Formalizing Quadratic Residue Zero Knowledge Proof in EasyCrypt

Shriya Thakur
University of Massachusetts Lowell, Lowell, USA

Frederick Stock
University of Massachusetts Lowell, Lowell, USA

Adriano Corbelino II
University of Massachusetts Lowell, Lowell, USA

1 Introduction

During the semester, we learned about zero-knowledge proofs (ZKP) – a protocol between two parties – a *prover* and a *verifier*, where the prover tries to prove a secret fact or statement to a verifier without revealing the statement to the verifier. We have used the Easycrypt proof assistant to formalize these properties.

Our zero-knowledge protocol is an interactive proof system which means that the protocol must have three properties: completeness, soundness, and zero knowledge [2]. Completeness assures us that if both parties follow the protocol honestly, the verifier will accept. On the other hand, Soundness tells us that no "dishonest" prover can convince a verifier that a false statement is true with a high probability. In our case, this probability should be greater than or equal to 1/2. Lastly, zero knowledge ensures that no information is leaked during the run of this protocol besides the truth value of the public statement. In reality, the knowledge gain is measured under a relaxed measure, where we can argue that it is computationally not feasible for the verifier to extract any meaningful information or simply that the verifier has a small probability of obtaining new information.

Some Zero-knowledge proofs are classified under the Σ-protocols umbrella [6]. Σ-protocols are protocols that have a structure of information flow that resembles a capital Σ. In these protocols, both the prover and the verifier have access to a claim s, and the prover is supposed to know a witness x to the claim The communication has four steps. First, the prover creates a commitment a. This commitment is then shared with the verifier. It is used to increase the verifier's trust in the prover's claim. Next, the verifier generates a challenge and sends it to the prover. Third, the prover responds to the verifier depending on the challenge that is given in the previous step. Lastly, the verifier checks the prover's response and returns its final answer.

To simplify our discussion, we follow a standard cryptographic naming scheme and use Alice to refer to the prover and Bob to refer to the verifier. In section 2 we define the quadratic residue protocol. In Section 3 we present informal "pen and paper" proofs of completeness, soundness, and zero-knowledge. In Section 4 we describe the work-in-progress zero-knowledge proof that is the basis of our project. Finally Section 5 we present our EasyCrypt formalization, and discuss where we succeeded v/s failed.

2 The Protocol

The Quadratic Residue protocol is a communication between two parties, a prover (Alice) and a verifier (Bob). It is assumed that there is a shared value $n \in \mathbb{Z}$, of which both Alice and Bob are aware. Alice is assumed to have knowledge of a number $x \in \mathbb{Z}_n^*$ such that for $s \in \mathbb{Z}_n^*$ $x^2 = s$ Assuming that both parties are not malicious, the communication follows the following steps:

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Alice	Bob
Publish s, a Quadratic Residue of $\mathbb{Z}_n^*$	
Draw a value at random from $\mathbb{Z}_n^*$, $y \xleftarrow{R} \mathbb{Z}_n^*$	
Send $a = y^2$ to Bob	Compute a random bit $b \xleftarrow{R} \{0,1\}$
	Send b to Alice
$z = \begin{cases} y & \text{if } b = 0 \\ xy & \text{if } b = 1 \end{cases}$	
Send z to Bob	
	Verify $z^2 = \begin{cases} a & \text{if } b = 0 \\ sa & \text{if } b = 1 \end{cases}$

Table 1 The steps of the Quadratic Residue ZKP

First, Alice samples a random number y from $\mathbb{Z}_n^*$, and sends a, which is equal to y^2, to Bob. Second, Bob will generate and send a random challenge bit b. Third, Alice sends a response z to Bob's challenge. If the challenge bit is 0, z equals y. Otherwise, z equals xy. Lastly, Bob's final response is the result of the check if z^2 is a if $b = 0$ or sa if $b = 1$.

3 Pen and Paper Proofs

There are three properties a zero-knowledge proof needs to have. Completeness, Soundness, and Zero-knowledge, a rough description of each follows:

Completeness: If Alice and Bob are a honest prover and verifier, then Bob accepts with probability 1.

Soundness: If Alice is dishonest, meaning she presents a value s that is not a quadratic residue, then Bob rejects with probability $\leq 1/2$.

Zero-knowledge: If Bob is dishonest, then he can learn nothing about the value of x from interacting with Alice.

