Lesson 6 - Confidential Tokens

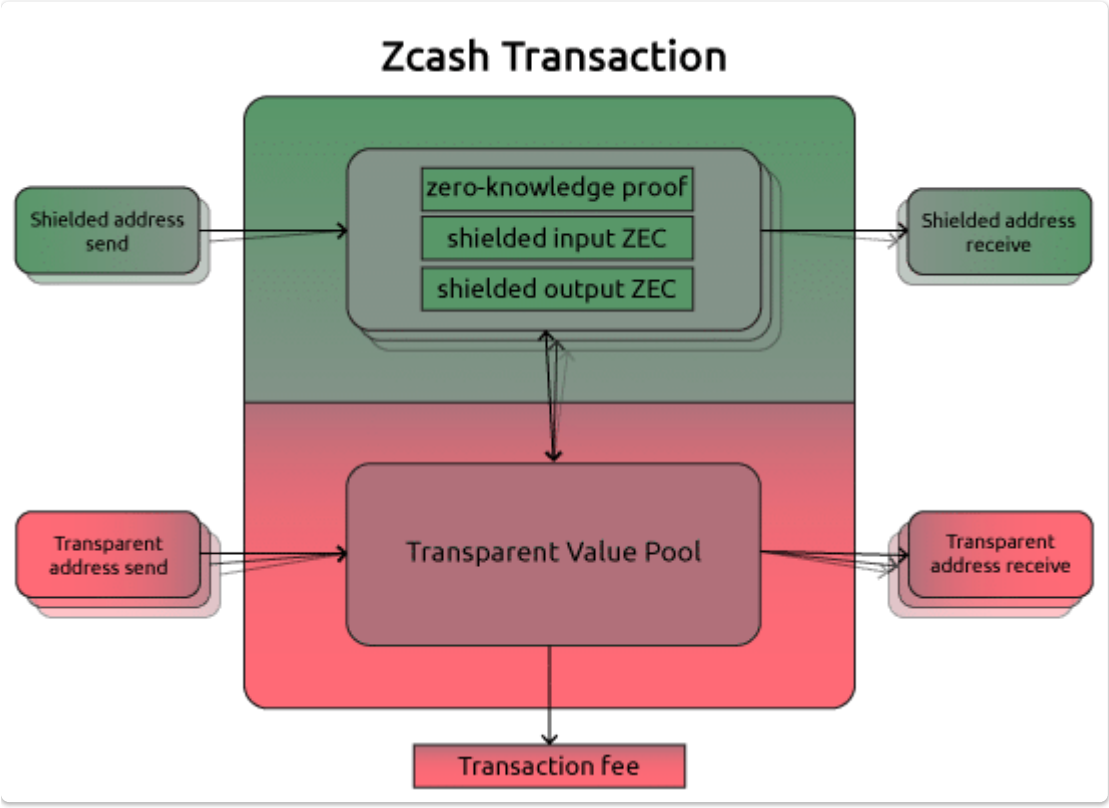
ZCash

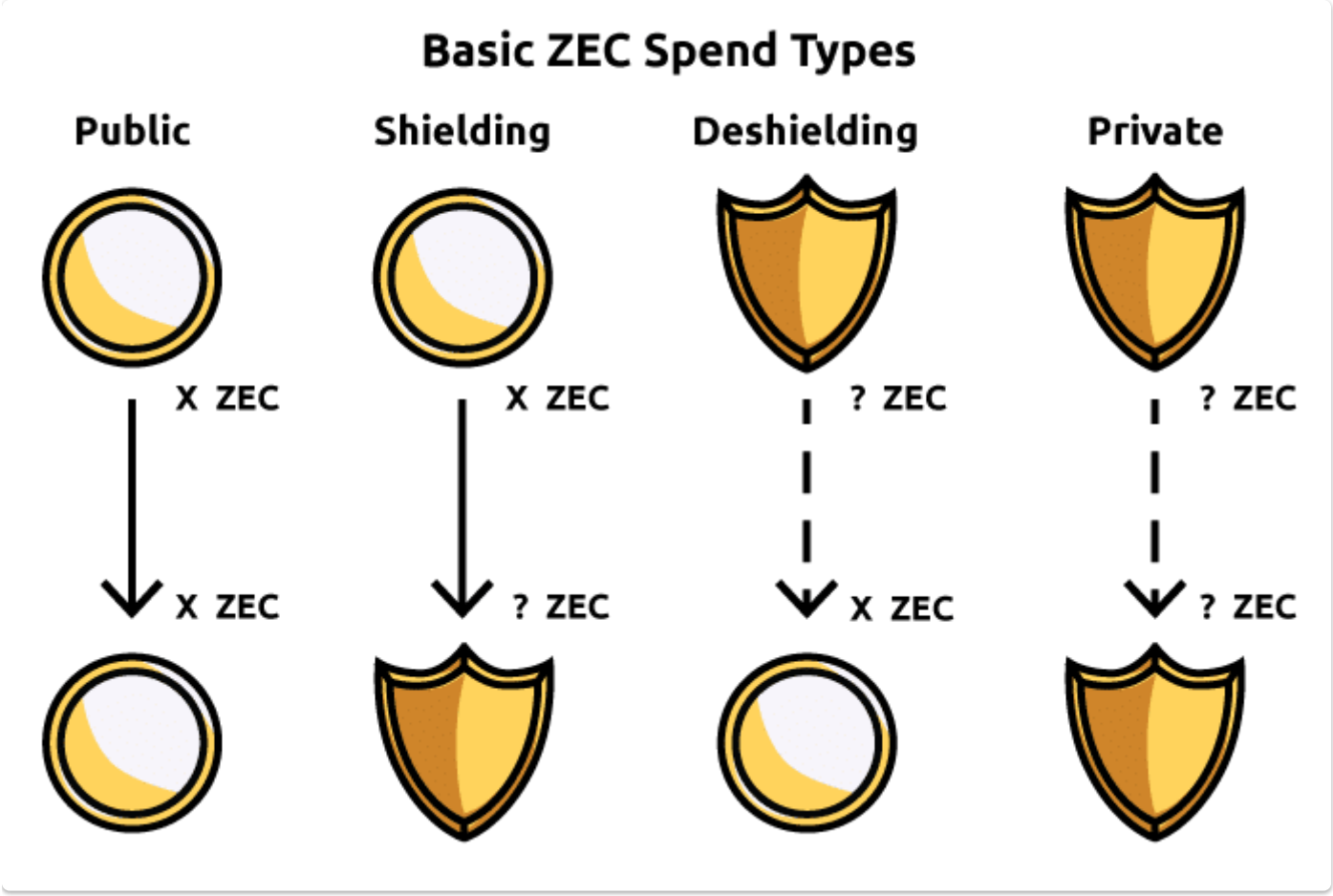
Zcash specification

"Zcash is an implementation of the Decentralized Anonymous Payment scheme Zerocash, with security fixes and improvements to performance and functionality. It bridges the existing transparent payment scheme used by Bitcoin with a shielded payment scheme secured by zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs). It attempted to address the problem of mining centralization by use of the Equihash memory-hard proof-of-work algorithm."

