**CS355: Applied Zero Knowledge Proofs** **Spring 2025**

# Programming Project

Due: 11:59 pm on Monday, May 19, 2025, on Gradescope

## 1 Introduction

In this project, you will build an *anonymous* bulletin board where authorized members can post messages, but their post will not reveal *who* posted. More specifically, when you post a *signed* message, you can select an *anonymity set*, i.e., a set of members including yourself that *may* have signed the message. Anyone can verify that indeed the message was signed by one of the members in the set but not know exactly who. You will implement the above functionality using a cool cryptographic primitive called a *Ring Signature*.

Note that this is a programming project and involves you writing real code that you can run on your own laptop! Please ask the TAs for any technical assistance with setting up the starter code and programming in the Rust programming language.

## 2 Ring Signatures

A ring signature [7, 4] is a cryptographic primitive that enables a user to sign a message with respect to a ring[1] of possible signers (including the actual signing user) such that the signature can be verified given only the message and the set of public keys of the ring of signers—it does not reveal the identity of the actual signing user. In such a setting, every user generates a public/secret keypair and publishes their public key. We treat a public key as the identity of that user[2].

We present the syntax of a ring signatures below. We informally state completeness, anonymity, and unforgeability which we encourage you to read but is not crucial for the project.

**Definition 1 (Ring signature [4])** *A ring signature scheme is a triple of PPT algorithms* ($\mathsf{RingGen}$, $\mathsf{RingSign}$, $\mathsf{RingVrfy}$) *such that,*

- $\mathsf{RingGen}(1^\lambda) \to (pk, sk)$. *Generates a keypair for a user in the system, where $\lambda$ is the security parameter, $pk$ is the user's public key, $sk$ is their secret key.*
- $\mathsf{RingSign}(pk, sk, R, m) \to \sigma$. *Signs the message $m$ with respect to the ring of public keys $R$, where $R = \{pk_1, \ldots, pk_n\}$ is a set of public keys containing $pk$.*
- $\mathsf{RingVrfy}(R, m, \sigma) \to 0/1$. *Verifies a purported signature $\sigma$ on a message $m$ with respect to the ring of public keys $R$.*

[1] The "ring" in ring signatures is not related to the mathematical object called rings. It is simply a set.

[2] We assume there exists some Public Key Infrastructure (PKI) that maintains the database of every user's public key.

See also Figure 1. We say that the scheme is **complete** if for every set $\{(pk_i, sk_i)\}_{i\in[n]}$ output by $n$ invocations of $\mathsf{RingGen}(1^\lambda)$, every $j \in [n]$, every $R \subseteq \{pk_1, \ldots, pk_n\}$ that contains $pk_j$, and every message $m$, we have $\mathsf{RingVrfy}(R, m, \mathsf{RingSign}(pk_j, sk_j, R, m)) = 1$. Informally, we say the scheme is **anonymous** if the signature hides which user's secret key was used to sign the message. Informally, we say that scheme is **unforgeable** if an adversary cannot generate $(R, m, \sigma)$ such that $\mathsf{RingVrfy}(R, m, \sigma) = 1$ unless (1) one of the public keys in $R$ was generated by the adversary, or (2) a valid signature of $m$ has been generated with respect to $R$ before. Students who are interested in the complete definition can refer to [4] for the details.

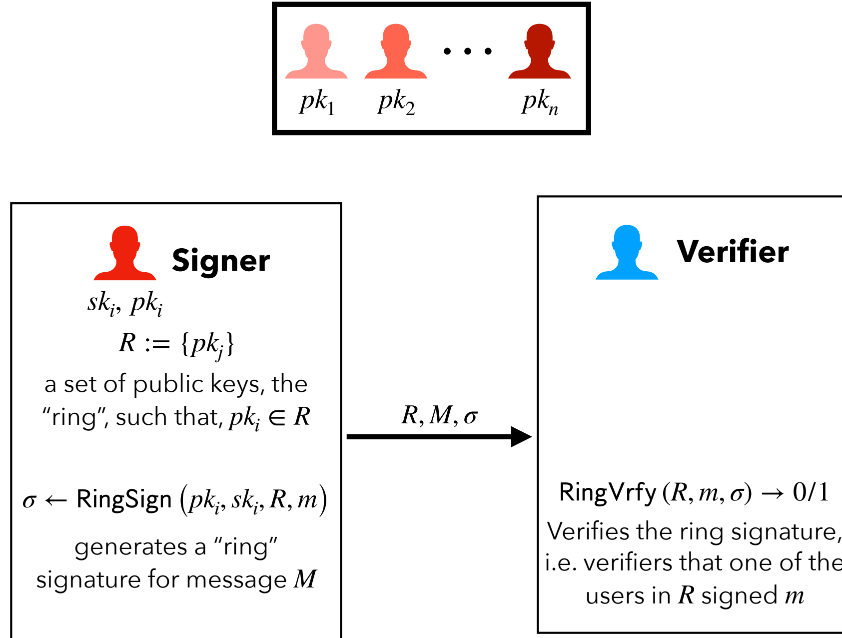


Figure 1: Ring signature syntax.

