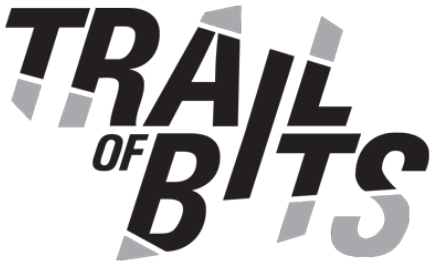


Zcash Whitepaper

Prepared For:
Benjamin Winston | *Electric Coin Co.*
bambam@electriccoin.co

Prepared By:
Benjamin Perez | *Trail of Bits*
benjamin.perez@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

James Miller | *Trail of Bits*
james.miller@trailofbits.com

Paul Kehrer | *Trail of Bits*
paul.kehrer@trailofbits.com

Introduction

The Zcash Protocol

Zcash is a privacy-preserving cryptocurrency based on the Bitcoin protocol. Unlike Bitcoin, however, Zcash provides an additional mechanism for anonymously transferring funds on the blockchain. These privacy features require the use of complex cryptographic primitives. As such, the Zcash specification is a highly technical document that can be difficult for newcomers to digest. Throughout our audit of ZIPs 213 and 221, the Trail of Bits team compiled notes on the high-level functionality of the Zcash system, and we believe these notes may be useful for other developers trying to get acquainted with the protocol.

ZCash users can have two different types of addresses: transparent addresses (t-addrs), and shielded addresses (z-addrs). The former behave identically to normal Bitcoin addresses, and the latter can be used to execute private transactions. At a high level, we can think of the shielded pool as a bulletin board. When someone wants to send ZEC to a shielded address, they simply post a note on the bulletin board that contains encrypted information about the transaction and a zero-knowledge proof (ZK-SNARK) that the underlying transfer of money is valid. The recipient can determine which notes on the bulletin board are theirs by attempting to decrypt each new note and extracting the contents when successful. As a result, the contents of the notes are only visible to the sender and receiver.

When a user wants to transfer one of their notes, they post a *nullifier* (a special object that indicates a note has been spent) to the bulletin board, along with a new note for their intended recipient and a SNARK of transaction validity. Observers of the bulletin board can't tell which note a nullifier corresponds to, only that it corresponds to *some* note. Since nullifiers are unique, double spending can be prevented by ensuring no duplicates arise.

Sapling

Recently, the Zcash protocol underwent a major upgrade called Sapling, primarily because generating the zero-knowledge proofs required for shielded transactions in the original Zcash protocol was computationally quite expensive. Indeed, proof generation could consume over three gigabytes of RAM and take several minutes to complete—performance that was unacceptable for resource-constrained hardware such as mobile phones and IoT devices. By switching to a new ZK-SNARK scheme, Sapling is able to generate proofs in under one second while using approximately 40 megabytes of RAM.

Besides improved proof generation, Sapling provides users with a more fine-grained set of permissions for each shielded address. Users can now separate proving and spending

capabilities for their account, allowing them to keep the spending key on trusted hardware and generate proofs on a more computationally powerful device. Furthermore, there are now viewing keys for both incoming and outgoing transactions. While viewing keys have always been a part of the Zcash protocol, the ability to allow a third party to inspect outgoing transactions facilitates auditing and compliance enforcement for companies and exchanges.

Finally, Sapling introduces diversified addresses, which provide an efficient way for recipients to generate randomized public addresses. If a recipient uses diversified addresses for multiple senders, it prevents the senders from learning whether they are sending to the same recipient, even if they collude.

Protocol

Keys and Permissions

This section details the various keys used throughout the Sapling protocol. We use the following colors to indicate which parties should possess these keys: red for secret keys that should only be stored by the user, orange for values that may be shared with trusted parties, and blue for public values.

The spending key (`sk`) is used to derive all of the other keys. Therefore, it can be used for all functionalities in the protocol: spending, receiving transactions, generating proofs, etc. This key must be kept private; ideally, this key is only stored on a user's trusted device.