Overview

Value in Zcash is either transparent or shielded. Transfers of transparent value work essentially as in Bitcoin and have the same privacy properties.
(See Block Explorer))





Examples

Public

https://blockchair.com/zcash/transaction/e10d50d127c55db1714af2b420d146c474c2787f95e0cabb5fe8a3879562b770

Shielding

https://blockchair.com/zcash/transaction/de1d78ee310ba2c9fbeb13881f431fdc8b55aa5f79e61dd3c294177401878f11

De-shielding

https://blockchair.com/zcash/transaction/218f5cee4aa8d3469ea0e4cd49fc0b5b60e3d53fcff9392384a2c137fc48ea45

Private

https://blockchair.com/zcash/transaction/35f6674a1691f21aff6a3819467dbba82aaebf061d50c6ac55f39fbeae73b9a6

ZCash Addresses

Addresses which start with "t" behave similarly to Bitcoin, exposing addresses and balances on the blockchain and we refer to these as "transparent addresses".
Addresses which start with "z" or "zc" or "zs" include the privacy enhancements provided by zk proofs and we refer to these as "shielded addresses".
It is possible to send ZEC between these two address types.

Trusted Setup

From ZCash Docs

"In order to ensure the toxic waste did not come into existence, our team designed multi-party computation (MPC) protocols which allowed multiple independent parties to collaboratively construct the parameters. These protocols had the property that, in order to compromise the final parameters, *all* of the participants would have had to be compromised or dishonest.

Through 2018, Zcash had created two distinct sets of public parameters. The first ceremony happened in October 2016 just before the launch of Zcash Sprout. The second was generated in 2018, anticipating the Sapling network upgrade later that year."

Security Problem

Zcash has a serious security problem in 2018
"The counterfeiting vulnerability was fixed by the Sapling network upgrade that activated on October 28th, 2018. The vulnerability was specific to counterfeiting and did not affect user privacy in any way.
Prior to its remediation, an attacker could have created fake Zcash without being detected. **The counterfeiting vulnerability has been fully remediated in Zcash and no action is required by Zcash users**."

On March 1, 2018, Ariel Gabizon, a cryptographer employed by the Zcash Company at the time, discovered a subtle cryptographic flaw in the BCTV14paper that describes the zk-SNARK construction used in the original launch of Zcash. The flaw allows an attacker to create counterfeit shielded value in any system that depends on parameters which are generated as described by the paper.

This vulnerability is so subtle that it evaded years of analysis by expert cryptographers focused on zero-knowledge proving systems and zk-SNARKs. In an analysis Parno15in 2015, Bryan Parno from Microsoft Research discovered a different mistake in the paper.

However, the vulnerability we discovered appears to have evaded his analysis. The vulnerability also appears in the subversion zero-knowledge SNARK scheme of Fuchsbauer17, where an adaptation of BCTV14 inherits the flaw.
The vulnerability also appears in the ADSNARK construction described in BBFR14.
Finally, the vulnerability evaded the Zcash Company's own cryptography team, which includes experts in the field that had identified several flaws in other parts of the system.

ZCash Design

ZCash was developed from the zerocash protocol which is based on Bitcoin.

The UTXO model

Imagine Alice has control of Note1, via her secret key sk_a
Alice
(sk_a) → Note1

She could send the Note1 to Bob by giving him the secret key sk that controls Note1, but then she would still have control of it.
A better idea is
Alice signs a transaction to cancel Note1 and allow Bob to create Note2 of the same value (ignoring refunds for the moment)

Alice
(sk_a) cancel Note1

Bob
(sk_b) → Note2

Then Bob has control of the new Note.

ZCash follows the UTXO model and develops it further,

- to the note we add a random value r as an identifier
- rather than storing these values directly, we store a hash of those values.

In order to cancel the notes we introduce the notion of a nullifier

The commitment / nullifier idea

1. Commitments
   A commitment scheme is defined by algorithms $Commit$ and $Open$
   Given a message m and randomness r, compute as the output a value c

```
c = Commit(m,r).                                                language-js
```

A commitment scheme has 2 properties:
1. Binding. Given a commitment c, it is hard to compute a different pair of message and randomness whose commitment is c. This property guarantees that there is no ambiguity in the commitment scheme, and thus after c is published it is hard to open it to a different value.
2. Hiding. It is hard to compute any information about m given c.

In ZCash Pedersen hashes are used to create the commitments using generator points on an elliptic curve
Given a value v which we want to commit to, and some random number r

commitment $c = v * G_v + r * G_r$
Where G_v and G_r are generator points on an elliptic curve.

2. Nullifiers
   Nullifiers are used to signal that a note has been spent. Each note can deterministically produce a unique nullifier.
   When spending a note,
   1. The nullifier set is checked to ascertain whether that note has already been spent.
   2. If no nullifier exists for that note, the note can be spent
   3. Once the note has been spent its nullifier is added to the nullifier set

Note that the nullifier is unlinkable, knowledge of a nullifier does not give knowledge of the note that produced it.

Shielded value is carried by notes ,which specify an amount and a shielded payment address, which is an address controlling the note.
As in Bitcoin, this is associated with a private key that can be used to spend notes sent to the address; in Zcash this is called a spending key.

Storing details of notes and nullifiers

To each note there is cryptographically associated a note commitment . Once the transaction creating a note has been mined, the note is associated with a fixed note position in a tree of note commitments.

Computing the nullifier requires the associated private spending key.
It is infeasible to correlate the note commitment or note position with the corresponding nullifier without knowledge of at least this key.

An unspent valid note, at a given point on the block chain, is one for which the note commitment has been publically revealed on the block chain prior to that point, but the nullifier has not.

For each shielded input ,
• there is a revealed value commitment to the same value as the input note
• if the value is nonzero, some revealed note commitment exists for this note;

and for each shielded output ,

• there is a revealed value commitment to the same value as the output note
• the note commitment is computed correctly;
• it is infeasible to cause the nullifier of the output note to collide with the nullifier of any other note.

Moving towards secrecy with zero knowledge proofs

Details from Slides

Keys

We have a number of keys developed from the original secret key, which have different roles

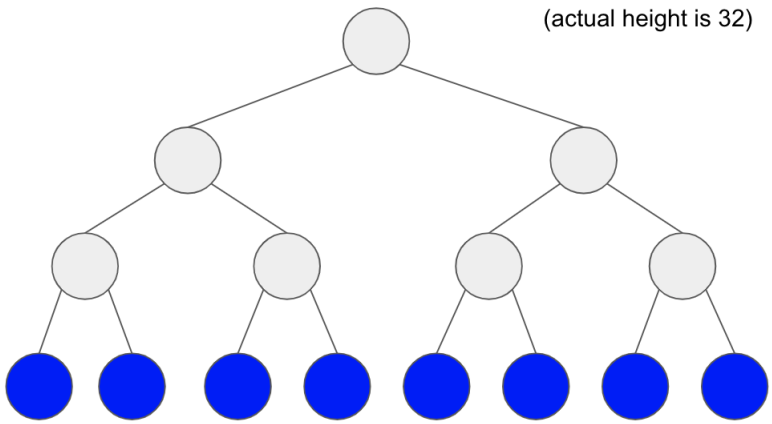
"Screenshot 2022-03-10 at 17.07.57.png" is not created yet. Click to create.

