Zero-Knowledge in EasyCrypt

Denis Firsov[1,2] and Dominique Unruh[3]

[1]Guardtime
[2]Tallinn University of Technology
[3]Tartu University

July 10, 2023

Main Goal

Implement a framework in EasyCrypt theorem prover which formally defines notions associated with sigma-protocols and provides generic lemmas which capture common patterns in proofs.

Main Contributions: Quick Overview

- Formal definitions: completeness, special soundness, soundness, proof-of-knowledge, zero-knowledge.
- Generic derivations (in computational and information-theoretical setting):
  - Proof-of-knowledge from special soundness.
  - Soundness from proof-of-knowledge.
  - Zero-knowledge from one-shot simulator.
  - Sequential composition.
- Use cases: Fiat-Shamir, Schnorr, and Blum protocols.
- Stepping stone for end-to-end verified and executable sigma-protocols.

Background: EasyCrypt

- EasyCrypt is a theorem prover for verifying cryptographic constructions, where protocols are specified as imperative programs and adversaries are modelled by abstract program modules.

Background: EasyCrypt

- EasyCrypt is a theorem prover for verifying cryptographic constructions, where protocols are specified as imperative programs and adversaries are modelled by abstract program modules.
- It has four built-in logics:
  - a probabilistic, relational Hoare logic (pRHL);
  - a probabilistic Hoare logic (pHL);
  - an ordinary (possibilistic) Hoare logic (HL);
  - an ambient higher-order logic for proving general mathematical facts and connecting judgments in the other logics.

Background: EasyCrypt

- EasyCrypt is a theorem prover for verifying cryptographic constructions, where protocols are specified as imperative programs and adversaries are modelled by abstract program modules.
- It has four built-in logics:
  - a probabilistic, relational Hoare logic (pRHL);
  - a probabilistic Hoare logic (pHL);
  - an ordinary (possibilistic) Hoare logic (HL);
  - an ambient higher-order logic for proving general mathematical facts and connecting judgments in the other logics.
- Also, we showed that EasyCrypt programs could be “reflected” into their probabilistic semantics to carry out proofs which rely on more advanced mathematical facts.

Notation

We write

$$\Pr[\, r \leftarrow X.\mathit{f(i)} \,@m : P \,]$$

to denote probability of event *P* after executing procedure *f* with argument *i* of module *X* at initial state *m*.

Background: Sigma Protocols

Every sigma protocol is designed to work with a specific formal NP-language. The language is induced by a relation between statements and witnesses. Then a protocol for the relation must allow the prover to convince the verifier that the prover knows a witness for some given statement without revealing anything else about itself.

$$\mathit{Prover}(s, w) \xrightarrow{\text{commitment}} \mathit{Verifier}(s)$$

$$\xleftarrow{\text{challenge}}$$

$$\xrightarrow{\text{response}} \quad \text{Accept/Reject}$$

Sigma protocols: Properties

- *Completeness* ensures the correct operation of the protocol if both prover and verifier follow the protocol honestly.
- *Soundness* ensures that for “wrong” statements (i.e., with no witness) a prover can convince the verifier with only small probability.
- *Proof-of-knowledge* guarantees that any prover that successfully convinces the verifier actually knows a witness (and not only abstractly that it exists).
- *Zero-knowledge* establishes that any cheating verifier cannot learn anything about the witness when running the protocol.

Sigma Protocols: Basic Parameters

```
abstract theory GenericProtocol.

  type statement.
  type witness.

  type relation = statement → witness → bool.

  op in_language (R:relation) statement: bool
      = ∃ witness, R statement witness.

  op completeness_relation    : relation.
  op soundness_relation       : relation.
  op zk_relation              : relation.
```

Sigma Protocols: Basic Parameters

```
  ...
  type commitment.
  type response.
  type challenge.

  type transcript = commitment × challenge × response.

  op verify_transcript: statement → transcript → bool.

  op challenge_set: challenge list.

  ...
end GenericProtocol.
```

Completeness

Completeness ensures the correct operation of the protocol if both prover and verifier follow the protocol **honestly.**

Completeness: Honest Prover

```
module type HonestProver = {
  proc commitment(s:statement,w:witness) : commitment
  proc response(ch:challenge) : response
}.
```

Completeness: Honest Verifier

```
module type HonestVerifier = {
  proc challenge(s:statement,c:commitment) : challenge
  proc verify(r:response) : bool
}.
```

Completeness: Game

In EasyCrypt we define completeness module to capture the interaction between honest parties:

```
module Completeness(P: HonestProver, V: HonestVerifier) = {

  proc run(s:statement, w:witness) = {
    var commit, challenge, response, accept;
    commit    <@ P.commitment(s,w);
    challenge <@ V.challenge(s,commit);
    response  <@ P.response(challenge);
    accept    <@ V.verify(response);
    return accept;
  }

}.
```

Completeness: Property

The module for default honest verifier **HV** is derived automatically. The user must specify honest prover **HP** and a completeness lower-bound δ:

```
op δ : real.

lemma statistical_completeness s w m: completeness_relation s w
    ⇒ Pr[out ← Completeness(HP,HV).run(s,w)@m: out ] ≥ δ.
```

"One-round" completeness must be proved manually.

