guardtime

EasyCrypt for the working cryptographers

Denis Firsov

EasyCrypt

- Toolset for reasoning about probabilistic computations with adversarial code.
- The main application is the construction and verification of cryptographic proofs (especially game-based).

Basics

- Total functional language with inductive datatypes:

```
op id ['a] : 'a -> 'a = λ x. x.
```

- Ambient higher-order classical logic:

```
lemma id_prop ['a] : forall (x : 'a), id x = x.
proof. trivial. qed.
```

Distributions

- Every type is associated with the type of discrete (sub-)distributions of its elements.

```
type x = int distr.
```

- A discrete distribution is fully defined by its mass function. i.e. by a non-negative function `f :: t -> real` so that $\Sigma_x f(x) \leq 1$.

Modules

- In EasyCrypt, cryptographic protocols are modeled as modules, which consists of global variables and procedures.
- Modules may be parameterized by other modules (for example, adversaries, oracles, etc.).
- Procedures are written in a simple imperative language, with while loops and random sampling.

Example: Guessing game

- The module `GuessingGame` has three global variables: c and q of type `int`, and `win` of type bool.
- For any initial memory **m** the state of the module is a tuple:

  ```
  glob GuessingGame
     = int * int * bool.
  ```

- The player has at most q attempts (set by initialization procedure).
- The player wins if they guess correctly at least once.

```
module GuessingGame = {
  var c q : int
  var win : bool

  proc init(x : int) = {
    (c, win, q) <- (0, false, x);
  }

  proc guess(x : bits) : bool = {
    var r;
    if (c < q) {
      r <$ bD;
      win <- win || r = x;
      c <- c + 1;
    }
    return win;
  }
}.
```

guardtime

Module types

- In EasyCrypt **module types** specify the types of a set of module procedures (similar to interfaces in Java).

- We can specify the module type of `GuessingGame` as follows:

```
module type GuessingGame = {
  proc init(x : int) : unit
  proc guess(x : bits) : bool
}.
```

Module types

- We can define a module type of protocol parties (adversaries/players), who receive an instance of a guessing game as a module parameter.

- An adversary must have a `play()` procedure which starts the game:

```
module type Adversary(G : GuessingGame) = {
        proc play() : unit {G.guess}
}.
```

- To forbid adversaries to reinitialize the game the `play()` procedure can only execute the `guess()` procedure of the parameter game.

Probability expressions

EasyCrypt has Pr-constructs which can be used to refer to the probabilities of events in program executions:

$$\texttt{Pr}[\texttt{r} \leftarrow \texttt{X.p() @ } \mathbf{m} \texttt{ : M r}]$$

denotes the probability that the return value r of procedure p of module X given initial memory m satisfies the predicate M.

Probability expressions: Example

We can express the probability of adversary `A` winning the guessing-game with q tries `(G := GuessingGame)`:

```
Pr[G.init(q); A(G).play() @ m: G.win].
```

Program logics

- Ordinary **Hoare logic:**
  ```
  hoare [ M.p : P ⇒ Q ].
  ```

- **Probabilistic Hoare logic** for proving probabilistic facts about single games:
  ```
  phoare [ M.p : P ⇒ Q ] = real.
  ```

- **Probabilistic Relational Hoare Logic** for proving relations between pairs of games:
  ```
  equiv [ M.p ~ W.b : P ⇒ Q ].
  ```

Program logics

- `equiv [ b <$ {0,1} ~ q <$ {0,1} : true ⇒ b = q ].`
- `equiv [ b <$ {0,1} ~ q <$ {0,1} : true ⇒ b ≠ q ].`

Probability expressions: Example

Using the program logics we can try to prove the upper bound on the winning event (let `G := GuessingGame`):

```
lemma winPr : ∀ (A : Adversary) m q, 0 ≤ q
   Pr[G.init(q); A(G).play() @ m: G.win]
         ≤ q / support_size bD.
```

Do you think it is provable?

