Zcash Protocol Specification

Sean Bowe — Daira Hopwood

December 17, 2015

1 Introduction

Zcash is an implementation of the *decentralized anonymous payment* (DAP) scheme **Zerocash** with minor adjustments to terminology, functionality and performance. It bridges the existing value transfer scheme used by Bitcoin with an anonymous payment scheme protected by zero-knowledge succinct non-interactive arguments of knowledge (**zk-SNARK**s). Sean: I want to make sure we add citations here for the original paper

2 Concepts

2.1 Integers and Endianness

Abstractly, integers have a signedness (signed or unsigned), and a bit length. The limits are the same as for the usual two's compliment system. All integers in the publicly-visible **Zcash** protocol are encoded in big endian two's compliment.

If unspecified, curve points, field elements, etc., are encoded according to the crypto libraries the **Zcash** implementation uses.

2.2 Cryptographic Functions

CRH is a collision-resistant hash function. In **Zcash** , the *SHA-256 compression* function is used which takes a 512-bit block and produces a 256-bit hash. This is different from the *SHA-256* function, which hashes arbitrary-length strings.

$\mathbf{PRF}_{\mathbf{x}}$ is a pseudo-random function seeded by x. Three *independent* $\mathbf{PRF}_{\mathbf{x}}$ are needed in our scheme: $\mathbf{PRF}^{\mathbf{addr}}_{\mathbf{x}}$, $\mathbf{PRF}^{\mathbf{sn}}_{\mathbf{x}}$, and $\mathbf{PRF}^{\mathbf{pk_i}}_{\mathbf{x}}$. It is required that $\mathbf{PRF}^{\mathbf{sn}}_{\mathbf{x}}$ be collision-resistant in order to prevent a double-spending attack Eli: I don't see how to use a collision to double spend. If anything, a collision in $\mathbf{PRF}^{\mathbf{pk_i}}_{\mathbf{x}}$ seems more usable to double spend Sean: If you could create two ρ such that there is a collision you could spend the same bucket twice. The original paper makes the claim that this must be collision resistant. In **Zcash** , the *SHA-256 compression* function is used to seed all three of these functions. The bits `00`, `01` and `10` are included (respectively) within the blocks that are hashed, ensuring that the functions are independent.

$$\mathsf{a_{pk}} = \mathbf{PRF}^{\mathbf{addr}}_{\mathsf{a_{sk}}}(0) = \mathbf{CRH}\left(\boxed{\text{256 bit } \mathsf{a_{sk}}}\boxed{0}\boxed{0}\boxed{0^{254}}\right)$$

$$\mathsf{sn} = \mathbf{PRF}^{\mathbf{sn}}_{\mathsf{a_{sk}}}(\rho) = \mathbf{CRH}\left(\boxed{\text{256 bit } \mathsf{a_{sk}}}\boxed{0}\boxed{1}\boxed{\text{254 bit truncated } \rho}\right)$$

$$h_i = \mathbf{PRF}^{\mathbf{pk_i}}_{\mathsf{a_{sk}}}(\mathsf{h_{Sig}}) = \mathbf{CRH}\left(\boxed{\text{256 bit } \mathsf{a_{sk}}}\boxed{1}\boxed{0}\boxed{\text{i}}\boxed{\text{253 bit truncated } \mathsf{h_{Sig}}}\right)$$

2.3 Confidential Address Keypair

A keypair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ is generated by users who wish to receive coins under this scheme. The public $\mathsf{addr}_{\mathsf{pk}}$ is called a *protected address* and is a tuple $(\mathsf{a}_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}})$ which are the public components of a *spend authority* keypair $(\mathsf{a}_{\mathsf{pk}}, \mathsf{a}_{\mathsf{sk}})$ and a *key-private encryption* keypair $(\mathsf{pk}_{\mathsf{enc}}, \mathsf{sk}_{\mathsf{enc}})$. The private $\mathsf{addr}_{\mathsf{sk}}$ is called a *protected address secret* and is a tuple $(\mathsf{a}_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}})$ which are the respective *private* components of the aforementioned *spend authority* and *key-private encryption* keypairs.

Although users can accept payment from multiple parties with a single $\mathsf{addr}_{\mathsf{pk}}$ without either party being aware, it is still recommended to generate a new address for each expected transaction to maximize privacy in the event that multiple sending parties are compromised or collude.