All of those properties have three flavors: Perfect, Statistical, and Computational [4]. Those flavors are related to the strength of the property itself. A perfect property states that something must always happen; in other words, it must have a probability of 1. A statistical guarantees that something will be valid within some predefined error and without considering the computational power of both parts. Lastly, A computational property says that a statement will be valid within some error and considering the computational power involved. In our work consider Perfect Completeness and Statistical Soundness and Zero-Knowledge.

We present the standard "pen and paper" proofs of these three properties, before discussing implementations of these proofs in EasyCrypt. Though they are our own work, these proofs were constructed with help from a set of lecture notes by Boaz Barak [1]. This source helped provide an understanding of the specific mathematical definition of each of these properties and demonstrate how to structure these proofs, namely the proof of zero-knowledge (section 3.3).

3.1 Completeness

▶ **Lemma 1.** *If Alice and Bob are honest provers and verifiers (they follow the protocol described in section 2), then Pr(Bob accepts)* $= 1$.

Proof. This lemma is proven by simple computation, we proceed by case analysis on the challenge bit b:

$b = 0$ In this case, Alice would respond to the challenge bit by sending $z = y$ and Bob would verify that $z^2 = a$. By definition, since Alice is honest, $a = y^2$, and $z = y$, then $z^2 = y^2 = a$ This is trivially true.

$b = 1$ Alice would respond to b by sending $z = xy$ and Bob verifies that $z^2 = sa$. By definition, $s = x^2$ and since $z = yr$. Then $z^2 = (yr)^2 = y^2r^2 = ax$.

Therefore, if Alice and Bob are honest, then Bob will accept with probability 1. ◀

3.2 Soundness

If a protocol is *Sound*, then a dishonest prover cannot consistently convince a verifier of its knowledge ($Pr(reject \mid dishonest\ prover) \geq 1/2$). We say a prover is *dishonest* if the value s, that it claims is a quadratic residue is not. In other words, Alice is dishonest if she claims to Bob that s is a quadratic residue when there is no $x \in \mathbb{Z}_n^*$ such that $x^2 = s$.

▶ **Lemma 2.** *If Alice is a dishonest prover, then Pr(Bob rejects)* $\geq 1/2$.

Proof. Observe that, Alice sends a to Bob before receiving the challenge bit b. We, therefore, proceed by casework on a as a malicious Alice must determine a value of a that will trick Bob regardless of the value of b.

a is a quadratic residue of $\mathbb{Z}_n^*$: Let $y^2 = a$. If $b = 1$, then Alice needs to send a value z to Bob such that $z^2 = ax$. However, if $z^2 = ax$, then $z^2y^{-2} = (ax) \times y^{-2} = (y^2x) \times y^{-2} = x$. Clearly z^2y^{-2} is a quadratic residue of $\mathbb{Z}_n^*$ as $zy^{-1} \in \mathbb{Z}_n^*$. This would imply that x is a quadratic residue of $\mathbb{Z}_n^*$. But this is a contradiction since Alice is a dishonest prover, we assume x is not a quadratic residue. Therefore, in this case, Bob will reject with a probability of at least $1/2$ since Bob always reject when $b = 1$.

a is not a quadratic residue of $\mathbb{Z}_n^*$: If y is not a quadratic residue, then when $b = 0$, Alice must send some value z to Bob such that $z^2 = a$. However, this is impossible if y is not a quadratic residue. Therefore, Bob rejects with at least probability $1/2$.

These are the only two cases for a. In either case, Bob will reject with a probability of at least $1/2$. Therefore, if Alice is dishonest, Pr(Bob rejects) $\geq 1/2$. ◀

3.3 Zero-Knowledge

In 1988, a paper by Feige, Fiat, and Shamir [3] defined Zero Knowledge as:

▶ **Definition 3.** *An interactive proof system of membership in L is zero knowledge, if, for all inputs restricted to L, for all B and KB its view of the communication in* $(\bar{A}, B)$ *can be recreated, by a polynomial-time probabilistic Turing machine M, with an indistinguishable probability distribution.*

Essentially, if Bob is a malicious verifier, attempting to learn more about x. They would do so by interacting with Alice and from the communication attempt to glean extra information about x. Therefore, if an equivalent communication (or transcript) could be efficiently generated by a program with no knowledge of x, then if Bob has no extra information than the messages exchanged with Alice, he should not be able to learn anything about the true value of x.