Probability expressions: Example

What if G is also adversary? Hence, we must exclude G from the set of adversaries.

```
lemma winPr : ∀ (A : Adversary{-G}) m q, 0 ≤ q
    Pr[G.init(q); A(G).play() @ m: G.win]
          ≤ q / support_size bD.
```

¡Proof Flash!

```
proof. move => A. move => q q_pos.

have ->:  Pr[ Main(GG,A).main(q) @ &m : GG.win ] = Pr[
Main(GG,A).main(q) @ &m : GG.win  /\ (0 <= GG.c <= q) ].

byequiv (_: ={glob A, glob GG, arg} /\ GG.q{1} = GG.q{2}
/\ arg{1} = q  ==> _). proc.

seq 1 1 : (={glob A, glob GG} /\ GG.q{1} = GG.q{2} /\ (0
<= GG.c <= GG.q){1} /\ GG.q{1} = q).

inline *.   wp. skip. progress.

 call (_: (0 <= GG.c <= GG.q){1} /\ ={glob GG} /\
GG.q{1} = q).

proc. sp. if. smt.  wp. rnd. skip. smt. skip. smt.

skip. progress. auto.  auto.

  fel 1 GG.c (fun x => 1%r / (supp_size bD)%r) q GG.win
[GG.guess : (GG.c < GG.q)] => //.

   rewrite BRA.sumr_const RField.intmulr count_predT.

        smt (size_range).

   inline *;auto.

proc;inline *;sp 1;if;last by hoare.

        wp.

        conseq (_ : _ ==> r = x)=> [ /# |
].

        rnd;auto => &hr /> ??? .

        move => z.

        rewrite mu1_uni_ll. apply bDU.
apply bDL.

  smt.

   move=> c;proc;sp;inline *.

        by rcondt 1 => //;wp;conseq (_: _
==> true) => // /#.

  move=> b c;proc;sp;inline *;if => //.

  sp. wp. rnd.  skip.  smt.

qed.
```

Example: Collision resistance

- Define set of collision resistance adversaries.
- Define collision resistance game (aka experiment) played by an adversary.
- We say that “h” is collision-resistant iff

  ∀ m A,Pr[CR(A).main(h) @ m : res] is small.

- Is CR preserved under self-composition?

```
module type Adv = {
  proc adv(g : D → D) : D * D
}.

module CR(A : Adv) = {

  proc main(h : D → D) : bool = {
    var x, x' : D;

    (x, x') <@ A.adv(h);

    return h x = h x'
                ∧ x ≠ x';
  }
}.
```

Example: Proof by reduction

- Assume there is an adversary A who breaks h ∘ h.
- Implement transformation B which can use A to break CR of h.
- If we succeed then we arrive at contradiction with assumption that h is CR.
- Conclude that h ∘ h is CR.

```
module B(A : Adv) = {
  proc adv(h : D → D) : D * D = {
    var x,x',r,r' : D;

    (x, x') <@ A.adv(h∘h);

    if ((h x) = (h x')) {
      r  ← x;
      r' ← x';
    } else {
      r  ← h x;
      r' ← h x';
    }
    return (r,r');
  }
}.
```

guardtime

Lemma and proof

```
lemma cr_preservation : ∀ (A : Adv) m,
  Pr[CR(A).main(h ∘ h) @ m : res]
       ≤ Pr[CR(B(A)).main(h) @ m : res].
proof.
  progress.
  byequiv => //.     (* KEY: using pRHL *)
  proc.
  inline*. wp.
  call (_:true).
  wp.
  skip.
  progress.
qed.
```

Up to here...

- We used probabilistic Hoare logic to derive an exact bound:

```
lemma winPr : ∀ (A : Adversary{-G}) m q, 0 ≤ q =>
   Pr[G.init(q); A(G).play() @ m: G.win]
     ≤ q / support_size bD.
```

- We used probabilistic relational Hoare logic to develop a proof by reduction:

```
lemma cr_preservation : ∀ (A : Adv) m,
  Pr[CR(A).main(h ∘ h) @ m : res]
    ≤ Pr[CR(B(A)).main(h) @ m : res].
```

- What about conceptually more complicated proofs?