2.4 Buckets

A bucket (denoted b) is a tuple $(\mathsf{v}, \mathsf{a}_{\mathsf{pk}}, \mathsf{r}, \rho)$ which represents that a value v is spendable by the recipient who holds the *spend authority* keypair $(\mathsf{a}_{\mathsf{pk}}, \mathsf{a}_{\mathsf{sk}})$ such that $\mathsf{a}_{\mathsf{pk}} = \mathbf{PRF}^{\mathbf{addr}}_{\mathsf{a}_{\mathsf{sk}}}(0)$. r and ρ are randomly generated tokens by the sender. Only a hash of these values is disclosed publicly, which allows these random tokens to blind the value and recipient *except* to those who possess these tokens.

In-band secret distribution In order to send the secret v, r and ρ to the recipient (necessary for the recipient to later spend) *without* requiring an out-of-band communication channel, the *key-private encryption* public key $\mathsf{pk}_{\mathsf{enc}}$ is used to encrypt these secrets to form an *encrypted bucket*. The recipient's possession of the associated $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$ (which contains both a_{pk} and $\mathsf{sk}_{\mathsf{enc}}$) is used to reconstruct the original bucket.

Bucket commitments The underlying v and a_{pk} are blinded with r and ρ using the collision-resistant hash function **CRH** in a multi-layered process. The resulting hash $\mathtt{bm} = \mathtt{BucketCommitment}(\mathsf{b})$.

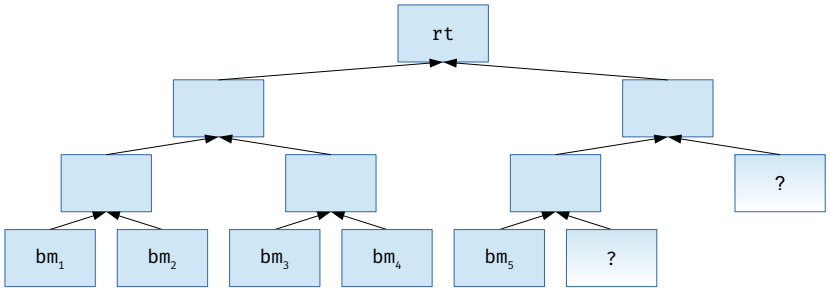
$$\mathsf{InternalH} := \mathbf{CRH}\left(\boxed{256 \text{ bit } \mathsf{a}_{\mathsf{pk}}}\boxed{256 \text{ bit } \rho}\right)$$

$$\mathsf{k} := \mathbf{CRH}\left(\boxed{256 \text{ bit } \mathsf{r}}\boxed{256 \text{ bit } \mathsf{InternalH}}\right)$$

$$\mathtt{bm} := \mathbf{CRH}\left(\boxed{64 \text{ bit } \mathsf{v}}\boxed{192 \text{ bit padding}}\boxed{256 \text{ bit } \mathsf{k}}\right)$$

Serials A serial number (denoted sn) equals $\mathbf{PRF}^{\mathbf{sn}}_{\mathsf{a}_{\mathsf{sk}}}(\rho)$. Buckets are spent by proving knowledge of ρ or a_{sk} in zero-knowledge while disclosing sn, allowing sn to be used to prevent double-spending.

2.5 Bucket Commitment Tree



The bucket commitment tree is an *incremental merkle tree* of depth `d` used to store bucket commitments that transactions produce. Just as the *unspent transaction output set* (UTXO) used in Bitcoin proper, it is used to express the existence of value and the capability to spend it. However, unlike the UTXO, it is *not* the job of this tree to protect against double-spending, as it is append-only.

Blocks in the blockchain are associated (by all nodes) with the root of this tree after all of its constituent transactions' bucket commitments have been entered into the tree associated with the previous block.

2.6 Spent Serials Map

Eli: Would be good to formally define the structure of a transaction, similar to the way a bucket is defined (as a quadruple). Transactions insert Eli: a tx is just a string, so it doesn't insert anything. Rather, nodes process tx's and the "good" ones lead to the addition of serials to the spent serials map. serials into a *spent serials map* which is maintained alongside the UTXO by all nodes. Transactions that attempt to insert a serial into this map that already exists within it are invalid as they are attempting to double-spend. Eli: After defining *transaction*, one should define what a *legal tx* is (this definition depends on a particular blockchain) and only then can one talk about "attempts" of transactions, and insertions of serial numbers into the spent serials map.