In order for a note to be spent, both a *spend authorizing signature* and a SNARK of the spend statement's validity are needed. The spend authorizing key (`ask`) is used to produce a *spend authorizing signature*. The proof authorizing key (`ak,nsk`) is used to create a SNARK of the spend statement, where `ak` is derived from `ask` and `nsk` is derived from `sk` (both keep their parent keys secret). Since generating SNARKs can be computationally intensive, `ak` and `nsk` may be shared with a trusted party so they can generate these proofs on the user's behalf, while `ask` must be kept secret to prevent the trusted party from spending.

Since all shielded transactions are encrypted, users are required to scan and attempt to decrypt every shielded transaction in every new block, which can be prohibitive for resource-constrained users. To remedy this, Sapling allows users to delegate the viewing of incoming and outgoing shielded transactions to third parties, while preventing them from actually spending such transactions. The incoming viewing key (`ivk`) is derived from the proof authorizing key (`ak,nsk`) and is used to view incoming outputs. In other words, all transactions sent to addresses corresponding to `sk` can be viewed by anyone in possession of `ivk`. This also means anyone authorized to generate proofs can view incoming transactions. The outgoing viewing key (`ovk`) is derived directly from `sk` and is used to view

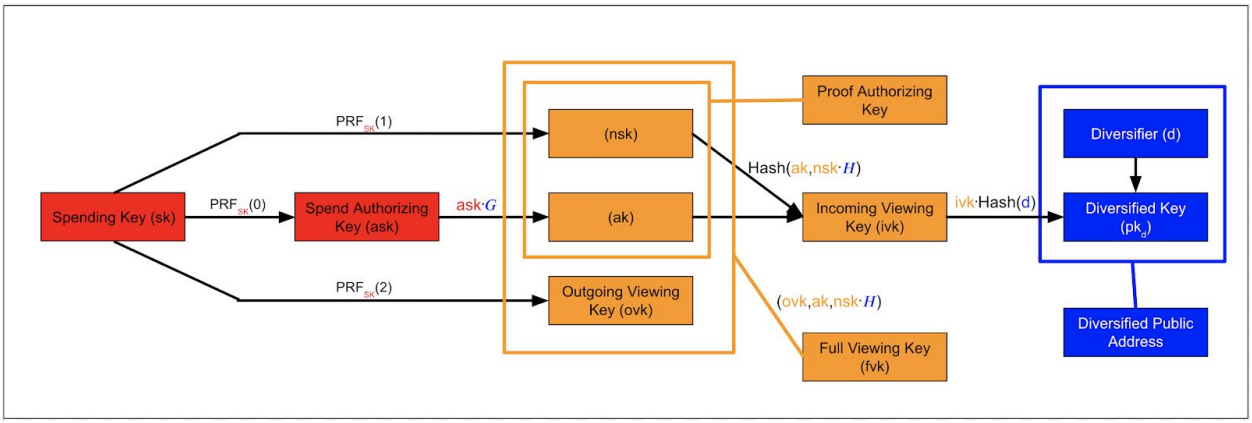
outgoing transactions, i.e., transactions sent from an address corresponding to sk. The full viewing key (fvk) is comprised of (ak,nsk) and ovk, and can be used to view both incoming and outgoing transactions.

To improve anonymity, Sapling provides an efficient way to generate multiple public addresses that correspond to the same sk. Specifically, a random value called the diversifier (d) is generated and is combined with ivk to generate the diversified public key (pk_d). Together, both d and pk_d comprise a *diversified public address*. A user can then generate a new *diversified public address* whenever another user requests their address, and multiple, colluding users cannot identify when they all send notes to the same spending authority.

The key generation process starts with randomly generating a 32-byte spending key, sk. We can then derive keys in the following ways:

- ask = $PRF_{sk}(0)$
- nsk = $PRF_{sk}(1)$
- ovk = Truncate(32,$PRF_{sk}(2)$)
- ak = ask * G
- nk = nsk * H
- fvk = (ovk,ak,nk)
- ivk = OctToInt(SomeHashFunction(ak,nk)) % 2^{251}
- pk_d = ivk * SomeHashFunction(d)

For a visualization of the key hierarchy and how each key is derived, see the following image.



