Assignment 2: Having Fun with Risc0

In this assignment, you will familiarize yourself with zkVMs using Risc0, which unfortunately means you will have to use Rust. Lucky for all of us, however, Rust is one of the best documented programming languages there is so it shouldn't take you long to get going (to give you a frame of reference I had close to zero experience in Rust while I was creating this assignment).

Installation

For this assignment, we will use Risc0 which is written in Rust. So, you will have to setup both Rust and Risc0.

To setup Rust, go to the installation instructions on the website.

To setup Risc0, go to the installation instructions on the website.

For the documentation, feel free to visit Rust's and Risc0's websites again.

WARNING: For the sake of everyone's sanity, do not use Windows directly. If you have Windows, run this in WSL .

Risc0 Overview

To greatly simplify how Risc0 works, Risc0 is a zkVM framework that has two environments: *host* and *guest.* A host is the place of actual application where everything is done in the clear. A guest on the other hand is Risc0's so called *zero-knowledge* environment where validity statements as well as verification processes reside. The idea is as the host you do some computation, feed the relevant parts to the guest and guest can both define what is a valid computation for your application as well as check the validity. Before starting the assignment, I strongly recommend checking the Hello World example of Risc0.

To prevent any kind of structural problems with your assignments I have given you the backbone of each assignment with relevant dependencies and marked places you must write your own code. This was done to prevent you with dealing with Risc0 specific syntax. However, I still suggest looking at the code (to both get familiar with it in the case you want to use Risc0 for your term project and see if I missed anything 😊) . In both tasks the common structure is as follows:

- core/ folder hosts common code that can be used by host/guest applications.
- methods/ defines the behavior of guest for verifying validity of computation
- src/ is the place for your actual application

For both tasks, if you want to run the prover you run the following:

```
cargo run –bin prove
```

Similarly, to run the verifier you can run:

```
cargo run –bin verify
```

For communication between the host and guest, the prove method outputs a *receipt* (receipt.bin by default). The receipt contains both public parts that can be used by anyone and private parts that can only be accessible during the proving named Journal and PrivateInput respectively. As part of the assignment, you will populate these sections correctly with relevant information. This simplified description on Risc0 is to give you an intuition, I will be marking task specific files to change for each part.

In this assignment, you will write two applications that generate and validate a proof without explicitly using any NIZKs you have seen so far. The first task is aimed to get you familiar with Risc0 itself whereas the second one is to make you learn one of the most common uses of zero-knowledge proofs: proof of Merkle tree membership.

Part A: Signatures, signatures, and more signatures

Your first task (that consists of two mini tasks) is to familiarize yourself with how Risc0 works through two signature examples.

WARNING: To make everything simple, we use the default ECDSA implementation of Rust. However, DO NOT, and I repeat DO NOT use this in a real application as it is, it does not have a proper security analysis.

A.1: "Ring" Signature
A ring signature let's Alice sign a message, but instead of revealing it came from here, she can prove it came from one of a set (called a Ring) of people. E.g., someone could write a complaint about working conditions and sign it, but then include all employee's public keys as part of the ring. These originally came from a paper "How to leak a secret" and required specialized cryptography. Today you are going to build one out of a normal signature scheme and a ZK proof.

You will be making a dummy version of a ring signature by proving a signature you provide indeed verifies under one of 2 public keys. You can technically reveal all the verification keys and individually check the signature but this breaks the anonymity we associate ring signatures with. Instead, you will first create two pairs of ECDSA key, sign the message

“Your account balance is XX.” with one of those keys and prove that the generated signature is correctly generated for one of the keys. You should replace XX with any arbitrary number that makes sense in that context. To prevent linear verification process, you can pass the index of the key as part of the secret information you prove.

Here is the list of things to do:

- core/src/lib.rs: You need to define private/public values that will be added to receipt.bin
- methods/guest/src/main.rs: You need to define the verification procedure that will be required to check if the provided signature is correctly generated and verifies under one of those keys.
- src/bin/prove.rs: This is the definition for your prover i.e. host application. You should generate the keys/signature here.
- src/bin/verify.rs: This is the definition for your verifier. It will check if the proof given in receipt.bin is correct and consistent.