Algorithm 1 A simulator for the QR Protocol

$y' \xleftarrow{R} \mathbb{Z}_n^*$
$b' \xleftarrow{R} \{0,1\}$
$z' \leftarrow \begin{cases} y'^2 & \text{if } b' = 0 \\ y'^2/s & \text{if } b' = 1 \end{cases}$
Communicate with Bob by sending z' and receive a challenge bit b
if $b' = b$ **then**
  Communicate with Bob by sending y'^2
  Publish the transcript
else
  Restart the algorithm
end if

We show that the Quadratic Residue protocol is zero-knowledge. First, however, we define a simulator that will be used in our proof

This simulator attempts to "guess" the value of the challenge bit that Bob will send. If this value is successfully guessed then the simulator can tailor its choice of z'. If $b = 0$ Bob will attempt verify that $z' = y'2$. If $b' = 0$ then the simulator sets $z' = y'$ so clearly this will succeed. If $b' = 1$ then Bob will verify that $y'^2 = z' * s$. Since the simulator set $z' = y'^2/s$ then $z' * s = y'^2/s * s = y'^2$. So Bob will always accept the values provided by this simulator.

▶ **Lemma 4.** *The Quadratic Residue ZKP is a Zero Knowledge Protocol*

Proof. We claim that Algorithm 1 is a simulator that satisfies these conditions. There are two facts to prove. First, that the transcript produced by Algorithm 1 is indistinguishable from a protocol between two honest parties. Second, that this algorithm terminates in polynomial time.

Note, that there is only one step Alice takes in Section 2 that is random. The first step, where y is created. This value is generated by drawing randomly from the values of $\mathbb{Z}_n^*$. Algorithm 1 generates its equivalent value, y', in the same way so these values are probabilistically indistinguishable. The only other random action in Algorithm 1 is when b' is generated. However, Algorithm 1 only terminates if b' equals the value of b generated by Bob. Therefore, for any published transcript b' has the same distribution as b.

So Algorithm 1 produces a transcript that is indistinguishable from a communication between two honest parties following this protocol. All that is left is to show that Algorithm 1 terminates in expected polynomial time. This is nearly trivial. The algorithm will end if $b' = b$. Both of these values are generated by a uniformly random drawing over $\{0,1\}$. Therefore, $Pr(b' = b) = 1/2$ and so, in expectation, this algorithm should terminate after 2 iterations. ◀

4 EasyCrypt Implementation

We have chosen EasyCrypt as our implementation language because it supports probabilistic reasoning via game-based proofs. In this theorem prover, we can use different logics to make probabilistic statements and relate programs involving them. The Ambient logic is the principal and it is very similar to what we have in Coq. In addition, we have three kinds of Hoare triples: plain triples, which are used to reassure pre and post-conditions; Probabilistic Hoare Logic, which is used to reason about probabilities of a single imperative procedure

call; and Probabilistic Relational Hoare Logic, which is used to relate two different programs [7]. We could have chosen other tools, such as SSProve [5], but EasyCrypt is a more mature tool, tailored for this kind of problem and with more learning resources.

Our implementation begins with necessary imports including Bool (for true, false, and their logical operators), IntDiv (for integer division and related modular arithmetic), DBool (for distributions on boolean values while sampling), Distr (to define distributions on various types), DInterval for uniform distributions on the integers, $ZmodP$ provides us an abstract ring of integers modulo P, and lastly we clone the module defining a ring and rename it to ZMR. Next, we define an operator zmod_distr as a uniform distribution over the type $zmod$. At this point, we are ready to define some types for our game. We start by defining the representation of our statement, commitment, and witness as the $ZmodP$ type and the challenge as a boolean. In our protocol, statement refers to the public input s which comes from the private witness x. The commitment represents the provers chosen number masked with modular arithmetic, the challenge represents the challenge bit generated by the verifier which can be one of 0 or 1 in one interaction of the protocol, and finally response is the response that the prover sends back to the verifier depending on the challenge bit received from the verifier.

At this stage, we can make our prover and verifier modules. We first define a prover type as a module that contains procedures for initializing the prover, generating the commitment, and responding to the verifier's challenge with their input and output types. Similarly, we define a verifier module to initialize the verifier, generate and send the challenge bit, and verify the provers response with all their input and output types as seen below:

```
module type Prover = {
  proc init(s: statement, x:witness): unit
  proc gen_commitment(): commitment
  proc res_challenge(b:bool): response
}.

module type Verifier = {
  proc init(s:statement): unit
  proc gen_challenge(c: commitment): challenge
  proc verify_res(r: response): bool
}.
```

