Lesson 6 - Confidential Tokens

Today's topics

- ZCash
- Bulletproofs
- Monero
- Tornado Cash
- Aztec Introduction
- Confidential Token Standards

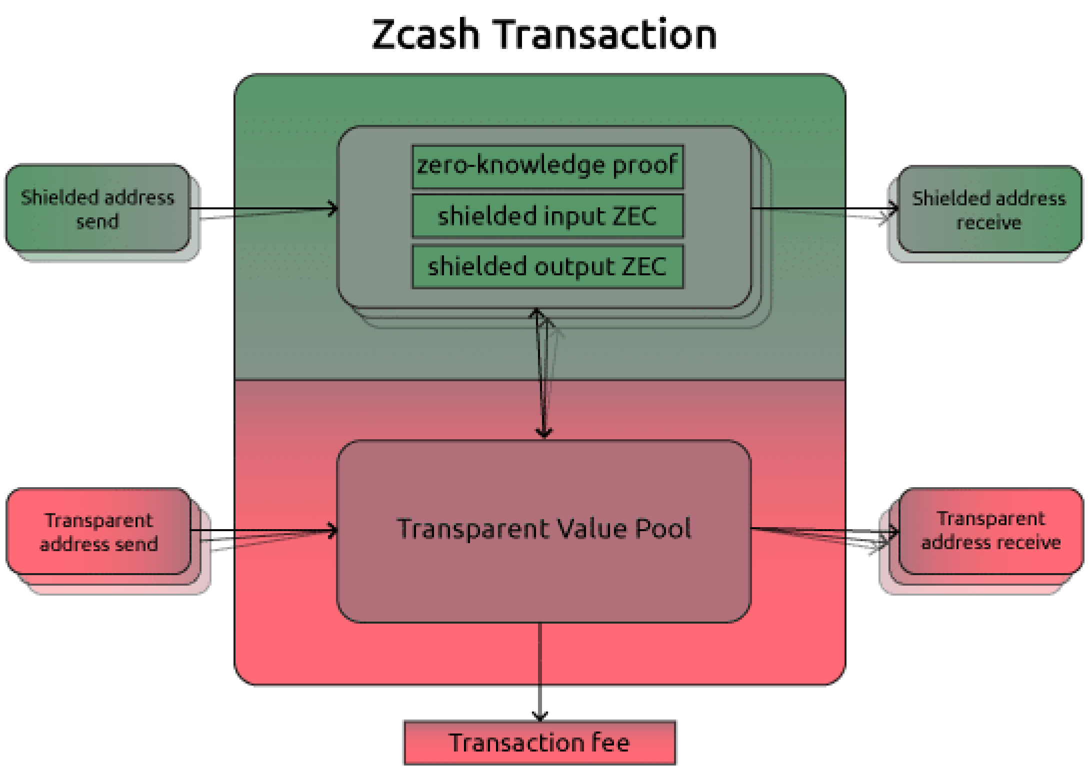
ZCash

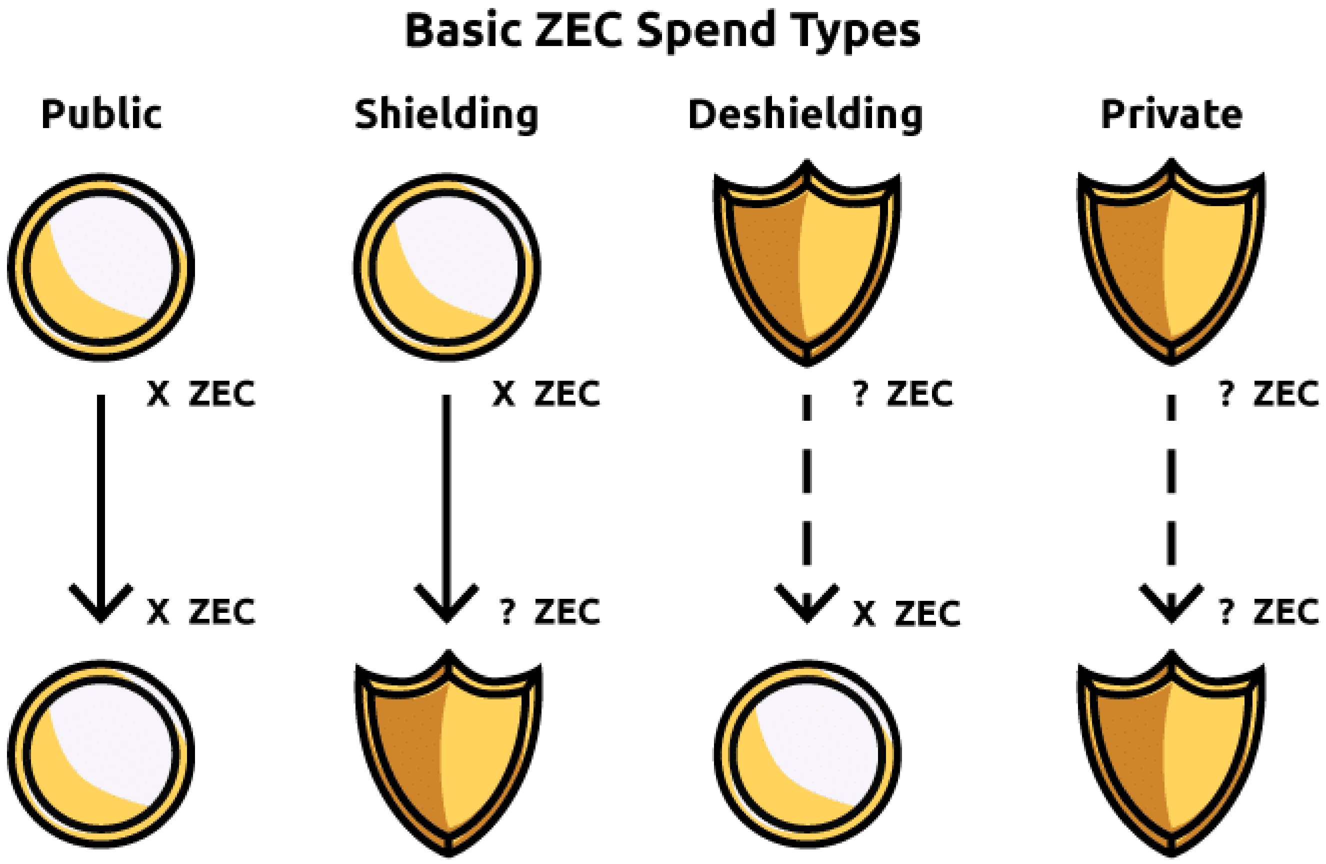
Zcash specification

"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).
```

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 ,

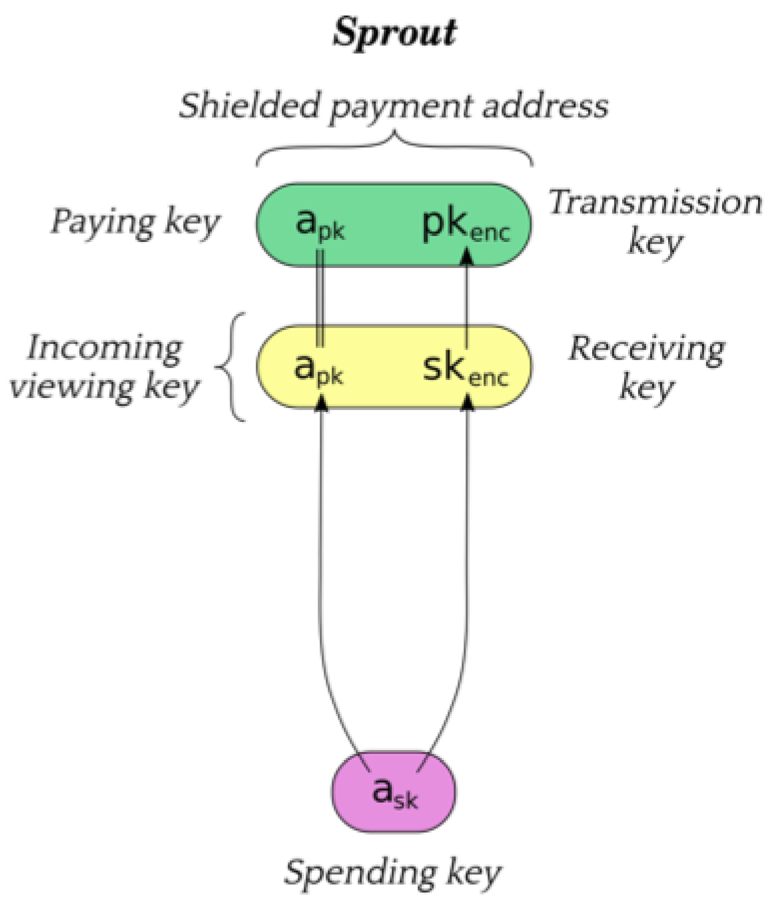
