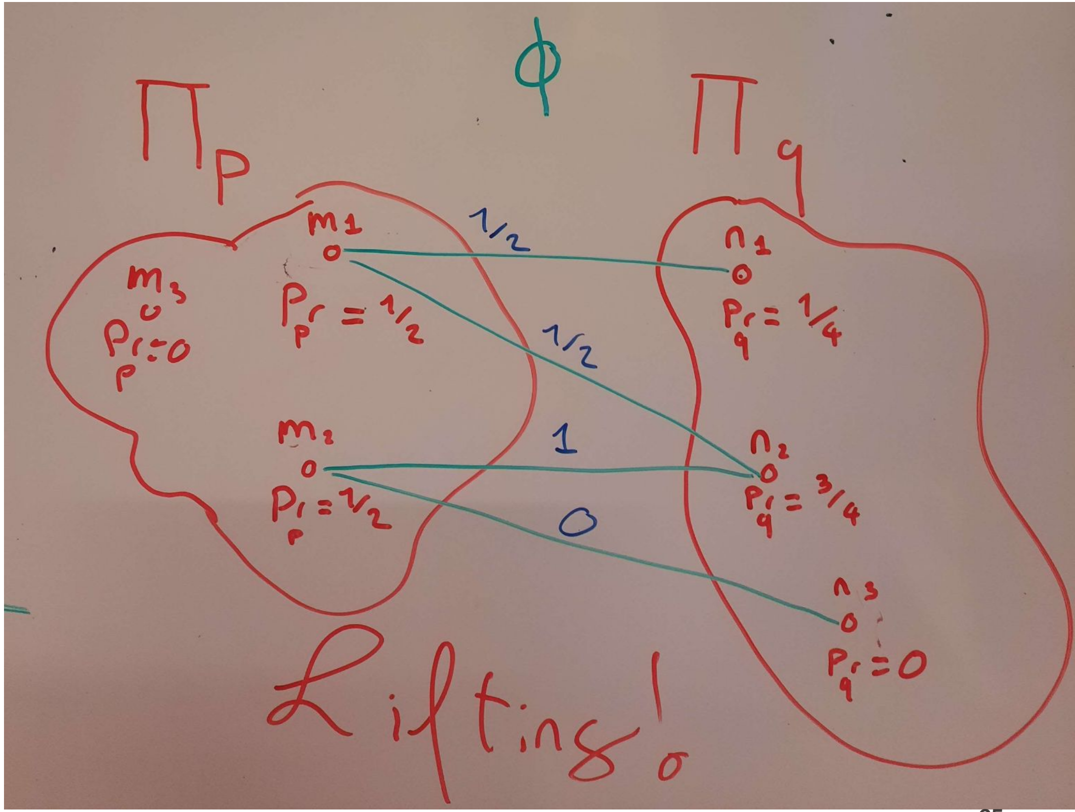
For any memories m_1 and m_2, if $\psi[m_1, m_2]$ then : $\phi[D_1, D_2]$ where…

EPFL

Lifting operator

??? $\phi[D_1, D_2]$???





Example of *pRHL judgment*

$\models c_1 \sim c_2 : \Psi \Rightarrow \Phi$ where Ψ has the form (E_1<1> => E_2<2>) we have :

For any memories m_1 and m_2, if $\psi[m_1, m_2]$ then : $Pr[c_1, m_1: E_1] \leq Pr[c_2, m_2: E_2]$

c_1 c_2 Ψ

```
equiv[
    RSAFP_game(A).main ~ RSAGOP_game(RSAGOP_using_RSAFP(A)).main : true
    ==> res{1} => res{2}
].
```

$\overbrace{E_1 \qquad E_2}^{\Phi}$

=> For any memories m_1 and m_2, - : $Pr[c_1, m_1: E_1] \leq Pr[c_2, m_2: B_2]$

=> For any memories m_1 and m_2: $Pr[c_1(m_1) = 1] \leq Pr[c_2(m_2) = 1]$

=> *If there exists an adversary for Factorization that wins with high probability, then we can solve GOP with at least the same probability*

Proofs (ex. RSAFP => RSAGOP)

```
pre = true

    RSAFP_game(A).main ~ RSAGOP_game(RSAGOP_using_RSAFP(A)).main

post = res{1} => res{2}
```

```
&1 (left ) : {p', q' : int}
&2 (right) : {z : int}

pre = true

(p', q') <@                  (1)  z <@
  A.factorize(               ( )    RSAGOP_using_RSAFP(A).compute_GO(
  p_n PK)                    ( )    p_n PK)

post =
  p'{1} * q'{1} = p_n PK && 1 < p'{1} && p'{1} < q'{1} && q'{1} < p_n PK =>
  z{2} = (s_p SK - 1) * (s_q SK - 1)
```

proc.

```
&1 (left ) : {p', q' : int}
&2 (right) : {z, n, p, q : int}

pre = true

(p', q') <@                    (1)  n <-  p_n PK
  A.factorize(p_n PK)          ( )
                               ( )
                               (2)  (p, q) <@  A.factorize(n)

post =
  p'{1} * q'{1} = p_n PK && 1 < p'{1} && p'{1} < q'{1} && q'{1} < p_n PK =>
  (p{2} - 1) * (q{2} - 1) = (s_p SK - 1) * (s_q SK - 1)
```

proc; inline *; auto.

Proofs (what goes wrong)

```
&1 (left ) : {p', q' : int}
&2 (right) : {z, n, p, q : int}

pre = true

(p', q') <@                    (1)  n <-
  A.factorize(                 ( )    p_n PK
  p_n PK)                      ( )
                               (2)  (p, q) <@
                               ( )    A.factorize(n)

post = p'{1} = p{2} /\ q'{1} = q{2} <- This fails for some reasons…
```

Conclusion

- We understand the basics of EasyCrypt
- We could encode our problems
- None of the proofs work !