Context

An ECDSA signature verification circuit written in Circom results in an R1CS with approximately 1.5 million constraints, a proving key that is close to a GB large, and a proving time that is non-trivial.

The point of doing an ECDSA signature verification using Circom is to verify that the prover owns a certain address without revealing the address. Although it is easier to check privKey * G == pubKey

, most wallets don't expose the private key of a wallet, therefore we are restricted to checking signatures. Signature verification without revealing the address has applications in, for example, zero-knowledge proof of membership.

(Which is why we want to have a way to prove ownership of an address anonymously)

In this post, I will decribe a method that could improve the efficiency of ECDSA signature verification written in Circom.

I focus on implementation using Circom to avoid ambiguity. However, the method is not completely dependent on Circom. You can swap "Circom" with "zkp", "zk-snark", or other high-level arithmetic circuit languages.

Overview of the method

The essence of the method is, that in order to do as less computations as possible in a circuit, we offload some signature verification calculations from the circuit.

The method

This credit for this technique goes to the answerer ofthis stack exchange post. Although, the method is not formally verified. If this method lacks correctness, soundness, or zero-knowledge-ness, then the entire scheme I describe in this post will not work. That being a possibility, I'm writing this post hoping it to be useful at least in some way, as a source of ideas.

Verify an ECDSA signature without revealing the s

part of the signature leads to reducing the required calculation that needs to be kept private (i.e. needs to be SNARKified)

First, you produce a signature with your private key as usual.

R = k * G

s = k^-1(m + da * r)

The signature = (r, s)

where

- G

: The generator point of the curve

- k

: The signature nonce

- R

: k * G

- Qa

: The public key

- m

: The message to sign

- r

: x-coordinate of R

The signature can be verified by checking the following equality

R = s^{-1} * m * G+ s^{-1}r * Qa

or

s * R = m * G + r * Qa

.

This equation can be perceived as s

being the private key, R

being the generator point, and m * G + r * Qa

being the public key.

Now we can prove the knowledge of s

without revealing s

itself, by generating a signature!

We calculate the signature as follows:

R’ = k’ * R

s’ = k^-1(m’ + s * r')

where

- k’

: The signature nonce

- m’

: The message to sign

- r’

: x-coordinate of R'

We verify the signature (s’, r')

by checking:

- s’ * R’ = m’ * R + r' * Qa’

This equation itself doesn’t reveal Qa

(the public key). So it can be checked publicly, without using Circom.

We also need to check that Qa’ actually comes from

Qa

.

This can be done by checking:

- Qa’ = m * G + r * Qa

Since we don’t want to reveal Qa

(the public key), this equality check is done using Circom.

Moreover, it is required to keep m

a secret. If m

is revealed, Qa

will be recoverable. That is, in zero-knowledge proof of membership

, the public keys that are members of a set are publicly known; someone can just check which public key matches Qa’

, by filling in m

and r

.

Summary and benchmarks

To sum up, the circuit will take Qa

and m

as the private input, and r'

and Qa'

as the public input. I constructed the outline of the circuithere. The circuit is not complete. The purpose of it is to demonstrate the outline.

The benchmarks:

- constraints: \approx

200,000

- proving key size: \approx

134MB

- proving time (witness generation + proving): \approx

15s on a MacBook Pro

Which is a meaningful improvement from the original circuit.

And that is it.

Feedback will be appreciated.