Transactions

As discussed in the introduction, there are two types of addresses in Zcash: transparent addresses (t-addrs) and shielded addresses (z-addrs). We can characterize transactions on the Zcash blockchain to be:

- Public (t-t)
- Shielding (t-z)
- Private (z-z)
- Deshielding (z-t)

Transactions between transparent addresses are identical to standard Bitcoin transactions. In particular, they contain a list of inputs and outputs, along with the sender's signature. Any transaction involving a z-addr does not reveal this address to the network, so in the case of shielding and deshielding operations, only the t-addr is made public. Private transactions do not reveal the amount of ZEC sent; however both shielding and deshielding transactions do. Therefore, such transactions are amenable to correlation analysis and may leak information about activity within the shielded pool. For more information see *An Empirical Analysis of Anonymity in Zcash* by Kappos et al.

To see how the entire transaction model works, we begin with basic Bitcoin transactions, which are comprised entirely of t-addrs. A block consists of a coinbase transaction, minting a “new coin” and giving some ZEC to the miner and a founder. The rest of the block is comprised of transactions that contain transparent inputs (value from addresses) and transparent outputs (value to addresses). A transaction cannot output more than the inputs, and the difference between input and output values are awarded to the miner who mined this block.

To handle shielded transactions, Zcash introduces Spend and Output transfers. The notes in Spend and Output transfers behave like banknotes, where someone holds some value, but no one knows who holds it or what the value is. The only publicly identifiable information about a note is its commitment. Spending a note requires publishing a unique nullifier, which is computed privately and publicly verified via a zero-knowledge proof. Outputting a note requires encrypting the note details to the sender and recipient, and generating a new commitment for this note.

A series of Spend and Output transfers can then be combined into a single transaction in order to spend old notes and produce new notes. Each Spend and Output transfer contains its own SNARK to verify its correctness (i.e., the notes were formed correctly, the spender has authority to spend, etc.). In addition, Spend transfers are accompanied by the nullifier for the note being spent. Depending on the note balances, values may be added or

removed from the transparent pool. A *balance and binding signature* is attached to a transaction to prove that it is properly balanced.

When the recipient of a transaction is a shielded address, extra information is required to allow them to both deduce when they have been sent ZEC, and come into the possession of cryptographic credentials required to spend the ZEC (i.e., the nullifier). One solution to this would be out-of-band communication between the sender and recipient. Zcash, however, is able to avoid this situation by including an encrypted `note` field containing the information required for the recipient to spend the ZEC. Therefore, shielded addresses must attempt to decrypt the note field of every incoming shielded transaction. This process can be computationally expensive, but is partially mitigated by the existence of the viewing key, which allows a shielded address to delegate the ability to detect incoming transactions to a third party without compromising their identity or the ability to spend the incoming funds.

Spend Descriptions

Since a spend transaction will be spending and nullifying an already existing note, the spend description must contain information pertaining to the note being spent, e.g., the nullifier, its Merkle Tree anchor and path, and its commitment. It will also need a zero-knowledge proof to prove its validity without revealing any sensitive information.

Specifically, a spend description includes the following:

- ρ, a Merkle Tree anchor, to indicate which block and commitment tree the note being spent corresponds to
- c_v, a Pedersen commitment to the value v of the note being spent
- n, a nullifier for the note, supposedly derived from the note commitment, c_m, and nsk
- k, a randomized public key supposedly derived from ak
- π, a SNARK that asserts the following:
    - The spender knows a valid commitment, c_m, to be spent, and the commitment value, c_v, is valid
    - The spender knows a valid Merkle path (in the same Merkle Tree as ρ) pointing to the valid commitment
    - n is valid and correctly derived from c_m and nsk
    - The randomized public key, k, is in fact derived from ak
    - The spender knows the public, diversified address that will be the recipient, and this address is valid (and not of small order on the elliptic curve)
- σ, a *spend authorization signature* for this spend description, which must be generated with knowledge of ask, which only allows the true owner of a note to spend it

It is important that one keeps $\mathtt{ask}$ secret, because this allows only the true owner of a note to authorize its spending and nullification.

Output Descriptions

The output description does not need as much information as spend descriptions require. They will contain the commitment notes that are being sent to the recipient. To maintain privacy, some of the contents of the output description must be encrypted under a key that only the sender and receiver can derive.