In later versions of ZCash the number of keys have increased.

Storing Notes and Nullifiers in merkle trees

The Global Merkle Tree of Commitment Notes

- Holds all “commitment notes”
  - similar to Bitcoin’s UTXO model
- A Pedersen hash of the **value** of the note, and its **owner**
- **Only additive**
  - Spending a note does not remove it from the tree (nullifier set’s role)
- Like the UTXO set, Miners have to update their local Merkle Tree based on incoming transactions



leaves are note note commitments (cm)

We have a similar tree for the nullifier set.

Transactions

Each transaction has a list of Spend and Output Descriptions
We also create zkps to prove existence of the note in the merkle tree, and ownership of the note

Spend Descriptions spend existing notes , the spender needs to show that

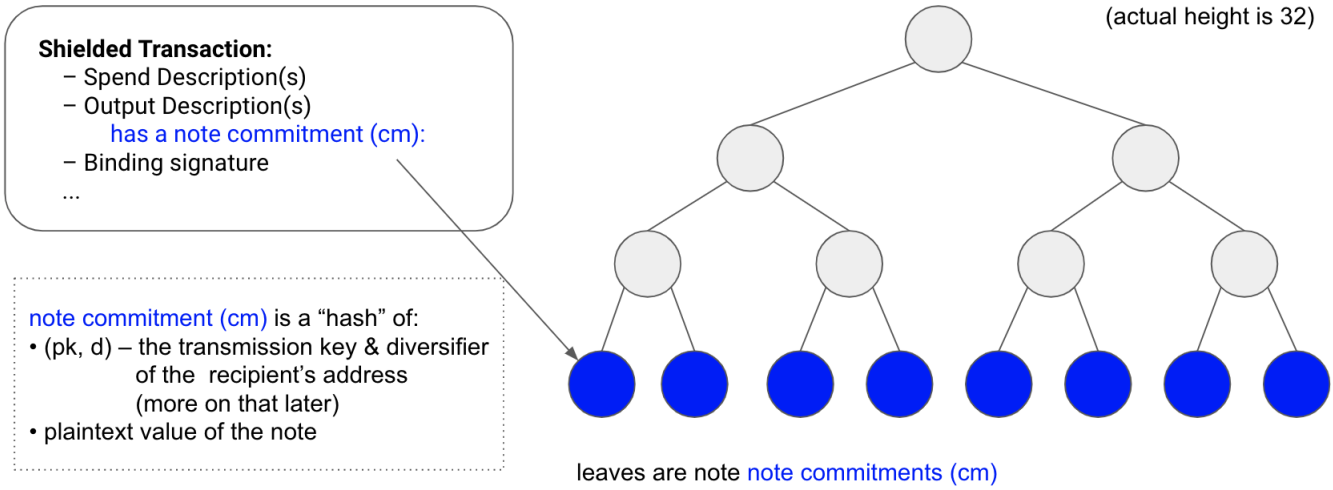
- The note exists
- The spender owns the note
- The note has not been spent before, by computing a nullifier unique to that note and checking this against the nullifier set.

The Spend description commits to a value commitment of the note
and all necessary public parameters needed to verify the proof.
The proof is constructed on private parameters to validate note ownership and spendability

Output Descriptions create new notes

Output Description

Creates a new note



- Only the sender’s outgoing view key and recipient’s incoming view key can decrypt the details
- Only the recipient can spend the new note
- Which note was used is not revealed
- Who the sender, recipient, or the amount is not revealed
- The nullifier is unique to each note, and is revealed when spent

The nullifiers are necessary to prevent double-spending: each note on the block chain only has one valid nullifier, and so attempting to spend a note twice would reveal the nullifier twice, which would cause the second transaction to be rejected.

How does the recipient know that they have a new note ?

A shielded payment address includes a transmission key for a “key-private” asymmetric encryption scheme.
Key-private means that ciphertexts do not reveal information about which key they were encrypted to, except to a holder of the corresponding private key, which in this context is called the receiving key.
This facility is used to communicate encrypted output notes on the block chain to their intended recipient, who can use the receiving key to scan the block chain for notes addressed to them and then decrypt those notes.

The basis of the privacy properties of Zcash is that when a note is spent, the spender only proves that some commitment for it had been revealed, *without revealing which one*.

This implies that a spent note cannot be linked to the transaction in which it was created. That is, from an adversary's point of view the set of possibilities for a given note input to a transaction —its note traceability set — includes all previous notes that the adversary does not control or know to have been spent

The ZKProof

The zk-SNARK circuit has a fixed size, and the maximum number of inputs and outputs per JoinSplit must be fixed in order to achieve that. 2 inputs and two outputs are the minimum needed in order to be able to join and split shielded notes (hence the name "JoinSplit"). This is the same as in the Pour proofs of the original Zerocash design.

Cryptography used

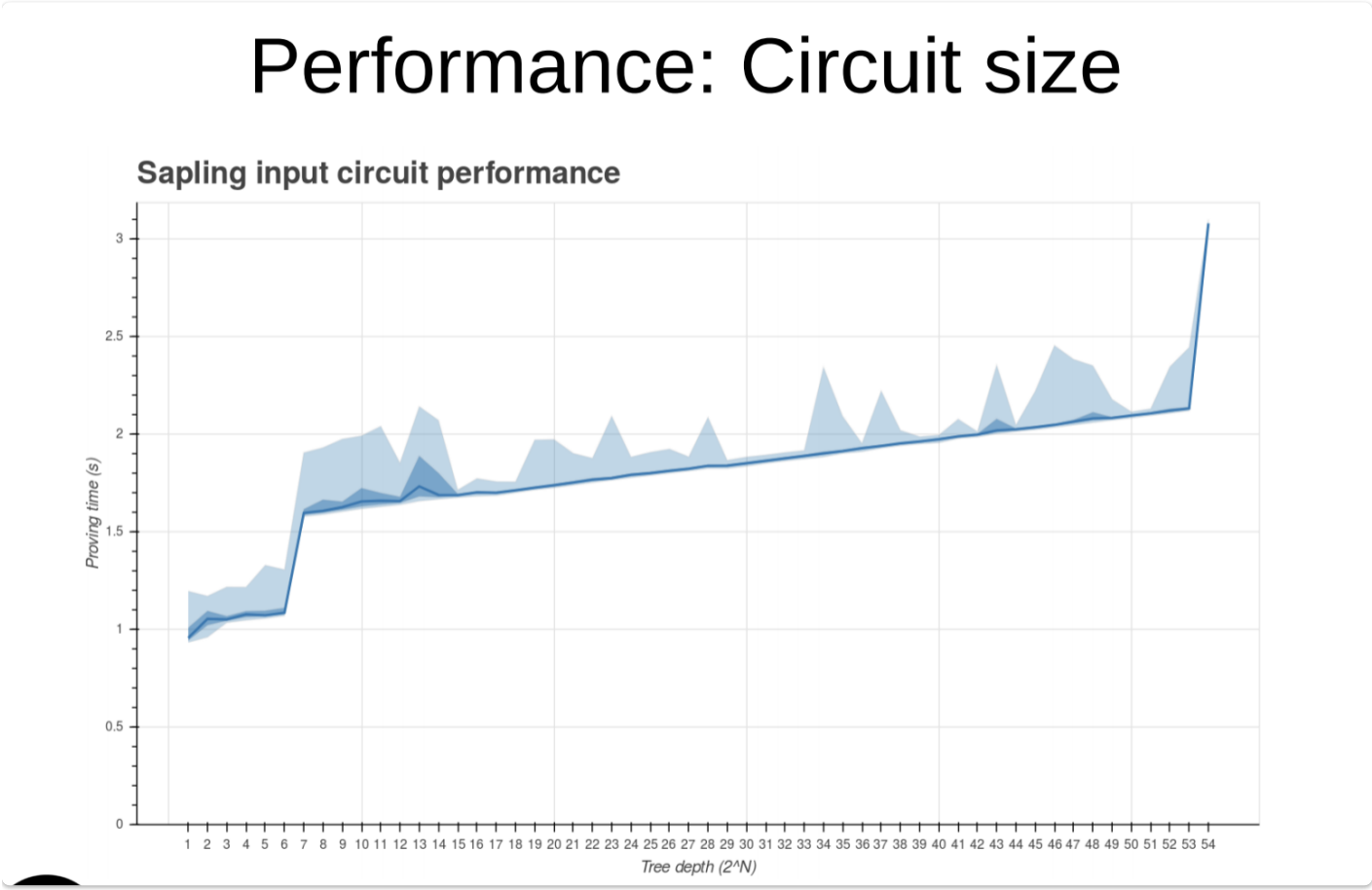
For hash functions, ZCash improved on the Pedersen hash function creating the Bowe-Hopwood Pedersen Hash