## 2.1 Building Ring Signatures Using SNARKs

In this project we will build an efficient ring signature scheme using a SNARK. Our starting point is some secure digital signature scheme, such as RSA signatures. We briefly recall the syntax of a signature scheme [5].

**Definition 2 (Digital Signature)** *A signature scheme is a triple of PPT algorithms* ($\mathsf{Gen}$, $\mathsf{Sign}$, $\mathsf{Vrfy}$) *such that,*

- $\mathsf{Gen}(1^\lambda) \rightarrow (pk, sk)$ *(key generation). Outputs a public verification key $pk$ and secret signing key $sk$.*
- $\mathsf{Sign}(sk, m) \rightarrow \rho$ *(signing). Generates signature $\rho$ on the message $m$ using secret key $sk$.*
- $\mathsf{Vrfy}(pk, m, \rho) \rightarrow 0/1$ *(verification). Verifies a purported signature $\rho$ on a message $m$ with respect to public key $pk$.*

We assume that the signature scheme is *existentially unforgeable* which informally means that a PPT adversary cannot forge valid signatures for any new messages (i.e., messages it has not seen signatures for before).

**Constructing a ring signature.** Let Sig be an unforgeable digital signature scheme.

- Algorithm $\mathsf{RingGen}(1^\lambda)$ simply runs $\mathsf{Sig.Gen}(1^\lambda)$ and outputs the resulting $pk$ and $sk$.
- The ring signing algorithm $\mathsf{RingSign}(pk, sk, R, m) \to \sigma$ uses the SNARK prover to generate a ZK proof $\pi$ that it knows $(pk, \rho)$ such that (1) $pk \in R$ and (2) $\mathsf{Vrfy}(pk, m, \rho) = 1$. Algorithm $\mathsf{RingSign}(pk, sk, R, m)$ outputs this proof $\pi$ as the ring signature.
- Anyone can verify the ring signature using the SNARK verification algorithm. The public statement is the pair $(R, m)$.

More precisely, your task is to build a SNARK for the following relation

$$\mathcal{R} := \Big\{ \big((R, m),\ (pk, \rho)\big) \ \Big|\ pk \in R \ \text{ and } \ \mathsf{Sig.Vrfy}(pk, m, \rho) = 1 \Big\}.$$

Let (P, V) be a SNARK prover and verifier for $\mathcal{R}$. Fig. 2 and fig. 3 show the construction of the ring signature and verifier respectively.

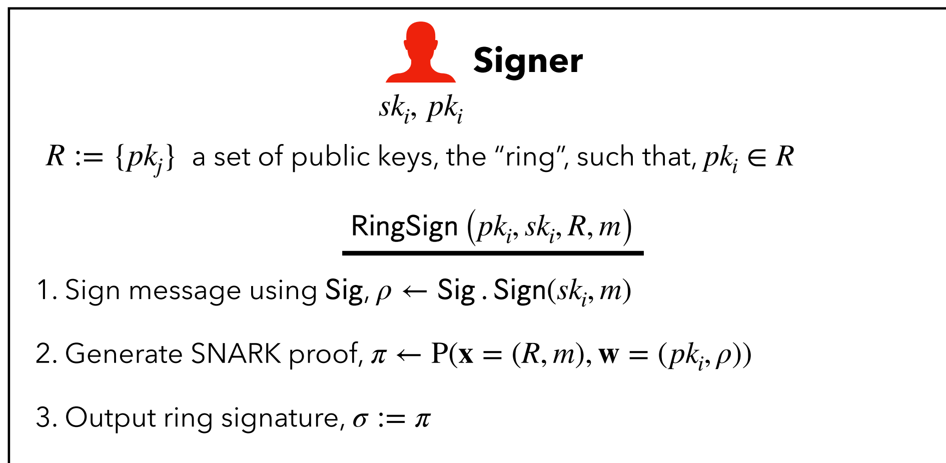


Figure 2: A ring signature signer using a SNARK prover.