Completeness: Sequential composition

One-round completeness implies completeness for sequential composition *generically*. Below, the module **CompletenessAmp** runs honest-interaction sequentially **n**-times.

```
lemma completeness_seq m s w n: completeness_relation s w ∧ 1 ≤ n
  ⇒ Pr[out ← CompletenessAmp(HP,HV).run(s,w,n)@m: out] ≥ δ^n.
```

Rewinding

Definition

The module A is rewindable if

1. There exists an injective mapping f from the type $\mathcal{G}_A$ to some parameter type *sbits*.
2. The module A must have a terminating procedure *getState*, so that the execution of *A.getState()* in state $\mathbf{m}$ must return the value $f(\mathcal{G}_A^{\mathbf{m}})$ without changing the state.

$$\Pr\left[r \leftarrow A.getState()\ @\mathbf{m} : \mathcal{G}_A^{\mathbf{fin}} = \mathcal{G}_A^{\mathbf{m}} \wedge r = f(\mathcal{G}_A^{\mathbf{m}})\right] = 1.$$

3. The module A must have a terminating procedure *setState*, so that whenever it gets an argument x : *sbits* and sets $\mathcal{G}_A^{\mathbf{m}}$ to $f^{-1}(x)$ if $f^{-1}(x)$ is defined. Formally, let g be of type $\mathcal{G}_A$ then

$$\Pr\left[r \leftarrow A.setState(f\ g)\ @\mathbf{m} : \mathcal{G}_A^{\mathbf{fin}} = g\right] = 1.$$

Zero-Knowledge

Zero-knowledge establishes that any **malicious rewindable** verifier cannot learn anything about the witness when running the protocol.

Zero-Knowledge: Rewindable Malicious Verifier

```
module type RewMaliciousVerifier = {

  proc challenge(s:statement, c:commitment): challenge
  proc summitup (r:response)   : summary

  proc getState()            : sbits
  proc setState(b:sbits)     : unit

}.
```

Zero-Knowledge: Distinguisher

```
module type ZKDistinguisher = {
  proc guess(s:statement,w:witness,sum:summary) : bool
}.
```

Zero-Knowledge: Real Experiment

```
module ZKReal(P: HonestProver,V: MaliciousVerifier,D: ZKDistinguisher)={

  proc run(s:statement, w:witness) = {
    var commit, challenge, response, summary, guess;

    commit    <@ P.commitment(s,w);
    challenge <@ V.challenge(s,commit);
    response  <@ P.response(challenge);
    summary   <@ V.summitup(s,response);

    guess     <@ D.guess(s,w,summary);
    return guess;
  }

}.
```

Zero-Knowledge: Simulator for Ideal Game

```
module type Simulator(V: RewMaliciousVerifier) = {
  proc simulate(s: statement) : summary
}.
```

Zero-Knowledge: Ideal Experiment

```
module ZKIdeal(S:Simulator,V:RewMaliciousVerifier,D:ZKDistinguisher) = {

    proc run(s: statement, w: witness) = {
      var summary, guess;

      summary <@ S(V).simulate(s);
      guess   <@ D.guess(s,w,summary);

      return guess;
    }

}.
```

Zero-Knowledge: Desired Property

There must exist an efficient simulator **Sim** so that for any rewindable malicious verifier **V**, and distinguisher **D** the absolute difference between real and ideal games is bounded from above by ε:

```
op ε: real.

lemma statistical_zk s w m: zk_relation s w ⇒
 let real_prob  = Pr[out ← ZKReal(HP,V,D).run(s,w)@m: out] in
 let ideal_prob = Pr[out ← ZKIdeal(Sim,V,D).run(s,w)@m: out] in

                  |ideal_prob - real_prob| ≤ ε.
```

Zero-Knowledge: Direct proofs are hard!

Proving zero-knowledge directly could be challenging. Alternative common strategy is to derive zero-knowledge from “one-shot” simulator.