More complex arguments?

$$\Pr\left[\begin{array}{c} A.init();\ s \leftarrow A.getState(); \\ r_1 \leftarrow A.main();\ A.setState(s); \\ r_2 \leftarrow A.main() \,@\, \boldsymbol{m} : r_1 \wedge r_2 \end{array}\right] \geq \Pr\left[\,A.init();\ r \leftarrow A.main() \,@\, \boldsymbol{m} : r\,\right]^2 .$$

More complex proofs?

- Step (1) applies "the averaging technique" by representing A.init() as a family of distributions D.
- Step (2) applies multiplication rule to two independent runs.
- Step (3) is an application of Jensen's inequality.
- Step (4) undoes the averaging.

$$\Pr\left[\begin{array}{c} A.init();\ s \leftarrow A.getState(); \\ r_1 \leftarrow A.main();\ A.setState(s); \\ r_2 \leftarrow A.main() \ @\ \mathbf{m} : r_1 \wedge r_2 \end{array}\right]$$

$$\stackrel{(1)}{=} \sum_{\mathbf{n}} \mu_1(D_A^{\mathbf{m}}, \mathbf{n}) \cdot \Pr\left[\begin{array}{c} s \leftarrow A.getState(); \\ r_1 \leftarrow A.main(); A.setState(s); \\ r_2 \leftarrow A.main() \ @\ \mathbf{n} : r_1 \wedge r_2 \end{array}\right]$$

$$\stackrel{(2)}{=} \sum_{\mathbf{n}} \mu_1(D_A^{\mathbf{m}}, \mathbf{n}) \cdot \Pr\left[r \leftarrow A.main() \ @\ \mathbf{n} : r\right]^2$$

$$\stackrel{(3)}{\geq} \left(\sum_{\mathbf{n}} \mu_1(D_A^{\mathbf{m}}, \mathbf{n}) \cdot \Pr\left[r \leftarrow A.main() \ @\ \mathbf{n} : r\right]\right)^2$$

$$\stackrel{(4)}{=} \Pr\left[\, A.init();\ r \leftarrow A.main() \ @\ \mathbf{m} : r \,\right]^2 .$$

More complex proofs?

- Problem: the built-in program logics/tactics can handle basic proof patterns, but (usually) will not work if you need more complex mathematical results.
- The main challenge is absence of reflection of programs into their denotation.
- Ideally we want the following theorem (inside the EasyCrypt):

Theorem 1.2. *For all memories* $\boldsymbol{m}$ *and programs* A *there exists a family of distributions* D_A^g *(with* g *of type* $\mathcal{G}_A$*) such that for all predicates* M *on values of type* $\mathcal{G}_A$*:*

$$\Pr\left[A.main() @ \boldsymbol{m} : M(\mathcal{G}_A^{\boldsymbol{fin}})\right] = \mu(D_A^{\mathcal{G}_A^{\boldsymbol{m}}}, M).$$

Probabilistic reflection

- Probabilistic reflection of modules

```
lemma prob_reflection : ∃ D, ∀ m M i,
    Pr[ r ← A.main(i) @ m: M (r, glob_fin A) ]
        = mu (D (glob A){m} i) M.
```

(At its core the proof is based on Axiom of Choice.)

- We also showed a monadic structure on the program composition.

- This result allows users to transfer mathematical results to denotation of programs and the programs themselves.

- Using this approach we derived a useful tool-set of results which are common to cryptographic proofs
    - **Finite approximation**: good for proofs by induction
    - **Jensen's inequality**: bread-and-butter of cryptography
    - **Averaging** (also with infinite support)
    - **Rewinding**
    - …