The HonestProver and HonestVerifier modules are modules that are parameterized by their respective type interface — Prover for prover type and Verifier for verifier—. These modules contain implementations for each of the procedures that we described in their type interface. The prover module is as follows:

```
module HonestProver: Prover = {
  var s: statement
  var x: witness
  var y: commitment

  proc init(s2:statement, x2:witness): unit = {
  s <- s2;
  x <- x2;
  }
```

```
  proc gen_commitment(): commitment = {
      y <$ zmod_distr;
      return y * y;
  }

  proc res_challenge(c:challenge): response = {
    var resp: response;
      if (c) {
      resp <- y * x;
    }
      else {
      resp <- y;
    }
      return resp;
  }
}.
```

We start by declaring the global variables mainly– public statement s, the private witness x, and the commitment generated by the prover y. The procedure **init** is similar to a constructor, which initializes the HonestProvers' global variables with the statement s2 and witness x2. **gen_commitment** is then used to sample a random value y from the uniform distribution of integers ZMoDP and send the masked value y^2 to the verifier. In reality the prover sends back y^2 *mod* p, but since we are in the group Z_p, we can omit writing mod p each time. Finally, **res_challenge** is a procedure that responds to the verifiers' challenge. If the verifier sends a challenge bit 0 then the prover responds with the original value, y, that was chosen. Likewise, if the verifier sends a challenge bit 1 then the prover responds with $y * x$ according to the protocol. Now we will go over the Honestverifier module which is as follows:

```
module HonestVerifier: Verifier = {
  var ch: bool
  var s: statement
  var a: commitment
  proc init( s2:statement): unit = {
    s <- s2;
  }

  proc gen_challenge(c:commitment): challenge = {
    a <- c;
    ch <$ {0,1};
    return ch;
  }

  proc verify_res(resp: response): bool = {
    var result: bool;
    if (ch) {
      result <- (resp * resp) = (a * s);
    }
    else {
      result <- (resp * resp) = a;
```

```
    }
    return result;
  }
}.
```

Our Honestverifier module is paramaterized by the Veririfer type. It contains some global variables including the public challenge bit ch, the public statement s, and a– the commitment sent back by the prover. We then initialise the verifier with statement s_2 using the constructor. In **gen_challenge**, the verifier stores the commitment received in variable a. It then chooses a random challenge bit from the sampling $0, 1$ and sends it to the prover. Lastly, the verifier checks the provers' response. If the challenge bit was 1, then the verifier checks if the square of the response received is equal to the product of commitment and statement. Else, if $b = 0$, then it checks if the square of the provers' response is equal to a. In this way, the verifier learned nothing but the fact that the given statement has a quadratic residue or not. From this interaction between the HonestProver and Honestverifier modules, we can derive completeness as shown in the following section.

4.1 Completeness

We model completeness as a game between the Honest Prover and Honest Verifier as seen below:

```
module Completeness(HP: Prover, HV: Verifier) = {
  proc run(s:statement, x:witness): bool = {
  var cm,ch,resp,result;
  HP.init(s, x);
  HV.init(s);
  cm <@ HP.gen_commitment();
  ch <@ HV.gen_challenge(cm);
  resp <@ HP.res_challenge(ch);
  result <@ HV.verify_res(resp);
  return result;
  }
}.
```

Within this module is a procedure named run which contains the interaction between the two modules:

1. It begins by initializing the HonestProver and HonestVerifier.
2. The HonestProver then generates and sends a commitment which is stored in the variable cm.
3. Next, the HonestVerifier issues a challenge bit and sends it to the HonestProver.
4. Depending on the challenge bit issued by the verifier, the Honestprover sends back a response which is captured in a variable named resp.
5. The verifier then checks the response sent by the prover and returns a boolean result depending on whether the check returned true or false.

Next, we formally state and prove our lemma for perfect completeness as follows:

```
lemma completeness: forall (s: statement) (x: witness), s = x * x =>
    hoare[Completeness(HonestProver, HonestVerifier).run : arg = (s,x) ==> res].
proof.
```

```
    move =>  s x H0.
    proc. inline*.
    wp. rnd. wp. rnd. wp.
    skip. simplify. progress. smt(@ZMR).
qed.
```

Our lemma for completeness is represented as a Hoare triple that contains the preconditions, program, and post-conditions. Formally, this states that for all statements s and witness x, if we know a precondition that $s = x^2$ holds and we know that both parties are honest, then the interaction between the two parties contained in the procedure named run always succeeds — terminates with the verifier accepting the validity of the claim —. In this case, **arg** and **res** are keywords in easycrypt. **arg** contains the inputs that were sent to the initial procedure and **res** contains the value returned by the procedure. So in our case, **res** contains the same value in memory as that of **result** at the end of our game. For example, if we had a different prover, that always returned $false$, we would not be able to complete this proof. In this aforementioned scenario, we would be required to prove that a boolean is not in the set $\{0, 1\}$, which in EasyCrypt is a contradiction with the definitions of $DBool$. Following this, we can now step through the proof one tactic at a time as follows:

1. **move**: The move tactic helps us to introduce the quantified variables s and x into the context. It also introduces our precondition or assumption that $s = x^2$ into our context and renames it to H0.
2. **proc**: Since the game between HonestProver and Honestverifier contains procedures, we can unfold these procedures using the tactic proc.
3. **inline***: This tactic helps us inline all the procedures that are contained in the HonestProver and HonestVerifier modules. We could also explicitly state all the tactics using the tactic inline followed by the name of the procedure, but to make it less verbose, we chose to use the asterisk.
4. At this point our proof has a precondition (s, x), a postcondition res, and the entire program, which is about 18 lines.
5. **wp**: Since our goal is a probabilistic Hoare logic statement judgment, which ends with an ordinary assignment (represented by the $\longleftarrow$) we run wp to consume this assignment and any if statements replacing the programs post-conditions by the weakest precondition.
6. **rnd**: Our program now ends with a random assignment so we use rnd to consume it.
7. **skip**: After multiple such applications of wp and rnd tactics we finally end up with just pre and post-conditions with no program. So we can use the tactic skip. Skip reduces the goal we have to an ambient logic formula as an implication between the precondition and post-condition.
8. **Simplify**: We run simplify to make reading the goal a bit simpler.
9. **Progress**: We use progress to break down our given goal into simpler goals. Progress is a more complex tactic that contains repeated applications of move, substitute, split, and trivial.
10. **smt(@ZMR)**: At this point, we are just left with modular arithmetic for rings. So we use smt solvers. Giving smt() the exact name of the library/solver makes it more explicit to the solver so the solver does not have to search or get lost searching down an incorrect path.
11. This completes our proof for completeness.

4.2 Soundness

To define statical Soundness we can start with the definition of a new module type for all malicious provers. The only difference between this module type and the Prover type is that it stresses the fact that a dishonest prover does not have access to a witness for the claim. With this definition, we can define the Soundness game:

```
module type MProver = {
  proc init(s: statement): unit
  proc gen_commitment(): commitment
  proc res_challenge(b: bool): response
}.

module Soundness(MP: MProver, HV: Verifier) = {
  proc run(s:statement): bool = {
    var cm,ch,resp,result;
    MP.init(s);
    HV.init(s);
    cm <@ MP.gen_commitment();
    ch <@ HV.gen_challenge(cm);
    resp <@ MP.res_challenge(ch);
    result <@ HV.verify_res(resp);
    return result;
  }
}.
```

But, how can we talk about those malicious provers? One naive approach is to have a concrete implementation; in other words, we would have to fix a specific malicious prover as our representative candidate for all malicious provers. We could work to find and argue that a specific malicious prover is the strongest, but this is less general and quickly derailed our thinking process. Another approach is to simply quantify the overall possible malicious prover and have a generic representation of a malicious prover. This approach is more general and goes along with what we have done in our paper proof. With that, we can write a formal statement for soundness and a formal proof sketch.

```
lemma soundness: forall (s:statement) (x:witness) (MP <: MProver) &m,
    s <> x * x =>
    Pr[Soundness(MP, HonestVerifier).run(s) @ &m : res] <= 0.5.
proof.
    progress.
    byphoare.
    proc. inline*.
    wp. admit.
    trivial.
    trivial.
qed.
```

This lemma states that for every two numbers that are not valid statements and witnesses, every malicious prover should have at most 50% of chance to convince the verifier. However, this framing imposes different challenges. Since we have this abstract candidate, we can only

interact with it through the interface declared in the module type, and we can not use the same unfolding strategy that we have used in the Completeness proof. This creates some issues, especially with assignments that the value came from an abstract method. We suspect that in those cases, we should use the **call** tactic, but the documentation is not clear about how we should frame our conditions when working with abstract calls.

The only new tactic used in the previous proof is **byphoare**. This tactic transforms a probability hoare statement into just a regular hoare logic statement equipped with some restriction regarding the probability of the result. Since we are proving statistical Soundness, we need to work with the probability of success of a single procedure call, and this requires us to change to the probabilistic Hoare world. Another challenge is that our informal proof is not intuitionist, we used the law of the excluded middle to reason about if a value is or is not a quadratic residue. However, in most theorem provers, not every proposition is decidable, and we need to define extra machinery to enable us to make a similar proof in a formal proof. For example, we could have added axioms to freely allow this type of argument, just like in [4].