A.2: Verifying a Substring in a Signature

The second task in this part is to verify validity of a substring in a given message. A simple use case scenario for this one to prove that you have a sufficient amount of money using a signed statement from a bank without revealing your actual balance (the message above makes sense now doesn’t it?)

You are going to replicate a dummy version of this. Using the same message as above your task is to prove you have a non-negative balance without revealing the message. The easiest way to do this to prove that a) the signature verifies and b) the message for the signature contains a non-zero balance.

Here is the list of things to do:

- core/src/lib.rs: You need to define private/public values that will be added to receipt.bin
- methods/guest/src/main.rs: You need to define the verification procedure that will be required to check if the provided signature is correctly generated and contains a non-negative balance.
- src/bin/prove.rs: This is the definition for your prover i.e. host application. You should generate the key/signature here.
- src/bin/verify.rs: This is the definition for your verifier. It will check if the proof given in receipt.bin is correct and consistent.

In addition to receipt.bin, both provers in this part also output signature.bin that contains your generated signature. Verifier then takes the signature which you should use to verify the consistency of the signature in public part of receipt.bin.

Part B: Proof of a Valid Birthday

Your second task is a bit more involved. Give a list of commitments to birthdays, you will prove that you can open one of the commitments and that that birthday is at least 18 years in the past.
. You can technically just reveal the birthday in the open but where is the fun in that? Instead, to keep the anonymity, you will generate a Merkle tree from a list of birthdays and 1) prove that the committed birthday is a member of that Merkle tree 2) the birthdate is over 18 years old.

We are using Merkle tree definition given by one of the Risc0 examples. However, since the example utilizes indexing as the search, they define an oracle on top of the Merkle tree to give fast access to leaves. DO NOT use those methods. Instead, you will write two member functions that are common for Merkle membership proofs: 1) Finding the path from a leaf to the root and 2) Given a root, a leaf and a path, verify that the path from the leaf corresponds to the given Merkle tree root. To simplify dealing with dates, I also defined a SimpleDate struct that has day, month, year as members, you can check it at birthday.rs in core/ folder. Why did I do that? Because to build a Merkle tree the type of the leaves should be hashable and none of the date types were liked by the given Merkle tree definition. If you can get it to work, you are fine to do so just let me know so we can improve this for the next time.

Here is the list of things to do:

- core/src/merkle.rs: You need to define two member functions to 1) Find a path in a Merkle tree for a specific leaf 2) Given the root, a leaf and a path should verify the leaf with the path reaches the root
- core/src/lib.rs: You need to define private/public values that will be added to receipt.bin

- methods/guest/src/main.rs: You need to define the verification procedure that will be required to show your birthday exists in a tree of birthdays and your age is over 18.
- src/bin/prove.rs: This is the definition for your prover i.e. host application. You should build a Merkle tree of dates that also includes your birthday.
- src/bin/verify.rs: This is the definition for your verifier. It will check if the proof given in receipt.bin is correct and consistent.

DISCLAIMER: When I say "your birthday", any specific date should suffice. You do not need to use your actual birthday, otherwise it would be antithetical for you to reveal the date when the reason we go through all this trouble is not to disclose anything 😊.

In addition to receipt.bin, the prover also outputs merkle_root.bin that contains the root of your generated Merkle tree. Verifier then takes the root which you should use to verify the consistency of the root in public part of receipt.bin.

If you have any questions, feel free to email me at dgur1@umd.edu and put [ZK Assignment] on the subject line.

And always follow the golden rule: Don't do what I have done, do whatever I tell you to do.

Note, for this assignment, it is perfectly acceptable to use ChatGPT , CoPilot, or the like, to write some of your code. You likely will not be able to get it to write code for Risc0, but if you need to know how to get Rust going, it might be beneficial.