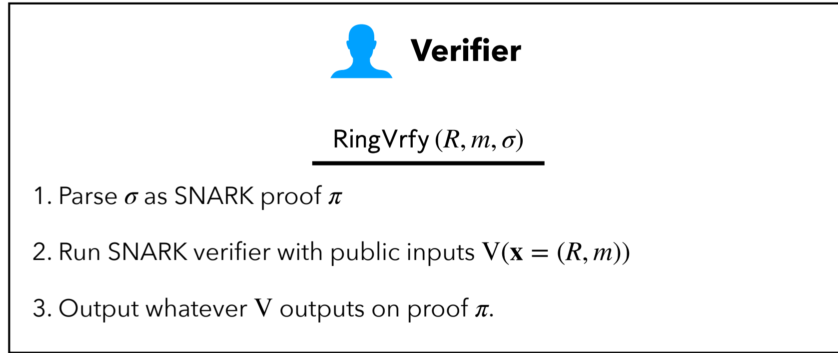


Figure 3: A ring signature verifier using a SNARK verifier.

In this project, we will build the SNARK for $\mathcal{R}$ using an open-source library called Plonky2 [1]. Plonky2 uses PLONK and FRI (covered in a later lecture) to implement SNARKs. Recall from class that PLONK allows us to build SNARKs for arithmetic circuits. Essentially, Plonky2 enables developers to build circuits after which the library handles proof generation and verification.

**Your task is to implement a SNARK for $\mathcal{R}$ in Plonky2!**

First, we need a concrete and signature scheme to instantiate $\mathsf{Sig}$.

## 2.2 RSA Signatures

The RSA signature scheme is a widely-used signature scheme based on the hardness of the RSA assumption [6]. Below, we present how to implement the algorithms. We will fix the public exponent, $e = 2^{16} + 1 = 65537$. We assume we have a hash function that maps strings of arbitrary length to 256 bits, $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^{256}$ (e.g. SHA256 or Poseidon).

**Definition 3 (RSA signature scheme)** *The RSA signature scheme consists of the following three PPT algorithms.*

- $\mathsf{Gen}(1^\lambda) \to (pk, sk)$.
  1. *Randomly generate two distinct large (e.g. 1024 bits) prime numbers, $p, q$.*
  2. *Compute $N = pq$. Compute Euler's totient function*[3] *on $N$, i.e., $\phi(N) = (p-1)(q-1)$.*
  3. *Compute $d = e^{-1} \mod \phi(N)$.*
  4. *Output $(pk, sk) = (N, d)$.*
- $\mathsf{Sign}(sk = d, m) \to \rho$.
  1. *Hash the message,* $\mathsf{hash} \leftarrow \mathsf{H}(m)$.

[3] https://en.wikipedia.org/wiki/Euler%27s_totient_function

2. *Pad the hash using the PKCS1 v1.5 schema*[4]*,*

$$\mathsf{PaddedHash} := 0\text{x}00 \parallel 0\text{x}01 \parallel 0\text{xFF} \parallel \ldots \parallel 0\text{xFF} \parallel 0\text{x}00 \parallel \mathsf{hash}$$

   *such that the resultant length is equal to the size of* $N$*. Here,* $\parallel$ *refers to concatenation of bits and* 0x00*,* 0x01*,* 0xFF *are bytes in hexadecimal. We treat the padded hash value as an element of* $\mathbb{Z}_N$*.*

3. *Output the signature,* $\rho := (\mathsf{PaddedHash})^d \pmod N$*.*

- $\mathsf{Vrfy}(pk = N, m, \rho)$.
  1. *Compute padded hash of* $m$ *as defined above,* $\mathsf{PaddedHash}_m$*.*
  2. *Output* $\rho^e \stackrel{?}{=} \mathsf{PaddedHash}_m \pmod N$

Note that the only algorithm you need to implement inside the SNARK is Vrfy. We already provide you with an implementation of Gen and Sign.

# 3 Building SNARKs using Plonky2 library

Plonky2 [1] is an open-source cryptographic library for SNARKs built by Polygon Zero [2]. It is built in Rust, a popular programming language with built-in memory safety, strong and static types, and an excellent static analyzer. We recommend going through Rust by Example [3] for a crash course on Rust. We will also cover the basics and essentially everything you would need to know about Rust to complete the assignment in section (May 9) so please attend!

As mentioned, your task is to build the ring signature verification circuit in Plonky2. In this section, we provide a brief overview on how to build circuits using Plonky2.

## 3.1 Circuit Builder

Circuits are built in Plonky2 using the `CircuitBuilder` class. In the starter code, we already configure `CircuitBuilder`. Using the builder, you construct the circuit by adding input wires, constructing gates and specifying the output condition. Each wire in the circuit is known as a "target" and the circuit is built by operating on these "targets".