4.3 Zero-Knowledge

Our third and last property to prove is zero knowledge of our protocol. Intuitively, zero knowledge, in our case, means that the verifier learns nothing more than that s is a valid statement. One way to prove zero knowledge is the Real-Ideal paradigm. This paradigm is from theoretical cryptography and has been used to define security and privacy in the past. We use the real-ideal paradigm to establish indistinguishability between the real and ideal protocol execution. Each of these makes a transcript and we will try to prove that these two transcripts are (probabilistically) indistinguishable. Intuitively speaking, indistinguishability states that for any malicious party interacting with the real protocol, there exists a simulator such that the malicious party cannot distinguish whether it is interacting with the real protocol or the ideal version. The real-ideal paradigm is a straightforward way to formalize the proof from Section 3.3. We start by defining the real and ideal world as follows:

1. Real scenario: An interaction or run between the honest prover and honest verifier.
2. Ideal scenario: A simulator run that generates a fake transcript that looks like a real one but is constructed without access to the witness x.

Showing that an adversary (in this case, the verifier) cannot distinguish between the real and ideal transcripts will prove indistinguishability. Let's start by looking at the **Real scenario** as mentioned below:

```
module type RealTrans = {
  proc generate(s: statement, x: witness): (commitment * challenge * response)
}.

module RealTranscript: RealTrans = {
  proc generate(s: statement, x: witness): (commitment * challenge * response) = {
    var c, ch, resp;

    HonestProver.init(s, x);
    HonestVerifier.init(s);

    c <@ HonestProver.gen_commitment();
    ch <@ HonestVerifier.gen_challenge(c);
```

```
    resp <@ HonestProver.res_challenge(ch);

    return (c, ch, resp);
  }
}.
```

We use a statement s and witness x. We start by initializing the honest prover and honest verifier modules. This protocol generates the transcript using the `HonestProver` and `HonestVerifier` interaction. We have the commitment c, the value generated by the prover whose square is sent to the verifier, a challenge bit b, and a response named $resp$. We return a tuple consisting of the commitment, challenge bit, and response to ensure that the verifier can reconstruct the entire transcript to verify the interaction. We do not add the witness x to the tuple since that would mean we would leak private information. Now we can look at the **Ideal scenario** as follows:

```
module type Simulate = {
proc init(s2: statement): unit
proc simulate(s: statement): (commitment * challenge * response)
}.

module Simulator: Simulate = {
  var s: statement
  var b: bool
  var c: response

  proc init(s2: statement): unit = {
    s <- s2;
  }

  proc simulate(s: statement): (commitment * challenge * response) = {
    var resp: commitment;
    var s': commitment;

    (* Use s as input *)
    b <$ {0,1};
    c <$ zmod_distr;

    if (b) {
      s' <- inv s;
      resp <- (c * c) * s';
    } else {
      resp <-  c * c;
    }

    return (resp, b, c);
  }
}.
```

This module produces a transcript that mimics the real protocol but does not depend on the knowledge of the witness x chosen by the prover. The simulator module consists of global variables, which store the public statement provided during initialization, a challenge bit randomly generated, and a variable c to store the fake commitment and procedures `init` and `simulate`. We start by defining the interface for our simulator which consists of two procedures– one to initialize state and a second to simulate the execution of a protocol given a statement s. The procedure `init` initializes the simulator with the public statement s_2. Then the procedure `simulate` returns a tuple containing the commitment, challenge, and response. It starts using local variables named resp and s' of type commitment, it then picks the challenge bit randomly from the sampling and a value from Z_p. In the case that the challenge bit is 1 it returns c^2/s. In our easycrypt implementation, since we are using the numbers ZModP, division wasnt available so we have multiplied the square of c with the inverse of s. The inverse of s was calculated and stored in another variable (of type commitment as seen above but really commitment is of type ZmodP so s' is of type ZmodP) s'. We then use this s' to compute the response when the challenge bit was 1. Now in the case of the challenge bit b being 0, we compute the response as the square of c. These calculations follow directly from our pen and paper proof. As mentioned before, the simulator generates outputs in the form of a tuple containing $(resp, b, c)$ that are distributed identically to the real protocol outputs despite lacking a witness x. By matching this format we get one step closer to mimic the indistinguishability of the output structure. Since we are trying to prove indistinguishability of protocols that contains samplings from uniform distributions, the transcript generated is never going to be identical, thus we need a tool that reasons about program equivalence in the presence of probabilistic outcomes. We use probabilistic relational Hoare logic (pRHL) to encode indistinguishability between our runs of the real and ideal scenario or protocol. We can now look at our lemma for zero knowledge and step through its proof one at a time as seen below:

```
lemma zero_knowledge:
  forall (s: statement) (x: witness),
    equiv[ RealTranscript.generate ~ Simulator.simulate :
           s{1} = s{2} ==> res{1} = res{2} ].
proof.
move => s x.
proc; inline*.
wp.
rnd{1}.
rnd{2}; rnd{2}.
wp.
rnd{1}.
wp.
skip.
progress.
apply zmoddistr_ll.
apply zmod_resp_y_squared_c_squared.
apply zmoddistr_ll.
apply zmod_resp.
apply zmoddistr_ll.
case ch0.
trivial.
```

```
admit.
have contrad: b.
admit.
trivial.
apply zmod_resp_y_squared_c.
apply zmoddistr_ll.
admit.
qed.
```