2.7 Bitcoin Transactions

Eli: Please formally define what a tx is. I don't think it's merely a sequence of inputs and outputs. The outputs are probably buckets (as defined above) and maybe the inputs are, too. Perhaps one should talk about a tx-bucket (which is unhidden) and a tx which is mostly hashed-stuff + a SNARK) Bitcoin transactions consist of a vector Eli: sequence? vector implies "vector space" which doens't exist here of inputs (`vin`) and a vector of outputs (`vout`). Inputs and outputs are associated with a value Eli: assuing a tx-bucket is a pair of sequences — an input-sequence and an output-sequence, and each sequence is a sequence of buckets, one should define the in-value of the tx-bucket as the sum of values in the in-buckets (ditto for out-value) and the remaining value is their difference. The total value of the outputs must not exceed the total value of the inputs.

Value pool Transaction inputs insert value into a *value pool*, and transaction outputs remove value from this pool. The remaining value in the pool is available to miners as a fee.

3 Pour Transactions (`PourTx`)

Eli: Hmm, I think things are starting to get confused here, let's try to clarify the theory/crypto. Informally, buckets and transactions are *data*, whereas Pour is best thought of as a *circuit* that outputs either 1 ("true") or 0 ("false"). The theory of SNARKs (as supported by libsnark) is such that if the circuit outputs "true" then you can generate a SNARK for that set of inputs, and otherwise you can't (its cryptographically infeasible to do so). So we should describe formally

what the *inputs* to the Pour circuit are, and then define the *computation preformed* by the Pour circuit, i.e., describe how it decides whether to output 0 or 1. More below

PourTxs are the primary operations Eli: In the academic paper, a Pour is a circuit (that defines an NP-language), and that circuit is the most crucial part of the construction. So if you want to use "Pour" to describe the algorithm that generates a tx, you'll be (i) deviating from the academic paper in a rather confusing way and (ii) you still need to define the "Pour-circuit" which is at the heart of the construction performed by transactions that interact with our scheme. In principle, it is the action of spending N_{Old} buckets $\mathsf{b}^{\mathsf{old}}$ and creating N_{New} buckets $\mathsf{b}^{\mathsf{new}}$. **Zcash** transactions have an additional field `vpour`, which is a vector of `PourTx`s. Each `PourTx` consists of: Eli: reached here

`vpub_old` which is a value $\mathsf{vpub}_{\mathsf{old}}$ that the pour removes from the value pool.

`vpub_new` which is a value $\mathsf{vpub}_{\mathsf{new}}$ that the pour inserts into the value pool.

`anchor` which is a merkle root rt of the bucket commitment tree at some block height in the past, or the merkle root produced by a previous pour in this transaction. **(TODO: clarify this)**

`scriptSig` which is a Bitcoin script which creates conditions for acceptance of a `PourTx` in a transaction. The `SHA256Compress` hash of this value is $\mathsf{h}_{\mathsf{Sig}}$.

`scriptPubKey` which is a Bitcoin script used to satisfy the conditions of the `scriptSig`.

`serials` which is an N_{Old} size vector of serials $\mathsf{sn}_1^{old}, \mathsf{sn}_2^{old}, ..., \mathsf{sn}_{N_{Old}}^{old}$.

`commitments` which is a N_{New} size vector of bucket commitments $\mathtt{bm}_1^{new}, \mathtt{bm}_2^{new}, ..., \mathtt{bm}_{N_{New}}^{new}$.

`encrypted_buckets` which is a N_{New} size vector of encrypted buckets.

`vmacs` which is a N_{Old} size vector of message authentication codes h which bind $\mathsf{h}_{\mathsf{Sig}}$ to each a_{sk} of the `PourTx`.

`zkproof` which is the zero-knowledge proof $\pi_{\mathtt{POUR}}$.

Merkle root validity A `PourTx` is valid if rt is a bucket commitment tree root found in either the blockchain or a merkle root produced by inserting the bucket commitments of a previous `PourTx` in the transaction to the bucket commitment tree identified by that previous `PourTx`'s `anchor`.

