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).

2 Concepts

2.1 Endianness

All numerical objects in Zcash are big endian.

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.

$\mathbf{PRF}_{\mathbf{x}}$ 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. 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 $(\mathsf{addr_{pk}}, \mathsf{addr_{sk}})$ is generated by a user any time they wish to receive value from another in the system Eli: The previous sentence seems to imply that a fresh pair is needed for each new receive, but this isn't the case. Perhaps change to "To receive/pay values in the system, a user must set-up a key pair . . . ; differing from Bitcoin, replacing keys is not needed to protect privacy.". The public $\mathsf{addr_{pk}}$ is called a *protected address* and is a tuple $(\mathsf{a_{pk}}, \mathsf{pk_{enc}})$ which are the public components of a *spend authority* keypair $(\mathsf{a_{pk}}, \mathsf{a_{sk}})$ and a *key-private encryption* keypair $(\mathsf{pk_{enc}}, \mathsf{sk_{enc}})$. The private $\mathsf{addr_{sk}}$ is called a *protected address secret* and is a tuple $(\mathsf{a_{sk}}, \mathsf{sk_{enc}})$ which are the respective *private* components of the aforementioned *spend authority* and *key-private encryption* keypairs.

2.4 Buckets

A bucket (denoted b) is a tuple $(\mathsf{v}, \mathsf{a}_{\mathsf{pk}_i}, \mathsf{r}, \rho)$ Eli: subscript i should be removed from bucket which represents that a value v is spendable by the recipient who holds the *spend authority* keypair $(\mathsf{a}_{\mathsf{pk}}, \mathsf{a}_{\mathsf{sk}})$ Eli: more precise: any user that knows a_{sk} such that $\mathsf{a}_{\mathsf{pk}} = \mathbf{PRF}^{\mathbf{addr}}_{\mathsf{a}_{\mathsf{sk}}}(0)$. r and ρ are randomly generated tokens which are used to blind the value and recipient *except* to those who possess these tokens. Eli: last sentence unclear. Zcash transactions contain only hashes of financial information (like v and a_{pk}) and r and ρ are used to hide this data.

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 $\mathtt{bm}$ is called a *bucket commitment*.

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

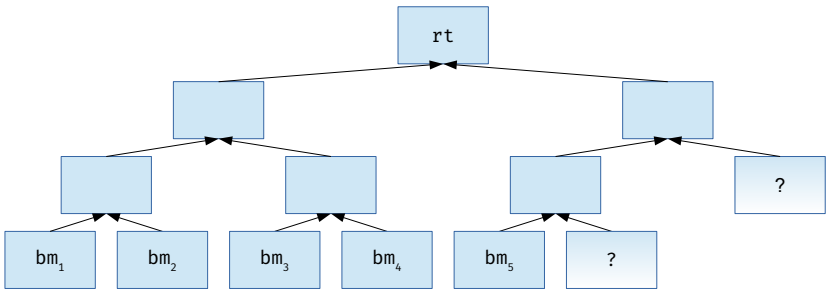
$$\mathsf{k} := \mathbf{CRH}\left(\boxed{256 \text{ bit } \mathsf{r}}\boxed{256 \text{ bit } \mathsf{InternalH}}\right)$$

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

We say that the bucket commitment of a bucket $\mathsf{b} = \mathtt{BucketCommitment}(\mathsf{b})$.Eli: circular definition: $b = f(b)$?

Serials A serial sn Eli: serial serial number? is produced by Eli: equals? $\mathbf{PRF}^{\mathbf{sn}}_{\mathsf{a}_{\mathsf{sk}}}(\rho)$. Part of the process of spending a bucket is disclosing this serial without disclosing either ρ or a_{sk}. This allows it to be used to prevent double-spending. Eli: More precise: *Knowledge* of ρ and a_{sk} is required to spend a bucket. Knowledge of ρ and a_{sk} is proved without revealing it explicitly via a zero-knowledge (zk)SNARK.

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

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

`Pour`s are the primary operations Eli: In the academic paper, a Pour is a circuit, 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 `Pour`s. Each `Pour` consists of:

`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 `Pour` 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 $\mathsf{h_{Sig}}$ to each $\mathsf{a_{sk}}$ of the `Pour`.

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

Merkle root validity A `Pour` 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 `Pour` in the transaction to the bucket commitment tree identified by that previous `Pour`'s `anchor`.

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

Balance A `Pour` 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 `Pours`, 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}, $\mathtt{bm}_1^{new}$, $\mathtt{bm}_2^{new}$, $\mathsf{vpub_{old}}$, $\mathsf{vpub_{new}}$, $\mathsf{h_{Sig}}$, h_1, h_2), a witness of *auxiliary input* (path_1, path_2, $\mathsf{b}_1^{\mathsf{old}}$, $\mathsf{b}_2^{\mathsf{old}}$, $\mathsf{a_{sk}}_1^{old}$, $\mathsf{a_{sk}}_2^{old}$, $\mathsf{b}_1^{\mathsf{new}}$, $\mathsf{b}_2^{\mathsf{new}}$) exists, where:

for each $i \in \{1, 2\}$: $\mathsf{b_i^{old}} = (\mathsf{v_i^{old}}, \mathsf{a_{pk}}_i^{old}, \mathsf{r}_i^{old}, \rho_i^{old})$

for each $i \in \{1, 2\}$: $\mathsf{b_i^{new}} = (\mathsf{v_i^{new}}, \mathsf{a_{pk}}_i^{new}, \mathsf{r}_i^{new}, \rho_i^{new})$.

The following conditions hold:

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

Balance $\mathsf{vpub_{old}} + \mathsf{v_1^{old}} + \mathsf{v_2^{old}} = \mathsf{vpub_{new}} + \mathsf{v_1^{new}} + \mathsf{v_2^{new}}$.

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

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

Non-malleability for each $i \in \{1, 2\}$: $h_i = \mathbf{PRF}^{\mathbf{pk_{i-1}}}_{\mathsf{a_{sk_i^{old}}}}(\mathsf{h_{Sig}})$

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