## 3.2 Big Integers: `BigUint`

From a prior programming class, you might be familiar with `uint` or the unsigned integers data type. In this project, we will use a custom data type, `BigUint`, the big bad cousin of `uint`. `BigUint` supports arithmetic over *arbitrarily* large (unsigned) integers! For security, we will need to work with very large integers (roughly 2048 bit) which is much larger than any native data type can hold. Plonky2 implements `BigUint` using a vector of unsigned 32-bit integers.

## 3.3 Targets

Each wire in the circuit is denoted as a "target" in Plonky2. Your goal is to create and connect targets using appropriate builder functions (that represent arithmetic gates).

[4]The schema can be found here for reference https://datatracker.ietf.org/doc/html/rfc8017#section-9.2. We ignore encoding the algorithm ID for this project.

### 3.4 Input Wires

We can specify the input wires for the circuit using two functions,

1. `builder.add_virtual_public_biguint_target(num_limbs)`: Add a *public statement* input wire for a BigUint of size `num_limbs` to the circuit. For this project, `num_limbs = 64` so that we can hold 2048-bit unsigned integers. Ouputs a `BigUintTarget`, a target (wire) holding a `BigUint` value.

2. `builder.add_virtual_biguint_target(num_limbs)`: Add a *witness* input wire for a BigUint of size `num_limbs` to the circuit. Ouputs a `BigUintTarget`.

### 3.5 Constants

We can create wire values that hold a constant value using `builder.constant(F::val)`. For this project, we will only need the values 0 and 1, which can be created using `builder.zero()` and `builder.one()`. Both output a `Target` (not a big integer or boolean target).

### 3.6 Equality gate

We can check if two big integer targets (`BigUintTarget`) are equal in the circuit using `let output = builder.eq_biguint(biguint_target1, biguint_target2)`. It outputs a `BoolTarget` which is a boolean target (wire holding a boolean value). The output `BoolTarget` holds the value 1 if they are equal and 0 otherwise. Later, we will need to convert a `BoolTarget` to a "normal" `Target` which can be done using `output.target`.

### 3.7 Or gate

Given two boolean targets (`BoolTarget`), we can compute the logical OR of their values using `builder.or(bool_target1, bool_target2)`. It outputs a `BoolTarget` similar to the equality gate.

### 3.8 And gate

Given two boolean targets (`BoolTarget`), we can compute the logical AND of their values using `builder.and(bool_target1, bool_target2)`. It outputs a `BoolTarget` similar to the or gate.

### 3.9 Add gate

We can add `BigUint`'s in the circuit using `builder.add_biguint(biguint_target1, biguint_target2)`. Outputs a `BigUintTarget` representing the sum.

### 3.10 Multiply gate

We can multiply `BigUint`'s in the circuit using `builder.mul_biguint(biguint_target1, biguint_target2)`. Outputs a `BigUintTarget` representing the product.

### 3.11 Modulo gate

We can mod `BigUint`'s in the circuit using `builder.rem_biguint(biguint_target1, biguint_modulus_target)`. Outputs a `BigUintTarget` representing the value of `biguint_target1` mod the value of `biguint_modulus_target`.

### 3.12 Output condition

Finally, Plonky2 defines the output condition using `builder.connect(target1, target2)`. It takes in two targets of type `Target` and checks that they are equal. `builder.connect` can be called multiple times to specify what the value of multiple output wires should be. Every connect constraint should be satisfied in order for the circuit to evaluate to true. Think of this as checking the correctness of each output wire and taking a logical AND of all the constraints.

### 3.13 Setting the Circuit Inputs in Proof Generation

Once you've built the circuit, the prover must prove that it knows some witness to the circuit. Plonky2 abstracts the proof generation procedure and you, the developer, must only set what the values to each of the input wires will be. This is done during the proof generation step using the `PartialWitness` class. It is "partial" since it includes both the public statement and witness inputs.

```
let mut pw = PartialWitness::new();
```

We call `pw.set_biguint_target(&circuit_target, biguint_val)` to set the value of `circuit_target` to a `BigUint` value, `biguint_val`.

Once the partial witness has been built by setting all the input wires (both the public statement and witness), you can generate the proof using

```
let proof = circuit.circuit.prove(pw)?;
```

and verify the proof using

```
circuit.circuit.verify(proof);
```

Please note that you do not need to call these functions yourself. You only need to build the circuit and set the input values. We describe the starter code and your tasks explicitly in Section 4.