Originally ZCash used BLS12-381 for its elliptic curve as optimal for for zkSNARKS with a security bit level of 128 which had an embedded twisted Edwards curve (named Jubjub).
The latest version, Orchard, is using two elliptic curves, Pallas and Vesta, that form a cycle: the base field of each is the scalar field of the other.
In Orchard, we use Vesta for the proof system (playing a similar rôle to BLS12-381 in Sapling), and Pallas for the application circuit (similar to Jubjub in Sapling).

Interaction is in Rust via the Bellman library.

Performance

In Sapling the proving time has been reduced to an average of 2.3s
Each proof requires 3 field elements, and 3 pairing checks.
The Merkle tree is based on Bowe-Hopwood Pedersen hashes, with depth 32, 1369 constraints per level, giving a total of 43,808 constraints.



Performance is not linear in tree depth / circuit size.

Block Explorer

Block Explorer Zchain

Bulletproofs

See paper : [Bulletproofs]

"Screenshot 2022-03-14 at 08.38.07.png" is not created yet. Click to create.

- Bullet proofs have short proofs without a trusted setup.
- Their underlying cryptographic assumption is the discrete log problem.
- They are made non interactive using the Fiat-Shamir heuristic.
- One of the paper's authors referred to them as
  “Short like a bullet with bulletproof security assumptions”
- They were designed to provide confidential transactions for cryptocurrencies.
- They support proof aggregation, so that proving that m transaction values are valid, adds only $O(log(m))$ additional elements to the size of a single proof.
- Pederson commitments are used for the inputs
- They do not require pairings and work with any elliptic curve with a reasonably large subgroup size
- The verifier cost scales linearly with the computation size.

Bulletproofs were based on ideas from Groth16 SNARKS, but changed various aspects.

- They give a more compact version of the inner product argument of knowledge
- Allow construction of a compact rangeproof using such an argument of knowledge
- They generalize this idea to general arithmetic circuits.

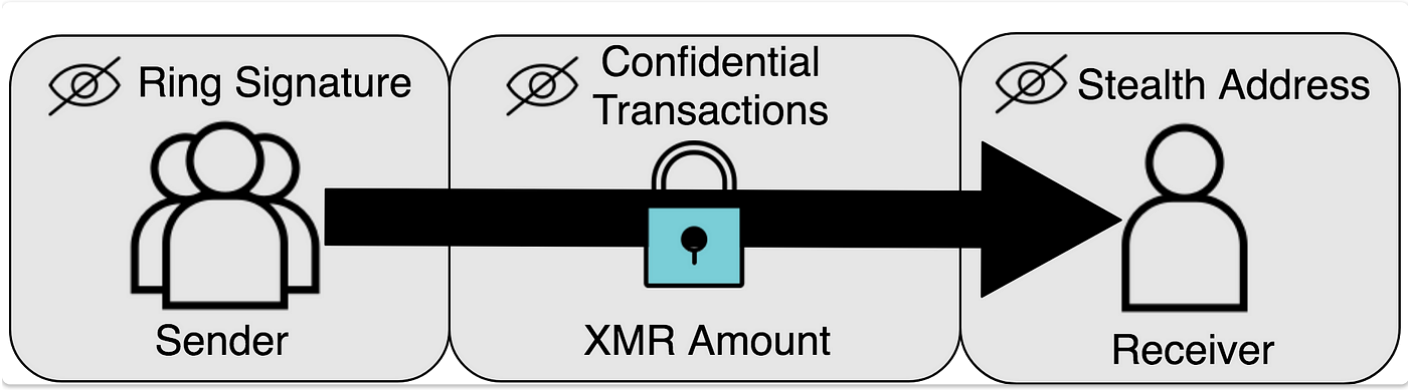
Some use cases for bulletproofs

- Range proofs
- Merkle proofs (accumulators)
- Proof of Solvency

Monero

Initial approach

Initially Monero used ring signatures



This approach had scalability and security issues.
They later moved to using bulletproofs

Overview

Committments to inputs are used to shield the input details

We need to show that the sum of inputs and outputs add up, but there is a potential problem with overflow since we work on a finite field.
This is solved with range proofs to show that the values are in the correct range, without revealing the values.

Monero comment on their move to bulletpoofs :

"With our current range proofs, the transaction is around 13.2 kB in size.
If I used single-output bulletproofs, the transaction reduces in size to only around 2.5 kB
This is, approximately, an 80% reduction in transaction size, which then translates to an 80% reduction in fees as well.

The space savings are even better with multiple-output proofs. This represents a significant decrease in transaction sizes.
Further, our initial testing shows that the time to verify a bulletproof is lower than for the existing range proofs, meaning speedier blockchain validation. "

Mimblewimble / Grin

Docs
- Similar process to ZCash
- Every transaction has to prove two basic things:
- Zero sum - The sum of outputs minus inputs should always equal zero, proving that a transaction did not create new coins, without revealing the actual amounts.
- Possession of private keys - ownership of outputs is guaranteed by the possession of ECC private keys. However, the proof that an entity owns those private keys is not achieved by directly signing the transaction, as with most other cryptocurrencies.

In 2022 implementation of Mimblewimble incorporated into Litecoin. It makes use of a number of technologies in order to ensure privacy:

- *Confidential Transactions* keeps the amount transferred visible only to participants in the transaction, while still cryptographically guaranteeing that no more coins can be spent than are available.
- *CoinJoin* acts like a mixer to conceal the sender of particular transactions, by combining multiple inputs from different parties into a single transaction.
- *Stealth Addresses* conceal the recipient of a transaction, through single-use addresses that cannot be seen on the blockchain without the corresponding viewing key. In Litecoin, these stealth addresses begin "*ltcmweb1*".