Our lemma states that for all (public) statements s and (private) witness x, our programs produce indistinguishable outputs. Our precondition for equivalence states that both the procedures are given the same public statement s and our post-condition ensures that our outputs, stored in res, are indistinguishable under the precondition given. The **equiv** keyword in EasyCrypt is used to write a pRHL judgement. This lemma proves that the verifier (adversary) cannot distinguish between the real transcript (generated using x) and the simulated transcript (generated without x), that the outputs of our program are statistically indistinguishable. Lastly, the simulator can mimic the real world without knowing x which helps us prove that no information has been leaked during this interaction. Note that our current proof is incomplete due to time constraints. Now we can step through the proof and understand each tactic

1. **move**: We use the tactic move to introduce our universally quantified variables s and x into our context. At this point, we end up with a precondition, our program equivalence statement, and our post-condition.
2. **proc; inline***: We then use proc to unfold all the procedures and inline* to unfold all the procedures within the Simulate and generate. The ; is used to combine or merge two tactics into one. In the end, this gives us two programs labeled with 1 and 2 on the screen.
3. **wp**: Both our programs end with either a simple assignment or an if statement, so we use the tactic wp, which stands for weakest precondition. This consumes any if statements and ordinary assignments and replaces the program's post-condition by the weakest precondition.
4. **rnd1**: At this point, we see that program 1 ends with a random sampling, so we run rnd1 on this program to consume this random sampling. **rnd2; rnd2**: Next, we see that program 2 ends with two random samplings, so we run rnd2 twice to consume these random samplings and we merge this using the ; per Easycrypt format.
5. **wp**: Since we see that program 1 ends with an ordinary assignment, we run wp to consume that.
6. **rnd1**: At this point, we noticed that our program 1 ends with a random sampling, so we run rnd on it.
7. **wp**: We now see that program 2 is empty and program 1 only consists of simple assignments, so we can run the weakest precondition to consume all these assignments.
8. **skip**: Since our goal state is a statement judgment with an empty program, we can use skip to reduce it to the goal whose conclusion is the ambient logic formula precondition $\implies$ post-condition. Our precondition is the original precondition we started with.
9. **Progress**: We now use progress to break down our given goal into smaller goals, and progress will apply tactics like move, split, trivial, etc. until it cannot reduce the current

goal anymore. After this, we ended up with a goal islossless zmod_distr since our program had random samplings.

10. **apply zmoddistr_ll**: For each random sampling, Easycrypt introduces a proof obligation to ensure that the distribution is lossless. This is Easycrypt confirming that the distribution Zmod_distr, which is used for c is lossless meaning that zmod_distr specifies a uniform distribution for Z_p aka all numbers are equally likely. In order to resolve this goal, we wrote an axiom that defines that Zmod_distr is a well-defined and lossless distribution. We repeat applying this axiom whenever this goal shows up.
11. **apply zmod_resp_y_squared_c_squared**: We then have a goal state that contains $y * y = (c2 * c2)/s$ and since we know that this should hold, we write another axiom named zmod_resp_y_squared_c_squared to work with modular arithmetic and make progress.
12. **apply zmod_resp.**: We end up with an equation $y * x1 = c2$ so we write an axiom to make progress.
13. **case ch0**: At this point, we can do a case analysis on the challenge bit. The first subgoal is trivial but the second subgoal is admitted due to time constraints.
14. **have contrad: b**: At this point we have a false in the conclusion and the way we proved this is by proving a contradiction. In eascyrpt we need to define a sub-lemma and prove that that is a contradiction. The first sub-goal generated by this contradiction is admitted but the second is trivial.
15. **apply zmod_resp_y_squared_c**: We now end up with a goal state like $y*y = c2*c2$ so we have written an axiom to prove this and make progress.
16. We have admitted a total of 3 goals due to time constraints.
17. This completes our proof of zero knowledge using the Real-Ideal paradigm.