Case-Study: Zero-knowledge

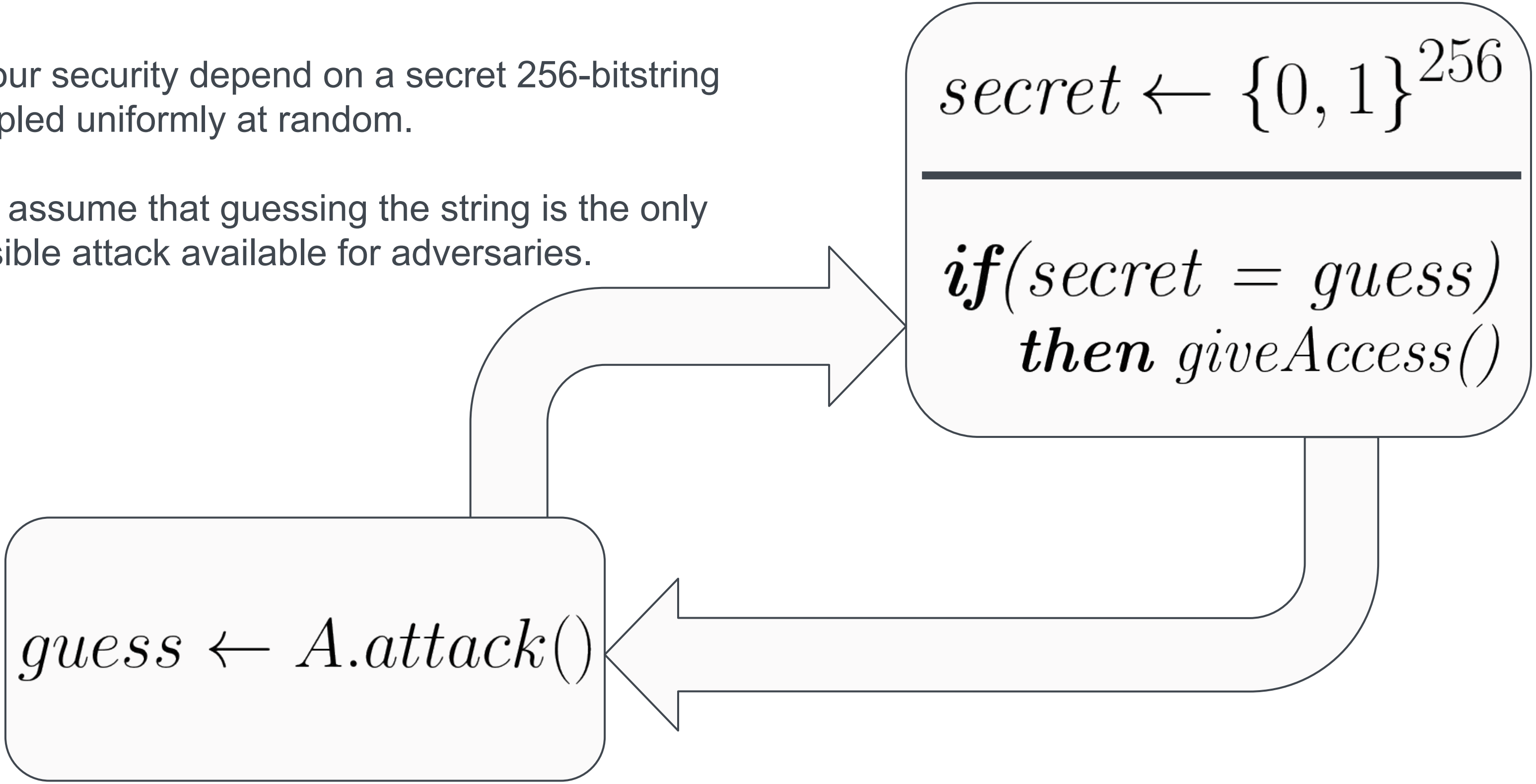
- We implemented a generic library of results for sigma-protocols.
- With reasonably small effort you can (semi-)automatically derive main properties for your favorite ZK sigma-protocol:
  - Completeness
  - Special Soundness
  - Extractability (from special soundness)
  - Soundness (from extractability)
  - Zero-Knowledge (from one-time simulators)
  - + Sequential Composition
- Proofs rely on lots of analysis and highly unlikely to be doable in program logics only.

Towards executable protocols!

How to get from mathematical EasyCrypt model of a protocol to the executable protocol and preserve the established guarantees and not introduce side-channels?

Motivation

- Let our security depend on a secret 256-bitstring sampled uniformly at random.
- Also assume that guessing the string is the only possible attack available for adversaries.