Beam

Grin and BEAM are two open-source projects that are implementing the Mimblewimble blockchain scheme. Both projects are building from scratch.
Grin uses Rust while BEAM uses C++

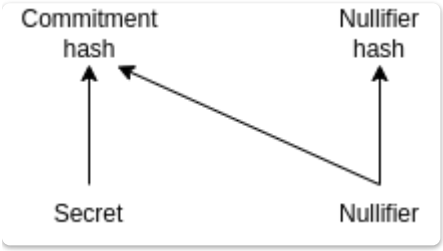
Comparisons between these currrencies

Comparison between Grin and Beam

Comparison between Beam / ZCash / Monero

Tornado Cash

To process a deposit, Tornado.Cash generates a random area of bytes, computes it through the Pederson Hash (as it is friendlier with zk-SNARKs), then send the token and the hash to the smart contract. The contract will then insert it into the Merkle tree.
The random bytes are known to the depositor and are used to form the commitment hash.



To process a withdrawal, 2 pieces of information are used.
The secret random bytes and a nullifier.

Two zero knowledge proofs are provided, to prove the hash of the initial commitment and of the nullifier without revealing any information.
The nullifier forms part of the commitment hash, so it cannot be changed by the spender.
Privacy is sustained as there is no way to link the hashed nullifier to the initial commitment.
Besides, even if the information that the transaction is present in the Merkle root, the information about the exact Merkle path, thus the location of the transaction, is still kept private.

Deposits are simple on a technological point of view, but expensive in terms of gas as they need to compute the hash and update the Merkle tree. At the opposite end, the withdrawal process is complex, but cheaper as gas is only needed for the nullifier hash and the zero-knowledge proof.

TORNADO CASH Verifier Contract

Contract

Example of Tornado Cash used by a hacker
Another article explaining Tornado Cash

Introduction to Aztec

"Aztec is an open source layer 2 network bringing scalability and privacy too Ethereum. Aztec uses zkSNARK proofs to provide privacy and scaling via our zkRollup service."

Aztec give some useful definitions of Privacy, Anonymity and Confidentiality

Privacy: all aspects of a transaction remain hidden from the public or third parties.

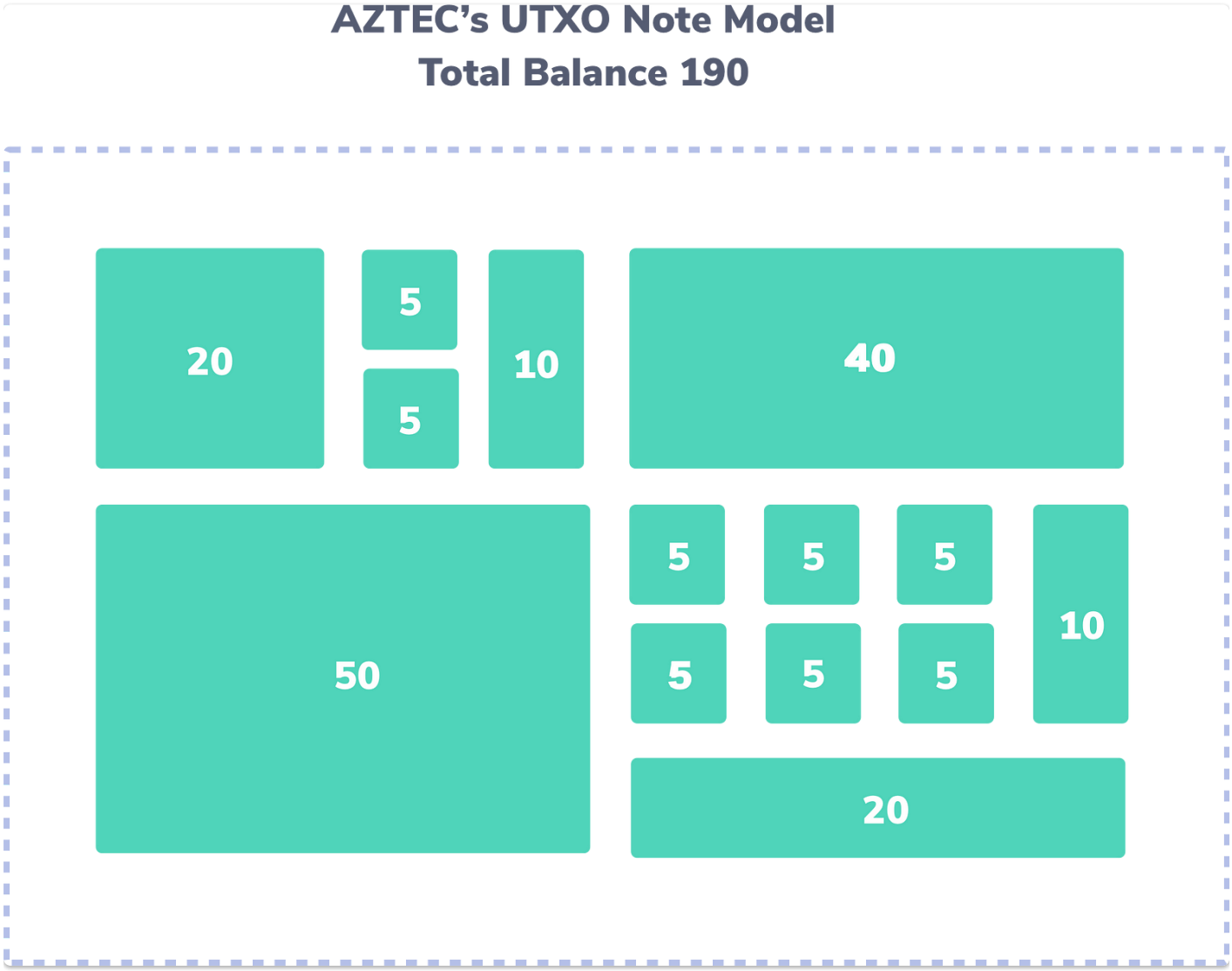
Confidentiality: the inputs and outputs of a transaction are hidden from the public but the transaction parties remain public.

Anonymity: the inputs and outputs of a transaction are public but the transaction graph is obscured from one transaction to the next, preventing the identification of the transaction parties.

From Aztec Documentation
See also yellow paper

The AZTEC protocol was created to enable privacy on public blockchains. It enables logical checks to be performed on encrypted values without the underlying values being revealed to the blockchain.
The inputs and outputs of a transaction are encrypted using a series of zero-knowledge proofs and homomorphic encryption, yet the blockchain can still test the logical correctness of these encrypted statements.

AZTEC follows a UTXO model similar to that of Bitcoin and ZCash.
The core of any AZTEC transaction is a *Note*. The state of notes are managed by a *Note Registry* for any given asset.
The AZTEC protocol does not represent 'value' like a traditional balance, which maps owners to how much they own.
The user's balance of any AZTEC asset is made up of the sum of all of the valid notes their address owns in a given *Note Registry*.