Specifically, an output description includes the following:

- $\mathtt{c}_v$, a value commitment for the output note
- $\mathtt{k}$, an ephemeral public key used by the receiver to derive a shared secret used to decrypt the note ciphertext
- $\mathtt{C}$, the ciphertext of the output note, which contains fields $\mathtt{d}$, the diversifier; $\mathtt{v}$, the note value; $\mathtt{r}$, the random value used in the note commitment; and a multi-purpose memo field
- $\mathtt{C'}$, the ciphertext that allows the owner of the outgoing viewing key to decrypt the recipient's $\mathtt{pk}_d$ and the ephemeral private key $\mathtt{p}$ used for this output note
- π, a SNARK that asserts the following:
    - The sender sent a valid commitment, $\mathtt{c}_m$, with a valid commitment value, $\mathtt{c}_v$
    - The sender knows the pair $(\mathtt{g}_d, \mathtt{pk}_d)$, derived from the diversifier, $\mathtt{d}$, and $\mathtt{g}_d$ is not of small order on the elliptic curve
    - The ephemeral key was generated properly from $\mathtt{g}_d$

Sapling Transactions

A Sapling shielded transaction consists of some combination of spend and output descriptions. Inside each shielded transaction we also put in plaintext the transaction balance, which will be equal to:

$$\sum_{v \in Spends} v - \sum_{v \in Outputs} v$$

This indicates how many notes will be added or subtracted from the transparent value pool. In order to tie multiple descriptions into a single transaction, a user will also produce a *balance and binding signature*, which establishes that the sum of spend commitments minus the sum of the output commitments is equal to this transaction balance. It also prevents adversaries from replaying outputs by proving the sender knows the randomness used to construct the commitments.

With all these building blocks now in place, we can dissect the anatomy of a Zcash transaction and understand how Sapling is able to maintain blockchain validity in zero

knowledge. We begin with the output note. A note is sent to a diversified address (d, pk_d) with note ciphertext encrypted from a shared secret derived from the public value $k = p * g_d$, where p is the private component known only by the sender. The shared secret is computed as ivk times k as the receiver and p times pk_d as the sender. Notice the equivalence:

$$ivk * k = ivk * (p * g_d) = p * (ivk * g_d) = p * pk_d$$

Thus, a shared symmetric encryption key may be derived to try and decrypt the note. The receiver now receives:

- The associated diversifier d
- The value of the note v
- The randomness r_x used in the commitment x
- A memo (which can contain various types of information)

The public commitment x is then published to the list of note commitments, where it awaits nullification for spending.

Now, a holder of a note knows d, the diversifier for the address the note was sent to; v, the value of the note; and r_x, the randomness used in the commitment. A holder of that note is thus able to recompute g_d and pk_d and reconstruct the commitment. They compute a nullifier using their nsk and authorize the spend transaction with their private ask. Utilizing a ZK proof, they convince verifiers of the blockchain that they have the correct knowledge to actually nullify and spend the notes. Since ask is never revealed to anyone, only the holder of the note can authorize spending. The commitment scheme and the ZK proofs ensure the correct nullifier key must be used when computing the nullifier, and it is tied to the commitment. In addition, the correct nullifier for the note is unique. Once the proof is validated, the published nullifier will be added to the nullifier list, preventing double spends. Therefore, we can, in zero knowledge, spend a note of unknown origin, value, and commitment.

Consensus

Like Bitcoin, Zcash has a proof of work consensus mechanism. In Zcash, all pending transactions are put into a transaction pool, and the full validators use the transactions in this pool to form new candidate blocks to be added into the blockchain. The proof of work scheme requires the full validators to solve a particular problem instance that they can derive from their candidate block. Unlike Bitcoin, Zcash uses a proof of work scheme known as EquiHash that is “memory hard,” meaning the work to find the block's solution is memory-intensive, and specialized computational hardware has less of an advantage over traditional computers. A block is added to the blockchain when a full validator finds a

solution that meets the problem's constraints and publishes this to other validators, which they then verify before accepting. When other validators receive multiple valid blockchains ties are broken by whichever required more work (in terms of the EquiHash scheme) and then by whichever was received first.