Motivation

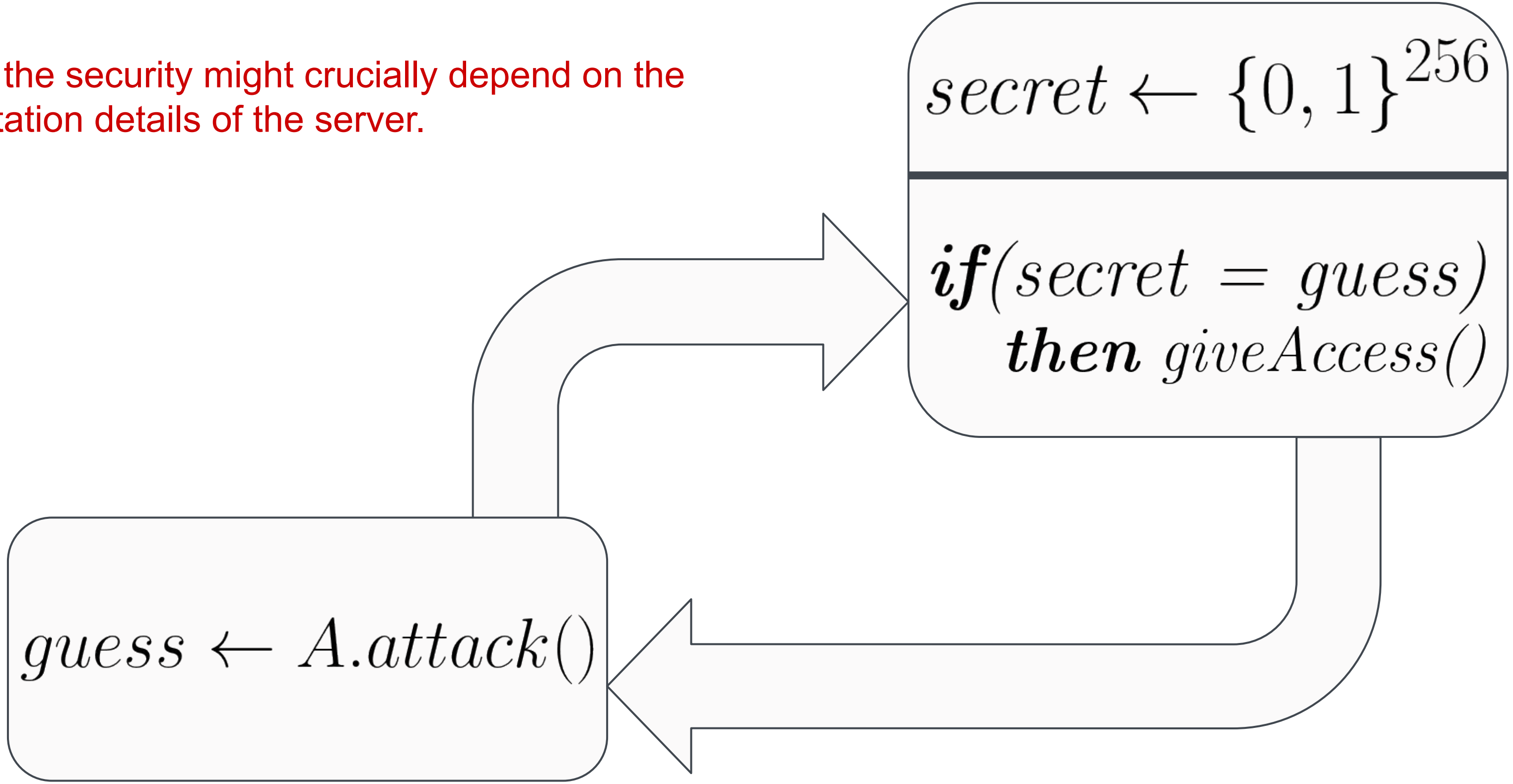
- What are the odds that an adversary will get access?
- The success of an adversary who does N tries is bounded from above as follows:

$$\mathbf{Pr}[\ adversary\ gets\ access\ ] \leq \frac{N}{2^{256}}$$

- So, in mathematical model we proved that our toy-system is “galactically” safe.

Motivation

In real life the security might crucially depend on the implementation details of the server.