## 4 Deliverables and Submission Requirements

Clone the following GitHub repository to get started.[5]

git clone https://github.com/entrohpy/CS355-project-ring-signature

You will only need to edit and submit the file, `./src/gadgets/rsa.rs`. You will need to complete the following functions,

[5]Please reach out to the TAs on Ed or OH for any assistance!

- `create_ring_circuit`: this function builds a circuit that captures the ring signature relation $\mathcal{R}$.

  Tip: You might be tempted to write code that directly checks the relation, but remember you are building a circuit for the relation, which means that you need to code up wires and gates in the circuit that check the relation $\mathcal{R}$.

- `create_ring_proof`: Uses the built circuit to generate the proof. Your task is to connect the prover's inputs (the public statement and witness values) to the appropriate input wires of the circuit.

- `compute_padded_hash`: Given a hash of the message as input, pad it using the PKCS1 v1.5 schema. Please refer to the RSA Sign algorithm in Definition 3. This is needed for security!

- `pow_65537`: Compute the `BigUintTarget` raised to the fixed power 65537 modulo some input modulus target. That is, your goal is to compute $z^{65537} \pmod{N}$ inside the circuit.

### 4.1 Running your code

Ready to test your circuit implementation? Follow the following steps to build and run your SNARK.

1. **Generate RSA keypair.**

   ```
   cargo run --release --example keygen
   ```

   Outputs two files, `key.json` and `key.pub.json` which contain your RSA secret signing and public verification key respectively.

2. **Compile prover and verifier circuits.**

   ```
   cargo run --release --example compile
   ```

   Compiles the prover and verifier circuits into `circuit_prover.json` and `circuit_verifier.json`. (Observe that the prover circuit size is rather large but the verifier circuit is significantly smaller.)

3. **Generate proof.**

   ```
   cargo run --release --example prove <PATH_TO_PUBLIC_INPUT>
   <PATH_TO_PROVER_CIRCUIT> <PATH_TO_PUB_KEY> <PATH_TO_SEC_KEY>
   ```

   Use the generated prover circuit to generate a SNARK proof saved in `proof.json`. The public input file consists of the ring $R$, i.e., the set of public keys to sign with, and the message $m$. Please see Section 4.3 on how to produce this file using our live demo.

   Proof generation might take a few minutes, possibly depending on how efficient your implementation is. When it is done, it will output a file, `proof.json`.

4. **Verify proof.**

```
cargo run --release --example verify <PATH_TO_VERIFIER_CIRCUIT>
    <PATH_TO_PROOF> <PATH_TO_PUBLIC_INPUT>
```

Outputs success if the proof is valid.

### 4.2 Tests

We have also provided you with some basic tests to verify your implementation. You can run all tests using `cargo test --release`. You may run specific tests by appending the name of the test to the end of the command.

### 4.3 Bulletin Board Demo

You've written your first SNARK, great! We're hosting a live bulletin board at

https://web.stanford.edu/class/cs355/

where you can post messages anonymously using the ring signature system that you implemented!

You will need to authenticate with your Stanford credentials to access the page. Once you're logged in, (1) please sign up in the top right corner by uploading your public key, and (2) upload your verifier circuit. Please use your real name while signing up.

You are now ready to post *anonymously*! Select the users in your ring, type in your message and download the proof input. Use your SNARK prover to generate a proof with respect to the downloaded public input and the compiled prover circuit and upload it on the page. Next, you can select your own circuit or someone else's in the class a circuit to verify it against and post your message[6]!

### 4.4 Submission on Gradescope

You will only submit your completed `./src/gadgets/rsa.rs` file to Gradescope, which will be graded. Your activity on the bulletin board will not be graded, but you are strongly encouraged to use the bulletin board to check that your proof system is working as expected. You should not edit any file other than `./src/gadgets/rsa.rs`.

## References


[1] 0xpolygonzero/plonky2. https://github.com/0xPolygonZero/plonky2.

[2] Polygon zero. https://polygon.technology/.

[3] Rust by Example. https://doc.rust-lang.org/rust-by-example/.

[4] Adam Bender, Jonathan Katz, and Ruggero Morselli. Ring signatures: Stronger definitions, and constructions without random oracles. Cryptology ePrint Archive, Paper 2005/304, 2005.


[6]Ensure that your message adheres to the Fundamental Standard. See https://communitystandards.stanford.edu/policies-guidance/fundamental-standard/fundamental-standard-may-1-2023.

[5] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. 2015. https://crypto.stanford.edu/~dabo/cryptobook/draft_0_2.pdf.

[6] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[7] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 552–565, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.