Non-malleability A `PourTx` is valid if the script formed by appending `scriptPubKey` to `scriptSig` returns *true*. The `scriptSig` is cryptographically bound to $\pi_{\mathtt{POUR}}$.

Balance A `PourTx` can be seen, from the perspective of the transaction, as an input and an output simultaneously. `vpub_old` takes value from the value pool and `vpub_new` adds value to the value pool. As a result, `vpub_old` is treated like an *output* value, whereas `vpub_new` is treated like an *input* value.

Commitments and Serials Transactions which contain `PourTx`s, when entered into the blockchain, append to the bucket commitment tree with all constituent bucket commitments. All of the constituent serials are also entered into the spent serials map of the blockchain *and* mempool. Transactions are not valid if they attempt to add a serial to the spent serials map that already exists.

3.1 $\pi_{\mathtt{POUR}}$

In **Zcash** , N_{Old} and N_{New} are both 2.

A valid instance of $\pi_{\mathtt{POUR}}$ assures that given a *primary input* (rt, sn_1^{old}, sn_2^{old}, $\mathtt{bm}_1^{new}$, $\mathtt{bm}_2^{new}$, $\mathsf{vpub}_{\mathsf{old}}$, $\mathsf{vpub}_{\mathsf{new}}$, $\mathsf{h}_{\mathsf{Sig}}$, h_1, h_2), a witness of *auxiliary input* (path_1, path_2, $\mathsf{b}_1^{\mathsf{old}}$, $\mathsf{b}_2^{\mathsf{old}}$, $\mathsf{a}_{\mathsf{sk}}{}_1^{old}$, $\mathsf{a}_{\mathsf{sk}}{}_2^{old}$, $\mathsf{b}_1^{\mathsf{new}}$, $\mathsf{b}_2^{\mathsf{new}}$) exists, where:

for each $i \in \{1, 2\}$: $\mathsf{b}_{\mathsf{i}}^{\mathsf{old}} = (\mathsf{v}_{\mathsf{i}}^{\mathsf{old}}, \mathsf{a}_{\mathsf{pk}}{}_i^{old}, \mathsf{r}_i^{old}, \rho_i^{old})$

for each $i \in \{1, 2\}$: $\mathsf{b}_{\mathsf{i}}^{\mathsf{new}} = (\mathsf{v}_{\mathsf{i}}^{\mathsf{new}}, \mathsf{a}_{\mathsf{pk}}{}_i^{new}, \mathsf{r}_i^{new}, \rho_i^{new})$.

The following conditions hold:

Merkle path validity for each $i \in \{1,2\} \mid \mathsf{v}_\mathsf{i}^{\mathsf{old}} \neq 0$: path_i must be a valid path of depth d from $\mathtt{BucketCommitment}(\mathsf{b}_\mathsf{i}^{\mathsf{old}})$ to bucket commitment merkle tree root rt.

Balance $\mathsf{vpub}_{\mathsf{old}} + \mathsf{v}_1^{\mathsf{old}} + \mathsf{v}_2^{\mathsf{old}} = \mathsf{vpub}_{\mathsf{new}} + \mathsf{v}_1^{\mathsf{new}} + \mathsf{v}_2^{\mathsf{new}}$.

Serial integrity for each $i \in \{1,2\}$: $\mathbf{PRF}^{\mathbf{sn}}_{\mathsf{a}_{\mathsf{sk}_\mathsf{i}}^{\mathsf{old}}}(\rho_i^{old}) = \mathsf{sn}_i^{old}$.

Spend authority for each $i \in \{1,2\}$: $\mathsf{a}_{\mathsf{pk}}{}_i^{old} = \mathbf{PRF}^{\mathbf{addr}}_{\mathsf{a}_{\mathsf{sk}_\mathsf{i}}^{\mathsf{old}}}(0)$.

Non-malleability for each $i \in \{1,2\}$: $h_i = \mathbf{PRF}^{\mathbf{pk_i}_{-1}}_{\mathsf{a}_{\mathsf{sk}_\mathsf{i}}^{\mathsf{old}}}(\mathsf{h}_{\mathsf{Sig}})$

Commitment integrity for each $i \in \{1,2\}$: $\mathtt{bm}_i^{new} = \mathtt{BucketCommitment}(\mathsf{b}_\mathsf{i}^{\mathsf{new}})$

4 Encoding addresses, private keys, buckets, and pours

This section describes how **Zcash** encodes public addresses, private keys, buckets, and pours.