A Zcash block consists of a block header and a list of transactions. The block header contains the Merkle Tree anchor, the EquiHash solution, and other information, like version number and time. The transaction contains, among many other things, the commitments, nullifiers, Merkle paths, and SNARK proofs for the notes being spent, as well as the commitments and SNARK proofs for the output notes. When a full validator sends a block to the other nodes, a series of verification checks are required to establish the validity of the block.

As mentioned, EquiHash provides an efficient way to verify solutions. The Merkle Tree anchor is used to verify that the list of transactions is valid and has not been changed. Specifically, the commitment tree can be reconstructed from the transaction list, and (assuming the hash function is collision-resistant) if the resulting Merkle Tree root is the same as the anchor provided, the commitments in the transactions will correspond to the same Merkle Tree as the anchor. Lastly, the block will be valid only if the correct version number is used and the other values are as expected (such as the value for time being within two hours of the node's current clock).

The information inside the transactions themselves also needs to be verified in order to establish the block's validity. Specifically, a validator checks that each nullifier corresponds to the note being spent, and that the nullifier does not already exist. The validator also uses the Merkle path to verify that the commitment corresponds to a valid commitment on a previous block's Merkle Tree. Lastly, the SNARKs attached to the transactions should also be checked by running the corresponding SNARK verification algorithm.

ZIPs

ZIP 213: Shielded Coinbase

ZIP 213 modifies the consensus rules of Zcash to allow coinbase transactions to contain shielded outputs. Previously, all coinbase transactions were required to have only transparent outputs and then subsequently enter the shielded pool. In particular, the following changes are made to the Sapling consensus rules:

1. The consensus rule preventing coinbase transactions from containing shielded outputs is no longer active, and coinbase transactions MAY contain Sapling outputs.
2. The consensus rules applied to `valueBalance`, `vShieldedOutput`, and `bindingSig` in non-coinbase transactions MUST also be applied to coinbase transactions.

3. The existing consensus rule requiring transactions that spend coinbase outputs to have an empty `vout` is amended to only apply to transactions that spend transparent coinbase outputs.
4. The existing consensus rule requiring coinbase outputs to have 100 confirmations before they may be spent (coinbase maturity) is amended to only apply to transparent coinbase outputs.
5. All Sapling outputs in coinbase transactions MUST have valid note commitments when recovered using a 32-byte array of zeros as the outgoing viewing key.

Trail of Bits did not find any security or privacy concerns in these modifications to the Sapling specification. However, we believe that users may misinterpret this ZIP to mean that coinbase transactions can now be made untraceable, or that it is now more difficult to analyze money flowing through the shielded pool. Before explaining why this is not the case, we'll look at prior work on de-anonymizing shielded transactions.

In *An Empirical Analysis of Anonymity in Zcash*, Kappos et al. shrink the anonymity set of Zcash substantially by identifying shielding and deshielding transactions that contain miner and founder rewards. In the case of `t-z` transactions, this task is fairly straightforward since ZEC from a coinbase transaction must go into the shielded pool. For `z-t` transactions, several heuristics can be used to determine if they are for miner rewards, such as timing information between shielding and deshielding transactions. Mining pools can be easily detected, since they often result in `z-t` transactions with many (> 100) outputs. Miner and founder rewards account for an overwhelming majority of `t-z` and `z-t` transactions—nearly 70%. Once these transactions have been identified, the remaining 30% are easier to analyze, especially considering that in total only about 10% of Zcash transactions are shielded.

Ideally, allowing coinbase transactions to have shielded outputs would prevent this type of analysis, or at least reduce its efficacy. However, ZIP 213 mandates that all shielded coinbase transactions have an outgoing viewing key of 0, meaning that any blockchain observer can identify these commitments and their recipient (though the recipient will be a diversified address). This is unfortunately necessary, since otherwise nodes would be unable to determine the validity of coinbase transactions. The ability for anyone to identify shielded coinbase transactions means that the techniques discussed above are still feasible. While it may no longer be possible to directly correlate transparent addresses entering miner rewards into the shielded pool with those receiving them through a deshielding transaction, it remains easy to detect which deshielding operations stem from coinbase transactions, especially in the case of mining pools. Therefore, it will still be possible to reduce the anonymity set by an amount similar to the one in the Kappos paper.