Note Details

A note contains the following public information:

- An AZTEC commitment: an encrypted representation of how much 'value' the note holds
- An Ethereum address of the note's owner

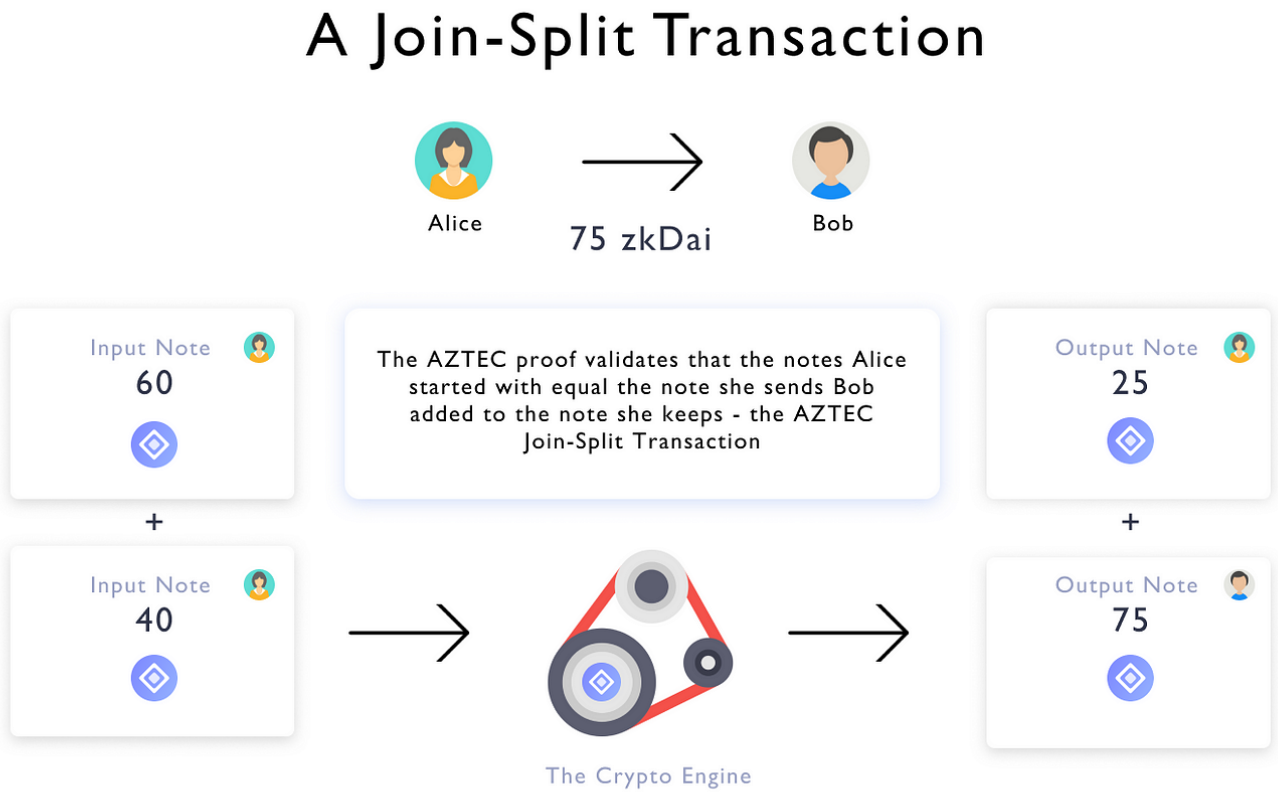
A note has the following private information

- The value of the note
- The note's *viewing key*. Knowledge of the viewing key enables a person to decrypt the note (but not spend it)
  One owner can have multiple notes.

A digital asset that conforms to the AZTEC protocol will contain a note registry, which allows a smart contract to recover the public information of every unspent note that currently exists.

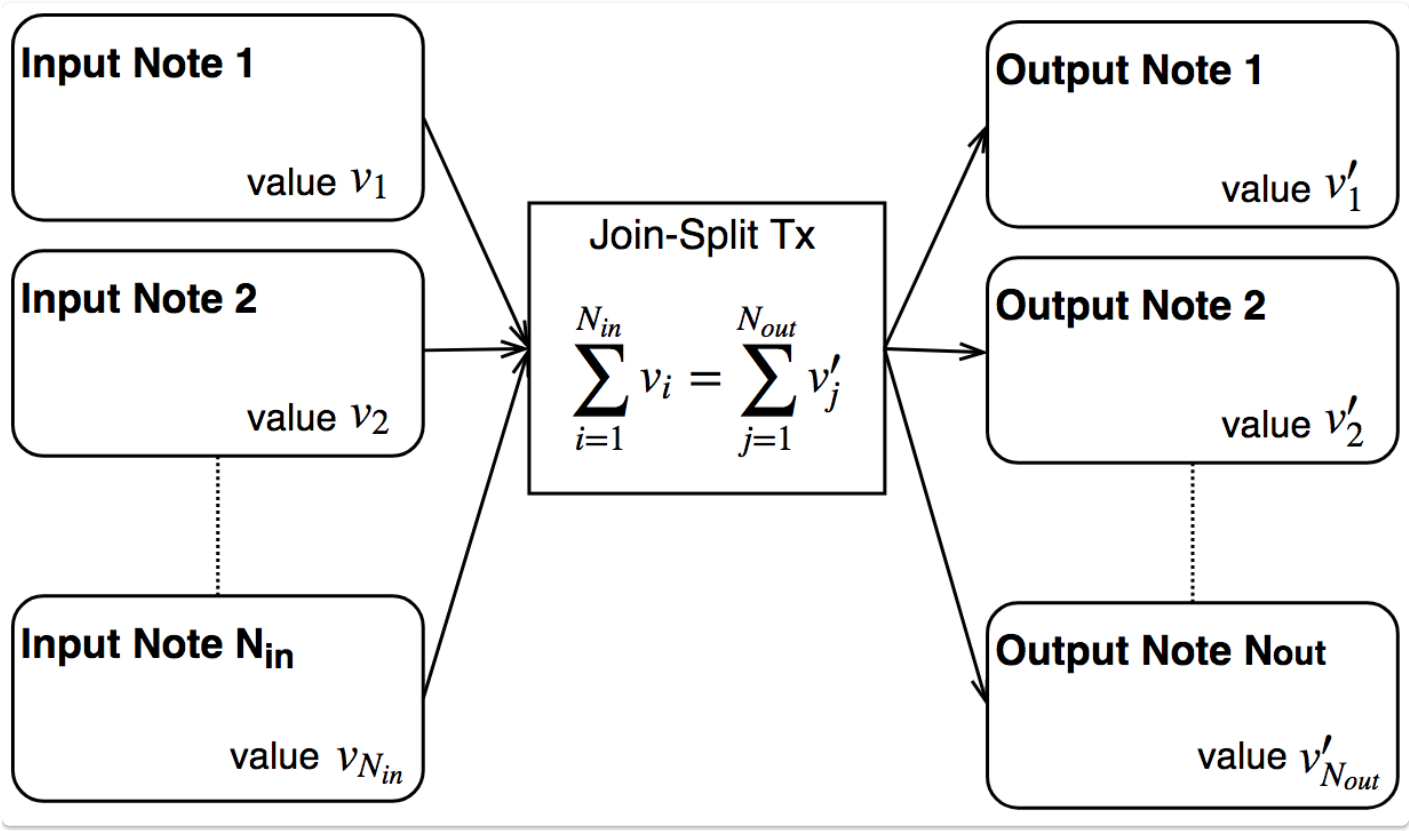
An AZTEC note owner can 'spend' their notes in a join-split style confidential transaction.
In this transaction, the note owner will destroy some unspent AZTEC notes they own. In their place, they will create a set of new notes. The sum of the *values* of the new notes must be equal to the sum of the *values* of the old notes.