• there is a revealed value commitment to the same value as the output note
• the note commitment is computed correctly;
• it is infeasible to cause the nullifier of the output note to collide with the nullifier of any other note.

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

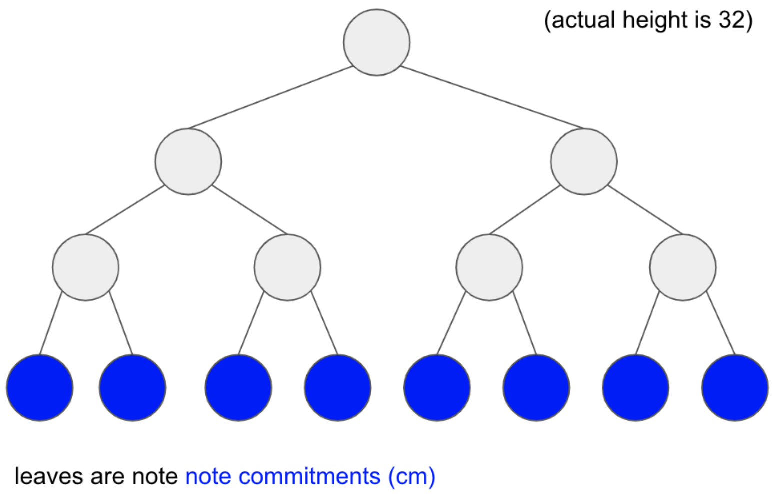


In later versions of ZCash the number of keys have increased.