Motivation

- For example, the optimizing compiler might decide to generate machine-code which checks equality UNTIL THE FIRST DIFFERENCE IS ENCOUNTERED.

secret:	b1	b2	b3	b4	…					
	?||	?||	?||	?||	…					
guess:	q1	q2	q3	q4	…					

- In this case if adversary can time responses of our server it can figure out the secret in a byte-by-byte manner with ~10^5 queries.

Motivation

- As a result we have discrepancy between the predictions of a mathematical model and the real-life implementation.

- The illustrated attack belongs to a family of side-channel attacks:
  - timing attack
  - cache side-channel attack
  - power-analysis attack
  - …

Challenge

How to show that an executable of a protocol is cryptographically secure and is free of side-channel attacks?

Jasmin programming workbench

- Jasmin combines high-level and low-level constructs to support “assembly in the head” programming paradigm.

- Programmers can control low-level features:
  - Instruction selection
  - Scheduling
  - Registers
  - Stack

- Also, programmers have “high-level” abstractions: variables, functions, arrays, loops, etc.

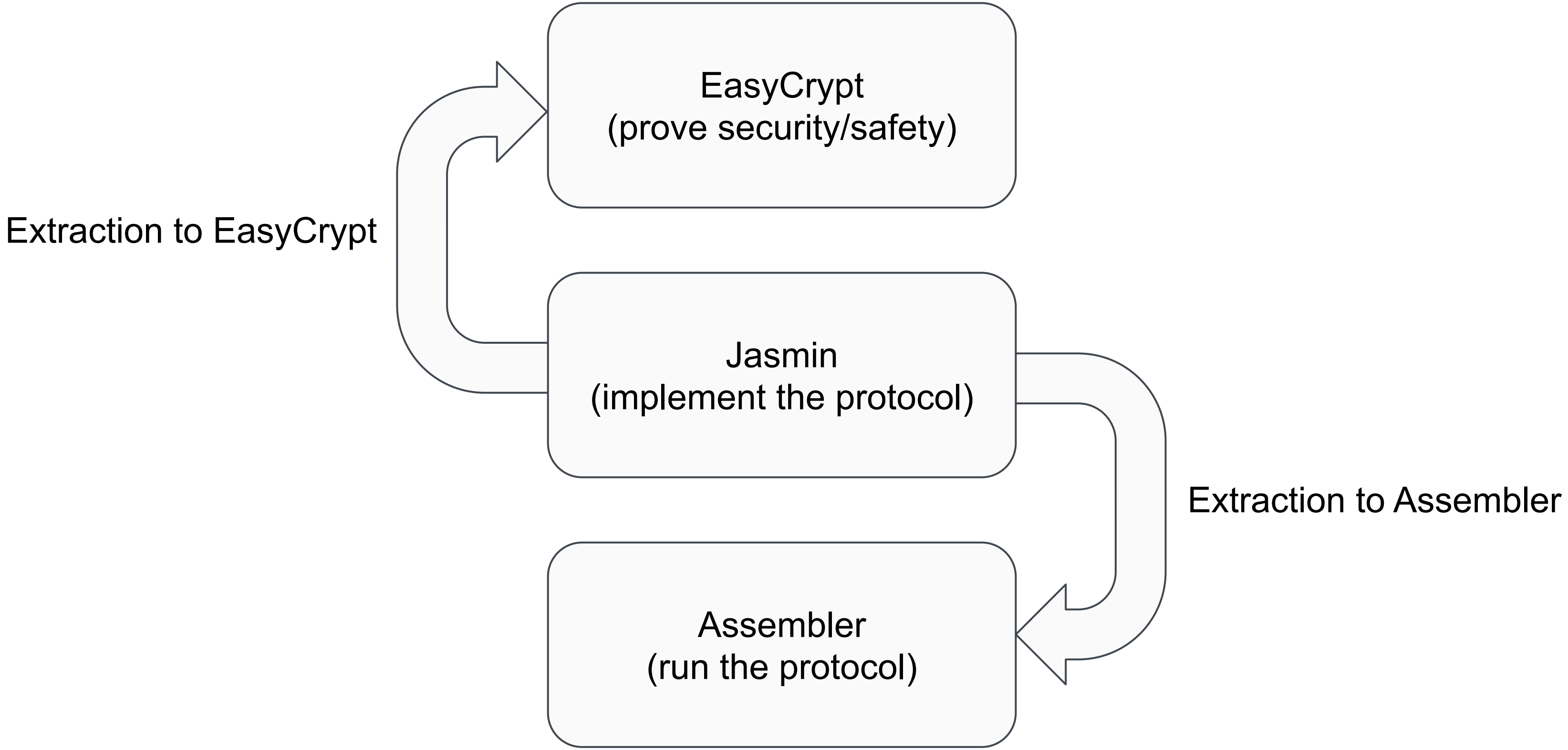
Jasmin programming workbench

- The semantics is formally defined in Coq to allow rigorous reasoning about program behaviors.

