ECC Posts / How Transactions Between Shielded Addresses Work

How Transactions Between Shielded Addresses Work

Ariel Gabizon | November 29, 2016 | Updated: October 1, 2018

In 'Anatomy of A Zcash Transaction' we gave a general overview of Zcash Transactions. The purpose of this post is to provide a simplified explanation of how *privacy-preserving* transactions work in Zcash, and where exactly Zero-Knowledge proofs come into the picture. In the terminology of that post, we are focusing exclusively here on transactions between shielded addresses (commonly referred to as z-addrs).

To focus on understanding the privacy-preserving aspect, let's put aside everything having to do with reaching consensus using Proof of Work and the blockchain, and focus on a particular node that has obtained the correct list of unspent transaction outputs.

Let's recall first how this list looks on the bitcoin blockchain. Each unspent transaction output (UTXO) can be thought of as an unspent 'note' that is described by the address/public key of its owner and the amount of BTC it contains. For simplicity of presentation, let's assume each such note contains exactly 1 BTC, and there is at most one note per address. Thus, at a given time, a node's database consists of a list of unspent notes, where each note can be described simply by the address of the owner. For example, the database may look like this.

$\mathsf{Note}_1 = (\mathsf{PK}_1), \mathsf{Note}_2 = (\mathsf{PK}_2), \mathsf{Note}_3 = (\mathsf{PK}_3)$

Suppose PK_1 is Alice's address and she wishes to send her 1 BTC note to Bob's address PK_4. She sends a message which essentially says "Move 1 BTC from PK_1 to PK_4" to all nodes. She signs this message with the secret key sk_1 corresponding to PK_1, and this convinces the node she has the right to move money from PK_1. After the node checks the signature, and checks that there is indeed a 1 BTC note with address PK_1, it will update its database accordingly:

$\mathsf{Note}_4 = (\mathsf{PK}_4), \mathsf{Note}_2 = (\mathsf{PK}_2), \mathsf{Note}_3 = (\mathsf{PK}_3)$

Now suppose each note also contains a random 'serial number' (aka unique identifier) r. We will soon see this is helpful for obtaining privacy. Thus, the database may look like this.

$\mathsf{Note}_1 = (\mathsf{PK}_1, r_1), \mathsf{Note}_2 = (\mathsf{PK}_1, r_2), \mathsf{Note}_3 = (\mathsf{PK}_2, r_3)$

A natural first step towards privacy would be to have the node store only "encryptions", or simply hashes, of the notes, rather than the notes themselves.

$\mathsf{H}_1 = \mathbf{HASH}(\mathsf{Note}_1), \mathsf{H}_2 = \mathbf{HASH}(\mathsf{Note}_2), \mathsf{H}_3 = \mathbf{HASH}(\mathsf{Note}_3)$

As a second step to preserve privacy, the node will continue to store the hash of a note *even after it has been spent*. Thus, it is no longer a database of unspent notes, but rather a database of *all notes that ever existed*.

The main question now is how to distinguish, without destroying privacy, between notes that have been spent and notes that haven't. This is where the *nullifier* set comes in. This is a list of hashes of all serial numbers of notes that have been spent. Each node stores the nullifier set, in addition to the set of hashed notes. For example, after Note_2 has been spent, the node's database might look like this.

Hashed notes	Nullifier set
$\mathsf{H}_1 = \mathbf{HASH}(\mathsf{Note}_1)$	$\mathsf{nf}_1 = \mathbf{HASH}(\mathsf{r}_2)$
$\mathsf{H}_2 = \mathbf{HASH}(\mathsf{Note}_2)$	
$\mathsf{H}_3 = \mathbf{HASH}(\mathsf{Note}_3)$	

How a transaction is made

Now suppose Alice owns Note_1 and wishes to send it to Bob, whose public key is PK_4. We will assume for simplicity that Alice and Bob have a private channel between them although this is not actually necessary in Zcash. Basically, Alice will invalidate her note by publishing its nullifier, and at the same time create a new note that is controlled by Bob.

More precisely, she does the following.

1. She randomly chooses a new serial number r_4 and defines the new note $\mathsf{Note}_4 = (\mathsf{PK}_4, r_4)$.
2. She sends Note_4 to Bob privately.
3. She sends the nullifier of Note_1, $\mathsf{nf}_2 = \mathbf{HASH}(\mathsf{r}_1)$ to all nodes.
4. She sends the hash of the new note $\mathsf{H}_4 = \mathbf{HASH}(\mathsf{Note}_4)$ to all nodes.

Now, when a node receives nf_2 and H_4, it will check whether the note corresponding to nf_2 has already been spent, simply by checking if nf_2 already exists in the nullifier set. If it doesn't, the node adds nf_2 to the nullifier set and adds H_4 to the set of hashed notes; thereby validating the transaction between Alice and Bob.

Hashed notes	**Nullifier set**
$\mathsf{H}_1 = \mathbf{HASH}(\mathsf{Note}_1)$	$\mathsf{nf}_1 = \mathbf{HASH}(\mathsf{r}_2)$
$\mathsf{H}_2 = \mathbf{HASH}(\mathsf{Note}_2)$	$\mathsf{nf}_2 = \mathbf{HASH}(\mathsf{r}_1)$
$\mathsf{H}_3 = \mathbf{HASH}(\mathsf{Note}_3)$	
$\mathsf{H}_4 = \mathbf{HASH}(\mathsf{Note}_4)$	

..but wait a second, we checked that Note_1 wasn't spent before.. but we didn't check whether it belongs to Alice. Actually, we didn't check that it was a 'real' note at all, real in the sense that its hash was in the node's table of hashed notes. The simple way to fix this would be for Alice to simply publish Note_1, rather than its hash; but of course, this would undermine the privacy we are trying to achieve.

This is where *Zero-Knowledge proofs* come to the rescue:

In addition to the steps above, Alice will publish a proof-string π convincing the nodes that *whomever published this transaction knows values PK_1, sk_1, and r_1 such that*

1. The hash of the note $\mathsf{Note}_1 = (\mathsf{PK}_1, r_1)$ exists in the set of hashed notes.
2. sk_1 is the private key corresponding to PK_1 (and thus, whomever knows it is the rightful owner of Note_1).
3. The hash of r_1 is nf_2, (and thus, if nf_2 – that we now know is the nullifier of Note_1 – is not currently in the nullifier set, Note_1 still hasn't been spent).

The properties of Zero-Knowledge proofs will ensure no information about PK_1, sk_1, or r_1 are revealed by π.

The main places above where we cheated or omitted details

We emphasize that this has been an oversimplified description, and recommend the protocol spec for full details.

Here are some of the main things that were overlooked:

1. The hashed notes need to be stored not just as a list, but in a Merkle tree. This plays an important role in making the Zero-Knowledge proofs efficient. Moreover, we need to store a *computationally hiding and binding commitment* of the note, rather than just its hash.
2. The nullifier needs to be defined in a slightly more complex way to ensure future privacy of the receiver in relation to the sender.
3. We did not go into details on how to eliminate the need for a private channel between sender and recipient.

Technical
#explainers, privacy, transactions, zksnarks

< Anatomy of A Zcash Transaction | Announcing the Winners of the Open Source Miner Challenge! >

Resources
Careers
Contact
Media Kit
Copyright Policy
Zcash Trademark Policy

Community
Z.cash
Zcash Community
Forums
Community chat

Newsletter signup

Email*

SUBMIT