A Join Split transaction
From (https://medium.com/aztec-protocol/aztec-how-the-ceremony-works-5c23a54e2dd9)



Suppose Alice has a cluster of notes summing to 100zkDAI, and wants to send 75 zkDAI to Bob.

Alice takes notes of size 60 zkDAI and 40 zkDAI (these are now called input notes — she needs both, because neither can cover the 75 zkDAI she's sending to Bob), and she will create two output notes — 75 zkDAI for Bob, 25 zkDAI as change.
In general



Note this is exactly how the UTXO model works— but AZTEC transactions need to be confidential. And Ethereum needs to validate them
i.e. check that $60 + 40 = 75 + 25$ in our example.

This is possible using the homomorphic additive property of elliptic curves

Alice would create an AZTEC zero-knowledge proof that proves this relationship in zero-knowledge (i.e. Alice does not reveal to anybody how much the notes are actually worth, just that the balancing relationship holds).

The AZTEC token smart contract will then validate this zero-knowledge proof, destroy Alice's input notes and then create the output notes in its note registry.

When Alice is creating Bob's notes, she constructs note viewing keys that Bob will be able to identify, via a non-interactive secret-sharing protocol.
Bob is dependent on Alice to act 'trustfully' in this regard and not provide viewing keys that can be decoded by observers. This is already implicitly required—after all Alice could broadcast to the world how much she is sending Bob if she did not want the transaction to be confidential.

To achieve interoperability with other DApps, all AZTEC assets share a common trusted setup and their state is managed by a single smart contract, the AZTEC Cryptography Engine or ACE

Example Confidential Transaction
https://etherscan.io/tx/0xf9a101682c637f7741f281c858527d17036f4df284b7064bd1ca44531ab88374

Anonymity

AZTEC notes have 'owners' defined by Ethereum addresses.
On the surface, note ownership is not anonymous (e.g. people can see my ethereum address has a zero-knowledge DAI note); the AZTEC protocol includes a Monero-style stealth-address protocol to derive Ethereum addresses that are single-use and cannot be linked to any other Ethereum address (e.g. if you have an AZTEC wallet, I can 'send' a note to an Ethereum address you control, but nobody but you and me will know this is the case).
The protocol supports both stealth addresses (which require a specific wallet to work; you need two public/private key pairs so a regular Ethereum account won't work) and regular Ethereum addresses (which are not anonymous — if you own a note everybody will be able to see that).

Accounts

An account is just a private/public key pair until it is registered
Before an account is registered, the private key is used to decrypt account notes as well as send value notes.
The account will not have any registered spending keys or an account alias until is is registered.
Once an account is registered, value notes that are sent to the account can either be marked to be spent by the account private key or the spending keys.
It is a best practice for a sender to mark value notes as spendable by the spending keys when an account is registered.

Account Registration

To register a new account, you need to choose an alias and a new spending public key. Optionally, you can include a recovery account public key and a deposit.

When you use an unregistered account, your notes are marked as spendable by the account key. It's the sender that defines whether notes are marked spendable with the account key.

A sender can check whether an account has registered spending keys before specifying the spending key.

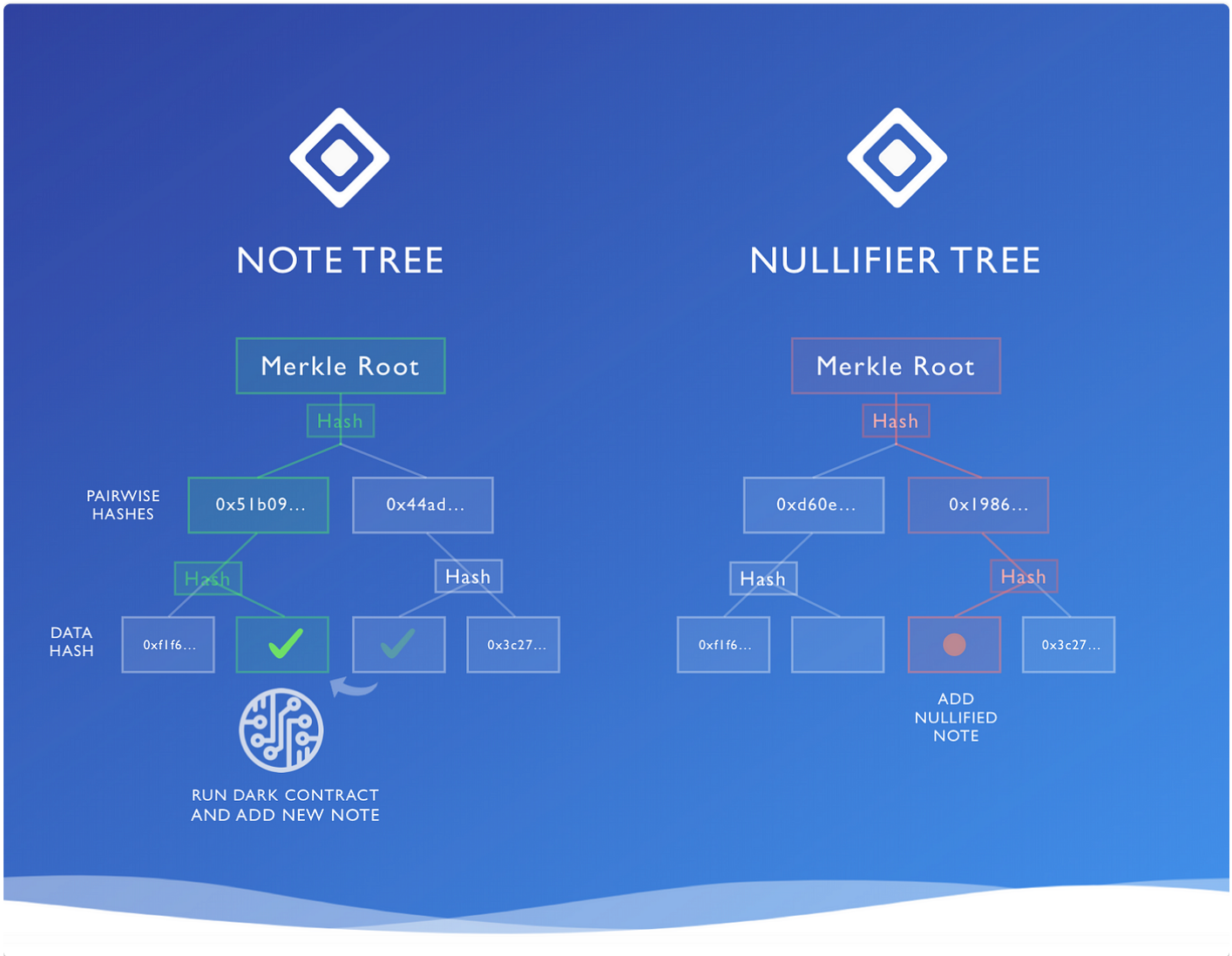
Account Alias

The main privacy account public key is associated with a human-readable alias when the account registers a new signing key.
The alias can be anything (20 alphanumeric, lowercase characters or less) as long as it hasn't been claimed yet.