- Jasmin programs can be automatically checked for safety:
  - termination;
  - array accesses are in bounds;
  - memory accesses are valid;
  - validity of arguments.

- Moreover, Jasmin programs can be extracted to EasyCrypt theorem prover for formal verification:
  - functional correctness;
  - cryptographic security;
  - security against side-channel attacks.

Overview


EasyCrypt
(prove security/safety)
Extraction to EasyCrypt
Jasmin
(implement the protocol)
Extraction to Assembler
Assembler
(run the protocol)

Example: Swap operation

- Let us implement `swap` operation such that
  - `swap(x,y,0) == (x,y)`
  - `swap(x,y,1) == (y,x)`
- Guess what will go wrong with the naive “if-then-else” implementation.

Example: 256-bit `swap` in Jasmin

```
inline fn swap(stack u64[4] x, stack u64[4] y, reg u64 swap) -> (stack u64[4], stack u64[4]) {

  reg u64 tmp1, tmp2, mask;
  inline int i;

  mask = swap * 0xffffffffffffffff;

  for i = 0 to 4 {
      tmp1   = x[i];
      tmp1  ^= y[i];
      tmp1  &= mask;
      x[i]  ^= tmp1;
      tmp2   = y[i];
      tmp2  ^= tmp1;
      y[i] = tmp2;
  }
  return x, y;
}
```

Example: Extraction to EasyCrypt

```
proc swap (x : W64.t Array4.t, y : W64.t Array4.t, swap_0 : W64.t) : W64.t Array4.t * W64.t Array4.t = {
    var aux : int;
    var mask : W64.t;
    var i : int;
    var tmp1 : W64.t;
    var tmp2 : W64.t;

    mask <- (swap_0 * (W64.of_int 18446744073709551615));
    i <- 0;
    while (i < 4) {
      tmp1 <- x.[i];
      tmp1 <- (tmp1 `^` y.[i]);
      tmp1 <- (tmp1 `&` mask);
      x.[i] <- (x.[i] `^` tmp1);
      tmp2 <- y.[i];
      tmp2 <- (tmp2 `^` tmp1);
      y.[i] <- tmp2;
      i <- i + 1;
    }
    return (x, y);
}
```

Example: Functional correctness

We use the Hoare Logic of EasyCrypt to establish the functional correctness:

```
lemma swap_correct: ∀ a b f,
 Pr[ swap(a,b,f) = if f then (b,a) else (a,b)] = 1.
```

```
proc swap (x : W64.t Array4.t, y : W64.t Array4.t, swap_0 : W64.t) : W64.t Array4.t * W64.t Array4.t = {
      var aux_0 i : int;
      var aux mask : W64.t;
      var tmp1 tmp2 : W64.t;
      leakages <- LeakAddr([]) :: leakages;
      aux <- (swap_0 * (W64.of_int 18446744073709551615));
      mask <- aux;
      leakages <- LeakFor(0,4) :: LeakAddr([]) :: leakages;
      i <- 0;
      while (i < 4) {
        leakages <- LeakAddr([i]) :: leakages;
        aux <- x.[i];
        tmp1 <- aux;
        leakages <- LeakAddr([i]) :: leakages;
        aux <- (tmp1 `^` y.[i]);
        tmp1 <- aux;
        leakages <- LeakAddr([]) :: leakages;
        aux <- (tmp1 `&` mask);
        tmp1 <- aux;
        leakages <- LeakAddr([i]) :: leakages;
        aux <- (x.[i] `^` tmp1);
        leakages <- LeakAddr([i]) :: leakages;
        x.[i] <- aux;
        leakages <- LeakAddr([i]) :: leakages;
        aux <- y.[i];
        tmp2 <- aux;
        leakages <- LeakAddr([]) :: leakages;
        aux <- (tmp2 `^` tmp1);
        tmp2 <- aux;
        leakages <- LeakAddr([]) :: leakages;
        aux <- tmp2;
        leakages <- LeakAddr([i]) :: leakages;
        y.[i] <- aux;
        i <- i + 1;
      }
      return (x, y);
}
```