The above discussion shouldn't be misconstrued as a critique of ZIP 213. Rather, we're highlighting the security properties that users may mistakenly believe ZIP 213 possesses. In general, discouraging the use of transparent addresses in the Zcash ecosystem is a boon to

everyone's privacy, and allowing shielded coinbase transactions is a major step in that direction.

ZIP 221: FlyClient Zcash SPV

ZIP 221 proposes to add a new data structure into the Zcash Sapling protocol: a Merkle Mountain Range (MMR). To accomplish this, each block in the blockchain is stored in this MMR, and the root of the resulting tree is added into the block headers. Inclusion of the MMR allows the FlyClient protocol to run on Zcash Sapling. This protocol also allows computationally limited devices to verify:

- The validity of a blockchain received from a node (without having to download the entire chain or even every block header)
- The inclusion of a block in the blockchain as well as certain block metadata

The authors of the ZIP mention that the addition of the MMR will result in an increased validation cost, as validators now have to maintain the MMR for every block added, which they are concerned could worsen existing denial-of-service (DoS) attacks. Specifically, they mention an attack scenario in which an adversary maintains two different blockchains of approximately equal length. The adversary can alternately send these chains to a node, causing them to repeatedly reorganize their MMR. However, as they mention, the cost to add a block to the MMR will be at worst `O(log(n))` for `append` and `delete` operations (where `n` is the number of blocks). Both of these operations use `Blake2b`, which is very fast. Also, unless this adversary has significantly better mining performance than the rest of the nodes, the probability of successfully maintaining a separate fork of length `k` is exponentially small in terms of the size of `k`. A successful attack would require on the order of `O(k*log(n))` operations for the validator (at most `O(log(n))` for each block).

In order for this attack to be effective, an attacker needs a large enough `k` to cause a DoS on the client, but a `k` value of that size makes the probability of success negligible. Therefore, this should only be a concern if this adversary has significantly better mining performance, which would break one of the fundamental security assumptions of the system.

The authors of the ZIP also expressed an interest in including commitments to nullifier vectors in the node metadata; they decided against it, but have also asked for comments on the idea. As the FlyClients are limited in resources, we assume they will not be storing the entire nullifier set locally. Presumably, the idea would be for the nodes to maintain a nullifier MMR (or similar data type), so that FlyClients can verify the nullifier set or verify that a particular nullifier is in the set, all without having to store the entire set themselves. However, in order for a FlyClient to verify whether a nullifier is in the set, they would have to query the FlyClient server for a proof for that particular nullifier. This would be a privacy concern because the server can clearly identify particular commitments that each FlyClient

is interested in (and presumably involved with). Therefore, we support the decision not to include this.

While this ZIP only covers the addition of MMR commitments in the block header, we believe it is worth noting the privacy risks involved with using FlyClient within the Zcash ecosystem. For transparent addresses, FlyClient poses no risk, since these are simply Bitcoin addresses, and transactions between them are public. However, using FlyClients in a privacy-preserving manner when sending or receiving funds from shielded addresses becomes difficult. For instance, if a user sends someone money from a shielded address and wants to verify the transaction has actually been committed to the blockchain, they will need to query a full node to get the MMR proof that the transaction was included in a block. Performing this query would signal to the proving node that the client is interested in the transaction in question. Therefore, shielded addresses running FlyClient cannot make queries about specific transactions without losing privacy.

Currently, the only solution for this issue is having FlyClients affiliated with shielded addresses scan every transaction committed to the chain. While this imposes a somewhat substantial computational burden on the client, it doesn't require any information storage. The Zcash team is currently discussing ways to make this scanning process more efficient. Despite the fact that shielded addresses need to perform this scanning operation, the FlyClient protocol is useful in reducing storage requirements for users.

Finally, there are a few small errors in the ZIP we wanted to point out. The top portion of Figure 2 seems to have swapped the position of `left_child.nEarliestHeight` and `right_child.nLatestHeight`. Also, in the pseudocode function `get_peaks`, it is possible for the input node to be a leaf node (or the only node in a single node MMR). This would result in a node with a `null left_child` and `right_child`. If this is the input, the current code would result in an error. A quick fix would be to verify the children are not `null`.