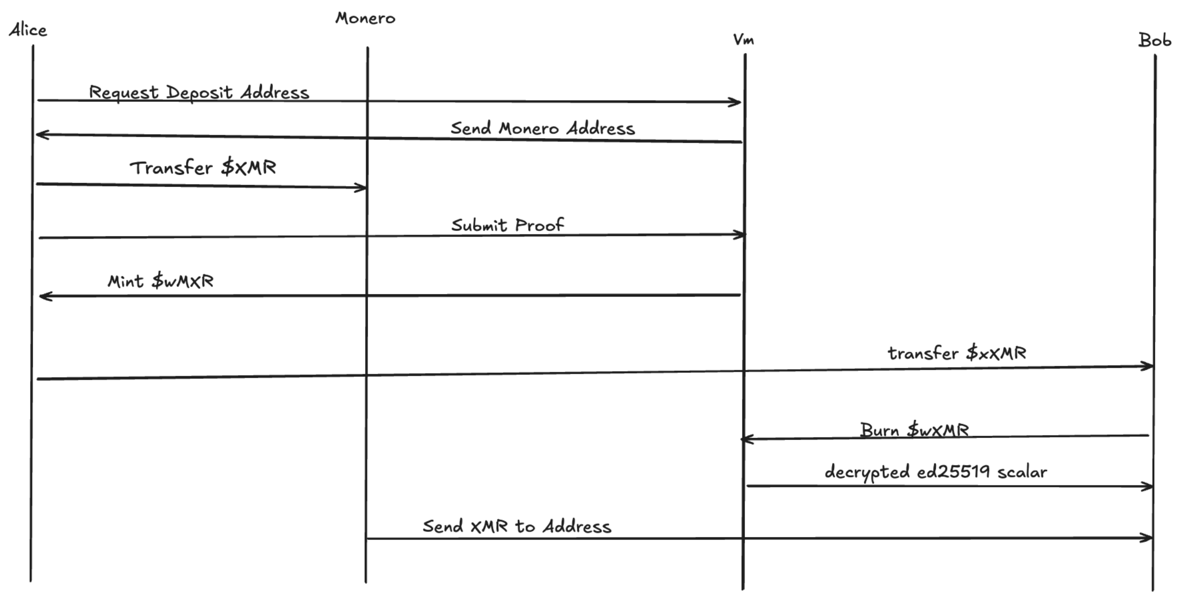
Wrapped XMR Flow

So basically we need two proofs, one for mint (lock $XMR on Monero and print $wXMR on the VM) and one for burn (brun $wXMR on VM and allow user to get the private key (adapter signature format) to send themself the amount of $XMR they just burned.

The initial idea is to create a new stealth address for each deposit. I'm not sure if this is necessary, perhaps if we can find a way to use MPC or something to create a signature with a fixed amount from a vault address, then we wouldn't need to spin up a new address each time. The main UX caveat to this, is that users will only be able to burn in fixed amounts, however they should always be able to withdraw as the amount of mint values must be equal to the amount of $wXMR. An easier version of a musig account also works for simplicity.

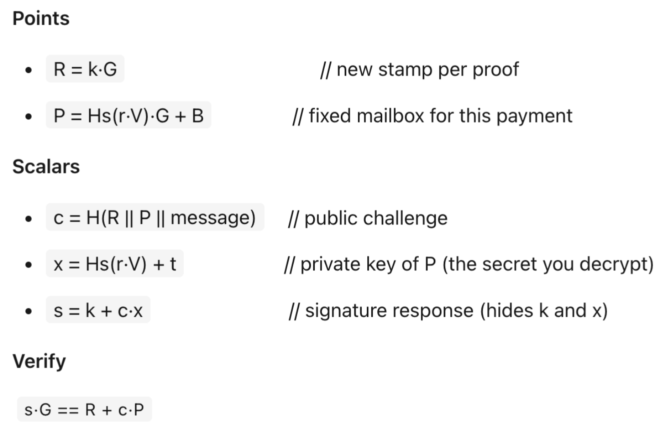


1. Alice requests the creation of a new Monero Account. Basically the idea here is to use Arcium to generate a new ed25519 keypair in order to get the Monero Deposit address. So under the hood what we need to do is derive a public key format from ed25519 (we do some formatting to convert it into a Monero compatible address).
    - t = SHA-512(k) → T = t*G
    - Once we get to T we are able to derive the Monero public address

   So during the initial contract call to the VM, Arcium will generate a random 256bit value (k).. Which will be used to also publicly show T, while enc(k) exists on-chain as a ciphertext.
2. Alice then sends X amount of $XMR to the public address from step 1.
3. To complete the deposit process Alice needs to format a proof of a valid transaction. At this point if we don't want the entire process to be private we don't necessarily need to zero knowledge verification. So for proof generation and verification actually is already handled via the Monero CLI. The default proof is a ~200byte envelope.
    a. Tx-id : 32B

   b. Sig(s,c) : 64Bytes
   c. Tx PubKey (R): 32Bytes

So the tricky part here is constructing s and c.
c = challenge hash = SHA-512(R + P + message) chopped to 252 bits.
Basically there’s a way to generate this proof via Monero cli if you pass in a few fields, but ideally you use zk here for full privacy.



Points

- $R = k \cdot G$ // new stamp per proof
- $P = Hs(r \cdot V) \cdot G + B$ // fixed mailbox for this payment

Scalars

- $c = H(R \| P \| \text{message})$ // public challenge
- $x = Hs(r \cdot V) + t$ // private key of P (the secret you decrypt)
- $s = k + c \cdot x$ // signature response (hides k and x)

Verify

$s \cdot G == R + c \cdot P$

4. After verification we can complete the mint process by minting to an erc-20, spl, or ideally a c-spl. To privacy max, If you mint to an Arcium c-spl ideally you would also verify the amount (via zk) and constrain that this is the same amount that was encrypted client-side via the rescue cipher. Ideally a zk proof would hide the amount as well, so that we could mint a confidential amount of $XMR to a c-spl that would now handle confidentiality on-chain for us.
5. Alice sending to Bob is as simple as a spl/c-spl transfer. User to user.
6. When Bob wants to burn $wXMR for XMR, he would call the burn function, which will verify the amount burnt and return a re-encrypted sealed output (from Arcium) of k, which is used along with public information to construct the private spend key. This is similar to how Monero handles atomic swaps via adapter signatures.
7. Bob can now send XMR on Monero to any address he wants.