Zero-Knowledge: One-Shot Simulator

- One-shot simulator **Sim1** is a simulator which in addition to summary returns a “success-event”:

```
module type Simulator1(V: RewMaliciousVerifier) = {
  proc run(s: statement) : bool × summary
}.
```

- We ask for the lower-bound σ on that “success-event”

```
op σ : real.

lemma sim1_lower_bound stat m:
  Pr[ (succ, _) ← Sim1(V).run(stat)@m: succ ] ≥ σ.
```

- We also ask simulator to rewind itself and the malicious verifer in case it was not successfull:

```
lemma rewind_sim istate m: (glob Sim1(V)) = istate
  Pr[ (succ, _) ← Sim1(V).run(s)@m:
        !succ ⇒ (glob Sim1(V)) = istate] = 1.
```

Zero-Knowledge: One-Shot Simulator

The absolute difference between success-probabilities of the real game conditioned on the success-event and the ideal game must be bounded from above by ε:

```
lemma sim1_zk_cond3 s w m: zk_relation s w ⇒
 let sim1_real
  = Pr[(succ, out) ← ZKReal'(HP,V,D).run(s,w)@m: succ ∧ out] in
 let sim1_ideal = Pr[out ← ZKIdeal(Sim1,V,D).main(s,w)@m: out] in
 let succ_event = Pr[(succ, _) ← Sim1(V).run(s)@m: succ] in


            |sim1_real / succ_event - sim1_ideal| ≤ ε.
```

Zero-Knowledge: Many-Shot Simulator for One-Round ZK

Given such one-shot simulator we define simulator **SimN** which runs one-shot simulator until it succeeds, but at most **N** times. Then we generically conclude the following statistical zero-knowledge:

```
lemma statistical_zk s w m: zk_relation s w ⇒
 let real_prob = Pr[out ← ZKReal(HP,V,D).run(s,w)@m: out] in
 let ideal_prob = Pr[out ← ZKIdeal(SimN,V,D).run(s,w)@m: out] in

          |ideal_prob - real_prob| ≤ ε + 2 · (1 - σ)^N.
```

Zero-Knowledge: Sequential Composition

From one-round zero-knowledge we can conclude multiple-round zero-knowledge generically!

Zero-Knowledge: Sequential Composition

We define “sequentially” composed “real” experiment:

```
module ZKRealAmp(P:HonestProver,V:MaliciousVerifier,D:ZKDistinguisher)={
  proc run(s: statement, w: witness) = {
    var commit, challenge, response, summary, guess,i;
    i ← 0;

    while(i < n){
      commit    <@ P.commitment(s,w);
      challenge <@ V.challenge(s,commit);
      response  <@ P.response(challenge);
      summary   <@ V.summitup(response);
      i ← i + 1;
    }

    guess <@ D.guess(s,w,summary);
    return guess;
  }
}.
```

Zero-Knowledge: Sequential Composition

Ideal game for sequentially composed ZK does not change.

Zero-Knowledge: Multiple-Run Simulator

Generic transformation of one-run to multiple-run simulator:

```
module SimAmp(S:Simulator,V:RewMaliciousVerifier) = {
  proc simulate(s:statement) = {
   var summary, i;
   i ← 0;

   while(i < n) {
     summary <@ S(V).simulate(s);
     i ← i + 1;
   }

   return summary;
  }
}.
```

Zero-Knowledge: Sequential Composition Generically

If **Sim** is δ-one-run simulator then **SimAmp(Sim)** is a **n**δ-multiple-run simulator for sequentially composed ZK:

```
lemma zk_seq s w m:
 let ideal_prob = Pr[out ← ZKIdeal(SimAmp(Sim),V,D).run(s,w)@m: out] in
 let real_prob  = Pr[out ← ZKRealAmp(P,V,D).run(s,w)@m: out] in

              |ideal_prob - real_prob| ≤ n · δ.
```

More generic derivations

- Proof-of-knowledge from special soundness.
- Soundness from proof-of-knowledge.

Use cases

- Schnorr protocol (discrete logarithm).
- Fiat-Shamir protocol (quadratic residue).
- Blum protocol (Hamiltonian cycles, NP-complete).

Fiat-Shamir protocol

- Completeness + sequential composition (50 lines of code);
- Special Soundness (60 lines of code);
- Proof-of-Knowledge (40 lines of code);
- Soundness + sequential composition (30 lines of code);
- One-Shot Simulator (200 lines of code).
- Zero-Knowledge + sequential composition (50 lines of code).

Conclusions

- It is relatively simple to instantiate and derive properties of sigma-protocols in our EasyCrypt framework.
- The downside of EasyCrypt formalizations is that the resulting protocols are not executable.
- In EasyCrypt formalizations are usually done at the very high-level of abstraction.
- For example, protocols are usually developed in context of abstract groups, particular distributions, etc.
- The naive compilation from high-level to low-level is not guaranteed to preserve cryptographic properties like zero-knowledge.

Work in progress: Sigma Protocols in Jasmin

- In the further work we implement sigma-protocols in assembly via Jasmin toolchain.
- Jasmin is a low-level programming language for high-assurance and high-speed cryptography.
- Jasmin programs can be extracted to EasyCrypt to address functional correctness, cryptographic security, or security against timing attacks.
- We derive properties for the sigma protocols in Jasmin by carrying them over from the our ZK framework.

Thank you!