Thus attempting the proof that the real and simulated transcripts are indistinguishable under the condition that the public statements remain consistent between them. This result guarantees that the verifier learns nothing about the prover private witness x beyond the validity of the statement $s = x \times x$ and it suffices to prove Statistical Zero-Knowledge.

5 Conclusion

This project was more challenging than we previously assumed. Initially, we used integers as our main data type in our implementation. However, we quickly got stuck with how we work with sampling. EasyCrypt represents samples using a special *mu* operator. However, we did not find documentation on how we could work with this operator. So we decided to check how previously published work [4] solved those issues.

We quickly noticed that they used a different data type, *zmod*, which represents the group of integers modulo a p where p is a prime number. This datatype ended up simplifying our workflow because it has an implicit module on every operation and also utilizes many properties that are only valid when our module is a prime number. After seeing that, we decided to change the main data type, and surprisingly, samples from *zmod* used lambdas, which are simpler to work with. This made our correctness proof a bit straightforward and succinct.

The next challenge we faced was related to the Soundness statement. At this first moment, we were not sure about what was the exact definition of a malicious prover and how we could talk about every possible malicious prover. After some time we decided again to look for insight in [4], but we quickly noticed that our paths already have diverged. In that work, they use the concept of rewindability to prove all the other properties besides completeness.

However, this is not something that EasyCrypt can do natively, so they abused axioms to make this machinery work, weakening their formal proofs. We decided to try to finish our proof using the same strategy as used in our informal proofs, but we did not manage to fully implement it. A similar situation happened with the zero-knowledge proof, but we attempted to use the Real/Ideal paradigm without using rewindability.

Another challenge we faced was implementing zero knowledge. Even though the concept of indistinguishability is intuitively not as hard to grasp as we originally thought, the details of implementation in Easycrypt using probabilistic hoare logic are not trivial. We had to follow closely with the pen and paper proof to model our simulator and make sure the types aligned. Coming up with pre and post-conditions was a challenge but it made sense to us that the given interaction will hold if and only if the real and ideal scenarios have been given the same initial statement. So this was made as a precondition. For the post-conditions, since we had to reason about indistinguishability between the generated transcripts we chose to ensure that the simulators output is indistinguishable from the real provers using the EasyCrypt's keyword **res**. Coming to the proof, we got stuck on the modular arithmetic segments so we decided to use axioms to make progress. One thing to improve on here would be to rewrite all the axioms as lemmas and prove each lemma before applying it.

Currently, our framework consists of a prover and verifier type and its corresponding interface implementation and related games to model the interactive proof of quadratic residues. The quadratic residue problem falls under a class of protocols called Σ-Protocols, our framework can be generalized to other Σ-protocols – with obvious changes to the prover and verifier due to the different procedures– but beyond that, we will have to spend some more time to think of how to implement those non Σ-protocols and how their implementation will differ from those with sigma protocols.

References

1 Boaz Barak. Lecture 15: Zero knowledge proofs. *Computer Science Department–Princeton University*, page 1, 2007.

2 Henry Corrigan-Gibbs. Lecture 4: Zero knowledge, 2019. URL: https://crypto.stanford.edu/cs355/19sp/lec4.pdf.

3 Uriel Fiege, Amos Fiat, and Adi Shamir. Zero knowledge proofs of identity. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 210–217, 1987.

4 Denis Firsov and Dominique Unruh. Zero-knowledge in EasyCrypt. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, pages 1–16. IEEE. URL: https://ieeexplore.ieee.org/document/10221929/, doi:10.1109/CSF57540.2023.00015.

5 Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in coq. Publication info: Published elsewhere. ACM Transactions on Programming Languages and Systems (TOPLAS). URL: https://eprint.iacr.org/2021/397.

6 Dima Kogan. Lecture 6: Sigma protocols, secret sharing, 2019. URL: https://crypto.stanford.edu/cs355/19sp/lec6.pdf.

7 Vitor Pereira. Easycrypt - a (brief) tutorial, 2023. URL: https://fm.csl.sri.com/SSFT23/easycrypt-tutorial.pdf.