Addresses, keys, and buckets, can be encoded as a byte string; this is called the *raw encoding*. This byte string can then be further encoded using Base58Check. The Base58Check layer is the same as for upstream Bitcoin addresses.

SHA-256 compression function outputs are always represented as strings of 32 bytes.

The language consisting of the following encoding possibilities is prefix-free.

4.1 Cleartext Public Addresses

TBD. Identical to Bitcoin?

4.2 Cleartext Private Keys

TBD. Identical to Bitcoin?

4.3 Protected Public Addresses

A protected address consists of a_{pk} and $\mathsf{pk}_{\mathsf{enc}}$. a_{pk} is a SHA-256 compression function output. $\mathsf{pk}_{\mathsf{enc}}$ is an encryption public key (currently ECIES, but this may change to Curve25519/crypto_box), which is an elliptic curve point.

4.3.1 Raw Encoding

The raw encoding of a protected address consists of:

0x??	a_{pk} (32 bytes)	A 33-byte encoding of $\mathsf{pk}_{\mathsf{enc}}$

- A byte, 0x??, indicating this version of the raw encoding of a **Zcash** public address.
- 32 bytes specifying a_{pk}.
- An encoding of $\mathsf{pk}_{\mathsf{enc}}$: The byte 0x01, followed by 32 bytes representing the x coordinate of the elliptic curve point according to the FE20SP primitive specified in section 5.5.4 of IEEE Std 1363-2000. [Non-normative note: Since the curve is over a prime field, this is just the 32-byte big-endian representation of the x coordinate. The overall encoding matches the EC20SP-X primitive specified in section 5.5.6.3 of IEEE Std 1363a-2004.]

TODO: pick a version byte distinct from other Bitcoin stuff, and that produces the correct Base58 leading character

TODO: what about the network version byte?

4.4 Protected Address Secrets

A protected address secret consists of $\mathsf{a_{sk}}$ and $\mathsf{sk_{enc}}$. $\mathsf{a_{sk}}$ is a SHA-256 compression function output. $\mathsf{sk_{enc}}$ is an encryption private key (currently ECIES), which is an integer.

4.4.1 Raw Encoding

The raw encoding of a protected address secret consists of, in order:

0x??	$\mathsf{a_{sk}}$ (32 bytes)	$\mathsf{sk_{enc}}$ (32 bytes)

- A byte 0x?? indicating this version of the raw encoding of a Zcash private key.
- 32 bytes specifying $\mathsf{a_{sk}}$.
- 32 bytes specifying a big-endian encoding of $\mathsf{sk_{enc}}$.

TODO: pick a version byte distinct from other Bitcoin stuff, and that produces the correct Base58 leading character
TODO: what about the network version byte?

4.5 Buckets (on the blockchain)

A bucket consists of $(\mathsf{addr_{pk}}, \mathsf{v}, \rho, \mathsf{r}, \mathtt{bm})$, where:

- $\mathsf{addr_{pk}}$ is a **Zcash** public address.
- v is a 64-bit unsigned integer representing the value of the bucket in zatoshi.
- ρ is a 32-byte $\mathbf{PRF}^{\mathbf{sn}}_{\mathsf{a_{sk}}}$ seed.
- r is a 32-byte COMM trapdoor.
- $\mathtt{bm}$ is a commitment which is a SHA-256 compression function output.

Note that the value s described as being part of a bucket/coin in the Zerocash paper is not encoded because it is fixed to zero.

4.6 Raw Encoding

The raw encoding of a **Zcash** bucket consists of, in order:

0x??	$\mathsf{addr_{pk}}$	v (8 bytes, big endian)	ρ (32 bytes)	r (32 bytes)	$\mathtt{bm}$ (32 bytes)

- A byte 0x?? indicating this version of the raw encoding of a **Zcash** bucket.
- 65 bytes specifying the raw encoding of the **Zcash** public address $\mathsf{addr_{pk}}$ (defined above).
- 8 bytes specifying a big-endian encoding of v.
- 32 bytes specifying ρ.
- 32 bytes specifying r.
- 32 bytes specifying $\mathtt{bm}$.

5 Pours (within a transaction on the blockchain)

TBD.

6 Transactions

TBD.