Privacy-preserving Credentials with zkSNARKs

An obvious privacy limitation with current technology doesn't even allow us to verify our age without revealing our passports and other sensitive data. Let's imagine the situation: Alice (Creditor) want's to prove to Bob (Bank) that she's eligible to get credit. She has to reveal her's id and whole bank history to Bob to prove that she is over specific age and proof that she's eligible for credit.

0.1 Assumptions:

- We must allow users to generate proofs on the third-party issued credentials.
- The verifier must verify that the statement is true concerning the third party-issued credential without knowing the actual credential value.
- The user should have the option to disclose credential values if necessary selectively.
- The prover can make any statement regarding its issued credentials and create proof which asserts that the statement is valid.
- The verifier is an entity that requires the prover to prove the validity of a specific statement.

This point allows us to integrate ZKP into a fully decentralized architecture.

We would change this by generating proof on Alice's side that she is satisfied with specific parameters without revealing sensitive data.

I think the magic of zkSNARKs lets you do this!

With a zkSNARK, you can "prove" that you have some secrets `age` and `amount` (i.e., `R1` and `R2`) that satisfy some programmable condition (i.e., `SHA(R1)=H1` and `SHA(R2)=H2`, based on public inputs (H1, H2, and X), without revealing those secrets.

That's pretty safe because if you receive the only preimage for R1, along with instructions in the onion saying ask Bob for a preimage for R2, and here's X and proof, then either:

0.2 Implementation

This reference implementation is built on top of the blueprint (*which is a fork of libsnark*) library: zkSNARKs are based on verifiable computation schemes.

It seems like there are research-level tools out there that make this practical to try out. I've had a go at implementing this using `blueprint`.

0.2.1 Verification

It is not necessary for a verifier to directly interact with the prover to verify proof.

0.2.2 Using it looks like:

1. initial setup of proof/verification keys

```
./age_test keygen > keygen.txt # initial setup
```

2. generate proof using a secret

```
./age_test -m proof -s "$SECRET" -x "$VAL" > proof.txt
```

3. Verify the proof:

```
./age_test -m verify -h "$F" -b "$B" -x "$VAL"
```

4. Verify it doesn't report a valid proof with different inputs:

```
./age_test -m verify -h "$B" -b "$F" -x "$VAL"
```

Some results:

- Everyone has to trust that nobody has kept the original random numbers used to generate it.
- proof/verification key data takes about a minute to generate on a modern laptop.
- generating the proof data for a given R1, X pair takes about 10 seconds
- verifying the proof is quick-ish – it takes 0.1s on my laptop,

The long proof generation time is probably more of a limitation – though you could generate them in advance quickly enough and store them until it would be best if you used them, which would avoid lag being a problem at least.

0.3 In the end

zkSNARKs are still pretty new as a concept, And it was hard to figure how to build it. I'm not familiar enough with zkSNARK theory to be sure I'm not misusing the concept somehow; My proof may not have implemented the approach entirely correctly, and So not a great idea to use this to protect real money today. But it could be a great start for building a private and scalable blockchain.

But still, this seems like it's not all /that/ far from being practical, and if the crypto's not fundamentally broken, it looks like it goes a long way to filling in the most significant privacy hole in blockchain today.