Example: Constant-timeness proof

Intuitively the following proves that `swap` does not leak anything about its arguments:

```
equiv swap_constant_time:
      swap ~ swap : ={M.leakages} ==> ={M.leakages}.
```

Example: Recap

- We used Jasmin to implement the `swap` function on 256-bit words.
- We extracted Jasmin implementation to EasyCrypt and proved functional correctness.
- We used “leakage”-annotated extraction to prove that the function is constant-time.
- In addition, we can use automatic checking to ensure memory safety, termination, etc.
- Finally, we can compile Jasmin to assembler which is preserves all the mentioned properties.

`swap` is great, but what about real cryptographic protocols?

Schnorr protocol in Jasmin

- In the Schnorr protocol the prover tries to convince a verifier that it knows a discrete logarithm of a statement.

- Maturity test case: Implement the Schnorr protocol in Jasmin and transfer the security proofs.

Honest prover (mathematical model)

```
module HonestProver = {
  proc commitment(s : statement, w : witness) : commitment = {
      r <-$ uniform_distr;
      return g ^ r;
  }

  proc response(b:challenge) : response = {
      return r + b * w;
  }
}.
```

Honest verifier (mathematical model)

```
module HonestVerifier = {
  proc challenge(s : statement, c : commitment) : challenge = {
      ch <$ dt;
      return ch;
  }

  proc verify(r : response) : bool = {
      return  g ^ r = (s ^ ch) * c;
  }
}.
```

Implementation in Jasmin?

- From the perspective of conventional programming both honest verifier and honest prover are exceptionally simple (but not the associated ZK properties).

- After all the implementation relies only on group operations, exponentiation, and sampling.

- Unfortunately, none of these operations are currently implemented in Jasmin in their full generality.

- For cryptographic protocols we need to develop an approach for sampling and prove indistinguishability results.
  - Perfect sampling is out-of-reach.

Example: Modular exponentiation

- We developed a modular exponentiation in Jasmin (denotationally just “(x ^ m) mod p”).
- However, the implementation makes use of specialized algorithms:
  - Montgomery ladder/form
  - Barrett reduction
- We proved that the result is correct, safe, and secure
  - Functional correctness (utilizes analysis in reals and then transfer to integer and then to machine words)
  - Memory safety properties
  - Side-channel freedom
- Implementation and proofs for “(x ^ m) mod p” ~1300loc.
- Performance wise we are 3x slower than specialized GMP library (not constant time, no correctness guarantees).

Jasmin goals

- The Jasmin workbench ambitiously aims at formal derivation of both high and low-level security properties.
- The approach needs more manpower to develop mature tools, libraries, and use cases.
- Most importantly, the resulting protocols must be executable, efficient, and provide unprecedented levels of security.

There is more!

- Resource analysis
    - In standard EC you must verify complexity of transformations by hand.
    - Resource analysis allows to prove the complexity bounds on transformations.
    - Also allows users to express properties more naturally.

- EasyPQC: for verification of post-quantum cryptography
    - Standard EC is not compatible with quantum cryptography

EasyCrypt applications

- Encryption schemes
    - Saber encryption at Crypto2022
- Commitments
    - hiding, binding, non-malleability
- Timestamping
    - Backdating-resistance analysis
- Digital signatures
    - Existential unforgeability
- Zero-knowledge
    - Sigma protocols
- Voting
- Differential privacy
- UC

Shortcomings

- Technical
  - Not (anymore) foundational
  - No parallelism
  - No timings
- General
  - Lack of educational resources
  - Partial and outdated manual
  - No good backwards compatibility
    - Tool is actively developed

Thank you!

guardtime.com

guardtime