Contents

1 Sapling payment scheme

Sapling is a type of transaction which hides both the sender and receiver data, as well as the amount transacted. This means it allows a fully private transaction between two addresses.

Generally, the Sapling payment scheme consists of two ZK proofs - **mint** and **burn**. We use the mint proof to create a new *coin* C, and we use the burn proof to spend a previously minted *coin*.

1.1 Mint proof

`{{#include ../../../../proof/mint.zk}}`

As you can see, the `Mint` proof basically consists of three operations. First one is hashing the *coin* C, and after that, we create *Pedersen commitments*[1] for both the coin's **value** and the coin's **token ID**. On top of the zkas code, we've declared two constant values that we are going to use for multiplication in the commitments.

The `constrain_instance` call can take any of our assigned variables and enforce a *public input*. Public inputs are an array (or vector) of revealed values used by verifiers to verify a zero knowledge proof. In the above case of the Mint proof, since we have five calls to `constrain_instance`, we would also have an array of five elements that represent these public inputs. The array's order **must match** the order of the `constrain_instance` calls since they will be constrained by their index in the array (which is incremented for every call).

In other words, the vector of public inputs could look like this:

```
let public_inputs = vec![
    coin,
    *value_coords.x(),
    *value_coords.y(),
    *token_coords.x(),
    *token_coords.y(),
];
```

And then the Verifier uses these public inputs to verify a given zero knowledge proof.

1.1.1 Coin

During the **Mint** phase we create a new coin C, which is bound to the public key P. The coin C is publicly revealed on the blockchain and added to the Merkle tree.

Let v be the coin's value, t be the token ID, ρ be the unique serial number for the coin, and r_C be a random blinding value. We create a commitment (hash) of these elements and produce the coin C in zero-knowledge:

$$C = H(P, v, t, \rho, r_C)$$

An interesting thing to keep in mind is that this commitment is extensible, so one could fit an arbitrary amount of different attributes inside it.

[1] See section 3: *The Commitment Scheme* of Torben Pryds Pedersen's paper on Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing

1.1.2 Value and token commitments

To have some value v for our coin, we ensure it's greater than zero, and then we can create a Pedersen commitment V where r_V is the blinding factor for the commitment, and G_1 and G_2 are two predefined generators:

$$v > 0$$
$$V = vG_1 + r_V G_2$$

The token ID can be thought of as an attribute we append to our coin so we can have a differentiation of assets we are working with. In practice, this allows us to work with different tokens, using the same zero-knowledge proof circuit. For this token ID, we can also build a Pedersen commitment T where t is the token ID, r_T is the blinding factor, and G_1 and G_2 are predefined generators:

$$T = tG_1 + r_T G_2$$

1.2 Pseudo-code

Knowing this we can extend our pseudo-code and build the before-mentioned public inputs for the circuit:

```
{{#include ../../../../proof/mint.rs:main}}
```

1.3 Burn

```
{{#include ../../../../proof/burn.zk}}
```

The `Burn` proof consists of operations similar to the `Mint` proof, with the addition of a *Merkle root*[2] calculation. In the same manner, we are doing a Poseidon hash instance, we're building Pedersen commitments for the value and token ID, and finally we're doing a public key derivation.

In this case, our vector of public inputs could look like:

```
let public_inputs = vec![
    nullifier,
    *value_coords.x(),
    *value_coords.y(),
    *token_coords.x(),
    *token_coords.y(),
    merkle_root,
    *sig_coords.x(),
    *sig_coords.y(),
];
```

1.3.1 Nullifier

When we spend the coin, we must ensure that the value of the coin cannot be double spent. We call this the *Burn* phase. The process relies on a nullifier N, which we create using the secret key x for the public key P and a unique random serial ρ. Nullifiers are unique per coin and prevent double spending:

$$N = H(x, \rho)$$

1.3.2 Merkle root

We check that the merkle root corresponds to a coin which is in the Merkle tree R

$$C = H(P, v, t, \rho, r_C)$$
$$C \in R$$

[2] Merkle tree on Wikipedia

1.3.3 Value and token commitments

Just like we calculated these for the `Mint` proof, we do the same here:

$$v > 0$$
$$V = vG_1 + r_V G_2$$
$$T = tG_1 + r_T G_2$$

1.4 Public key derivation

We check that the secret key x corresponds to a public key P. Usually, we do public key derivation my multiplying our secret key with a genera tor G, which results in a public key:

$$P = xG$$

1.5 Pseudo-code

Knowing this we can extend our pseudo-code and build the before-mentioned public inputs for the circuit:

```
{{#include ../../../../proof/burn.rs:main}}
```