Data Structures and nullifiers

The above is an abstraction of the process, in order to do this in practice 2 merkle trees are used

- A Note Tree of all output notes ever created, and
- A Nullifier Tree keeping copies of the spent notes
  The idea is — instead of deleting a note from the Note Tree, you need to check whether that note also turns up in the Nullifier Tree to work out if it's already spent. If it's not there, it's still spendable.



Hashes required for this :

- Note Tree: 30 hashes to add a new output note
- Nullifier Tree: 30 hashes to add a note, marking it as spent
- Total: 60 Hashes

Each SHA-256 hash in PLONK requires ~27,000 gates for a 64 byte input, so
60 hashes consume ~1.6m gates.

As with Monero, because we are using finite fields we face the problem of checking the range of the inputs, for this we need range proofs.

For more details of how the proofs are constructed from the details in the trusted setup see this article

zk.money (ZK-ZK-rollups)

Aztec's privacy architecture can enable simple asset transfers, allowing users to privately send $DAI, $ETH, and $renBTC.

Private ZK-rollups provide the scaling benefits of rollups, in addition the transaction inputs/outputs are encrypted.
The zero-knowledge proof that proves the correctness of every transaction also proves that the encrypted data was correctly derived from the non-encrypted 'plaintext' data. But the plaintext is known only to the users that constructed their private transactions.

The rollup cannot simply process a list of transactions like before, it must verify a list of zero-knowledge proofs that each validate a private transaction. (This extra layer of zero-knowledge proof verification is why they're called 'ZK-ZK-rollups').

The result is reduced-cost transactions with full transaction privacy.
Both the identities of senders/recipients are hidden, as well as the values being transferred.

Despite this, users of the protocol can have complete confidence in the correctness of transactions (no double spending etc), because only legitimate transactions can produce a valid zero-knowledge proof of correctness.

The architecture is composed of two programs that are encoded into ZK-SNARK 'circuits': A privacy circuit and a rollup circuit.

The privacy circuit proves the correctness of a single private transaction. It is constructed by users that want to send private transactions, directly on their hardware to ensure no secrets are leaked.

The rollup circuit validates the correctness of a batch of privacy proofs (currently 128) and updates the rollup's database with the new encrypted transaction data.

The rollup proofs are constructed by a rollup provider, a 3rd party that has access to significant computing resources (for the moment, Aztec are the rollup provider, later Aztec will decentralize the rollup service.
The rollup provider is completely untrusted. They do not have access to any user data and only see the encrypted outputs of privacy proofs. This also makes it impossible to launch selective censorship attacks, because all transactions look like uniform random numbers.

The way Aztec worked under the hood prior to this most recent upgrade is:

- A proof is generated client-side in-browser
- 28 client proofs are then aggregated into an “inner” rollup proof
- 4 inner rollup proofs are then aggregated into an “outer” rollup proof

That “outer” rollup proof is then verified in what we call the root rollup circuit — the circuit that establishes the validity of all the underlying work that goes into ensuring execution on Aztec happened as expected. Then that final proof gets posted on-chain.

For the release of Aztec Connect SDK, the outer rollup's capacity has increased to 32 inner proofs by optimizing the outer rollup circuit.
In total we get : 28 * 32 = 896.

Aztec Connect

From article

Aztec Connect allows users to bridge private assets to mainnet for a DeFi interaction and return to Aztec in the same transaction.

Aztec Connect serves as a bridge to Ethereum, allowing users to bring privacy-shielded zk-assets on Aztec to public DeFi protocols on Ethereum.

As a result, users save 80–90% on gas fees with privacy thrown in for free.

Users deposit funds into Aztec's Layer 1 rollup contract, and privacy-shielded notes are minted by the Aztec system, identified by their zk- prefixes (e.g. zkETH and zkDAI).

Previously, functionality and usability of zk-assets was limited to internal-to-Aztec private sends and private withdrawals to Ethereum L1 addresses.

Now, users can do *any* DeFi transaction supported by an Aztec Connect Bridge Contract

- a 50 to 100-line interface allowing Aztec's roll-up to interact with a given Layer 1 smart contract.

Looking at a Uniswap swap

A basic Uniswap swap, which costs ~130,000 gas

Because the Aztec rollup supports large batch sizes, up to 896 transactions at launch, courtesy of Flashbots, the cost of validating Aztec zero-knowledge proofs is amortized across many users.
At current proof construction costs, this is just 1,875 gas per transaction.

Say 100 users want to execute the same swap on Uniswap
Splitting the cost of the Uniswap transaction and cost of posting data on Ethereum, they each pay 15,762 gas.

In total, the cost of a Uniswap transaction becomes just 17,637 gas,
an 86% savings over L1.
Swaps become 7.4x cheaper, with privacy as a bonus.

Aztec Connect vastly expands Aztec Network's capabilities at launch, adding whitelisted DeFi functionality with select partners.
Any developer looking to integrate Aztec to an existing DeFi application can write an Aztec Connect Bridge Contract.

To contribute see repo

Standardising Confidential Tokens.

EIP-1724

This EIP defines the standard interface and behaviours of a confidential token contract, where ownership values and the values of transfers are encrypted.

```
  interface zkERC20 {
   event CreateConfidentialNote(address indexed _owner, bytes _metadata);
   event DestroyConfidentialNote(address indexed _owner, bytes32 _noteHash);

   function cryptographyEngine() external view returns (address);
   function confidentialIsApproved(address _spender, bytes32 _noteHash) external
view returns (bool);
   function confidentialTotalSupply() external view returns (uint256);
   function publicToken() external view returns (address);
   function supportsProof(uint16 _proofId) external view returns (bool);
   function scalingFactor() external view returns (uint256);

   function confidentialApprove(bytes32 _noteHash, address _spender, bool _status,
bytes _signature) public;
   function confidentialTransfer(bytes _proofData) public;
   function confidentialTransferFrom(uint16 _proofId, bytes _proofOutput) public;
}
```

compare this to the ERC20 interface

```
  interface IERC20 {

function totalSupply() external view returns (uint256);
function balanceOf(address who) external view returns (uint256);
function allowance(address owner, address spender)
external view returns (uint256);

function transfer(address to, uint256 value) external returns (bool);
function approve(address spender, uint256 value)
external returns (bool);

function transferFrom(address from, address to, uint256 value)
external returns (bool);

event Transfer(
address indexed from,
address indexed to,
uint256 value
);
```

```
event Approval(
address indexed owner,
address indexed spender,
uint256 value
);

}
```