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



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

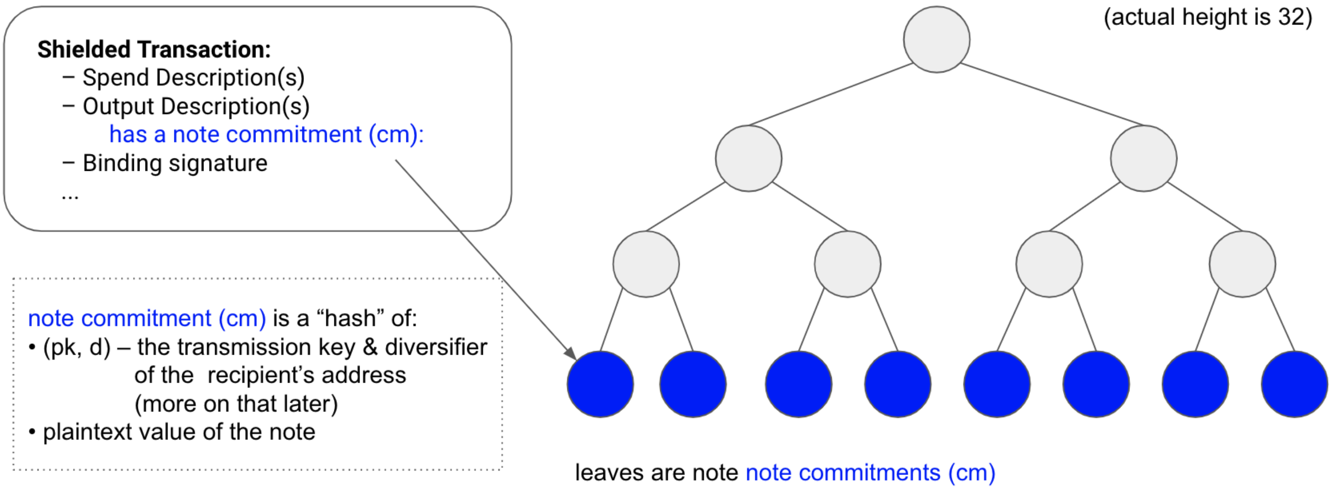
- The note exists
- The spender owns the note
- The note has not been spent before, by computing a nullifier unique to that note and checking this against the nullifier set.

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

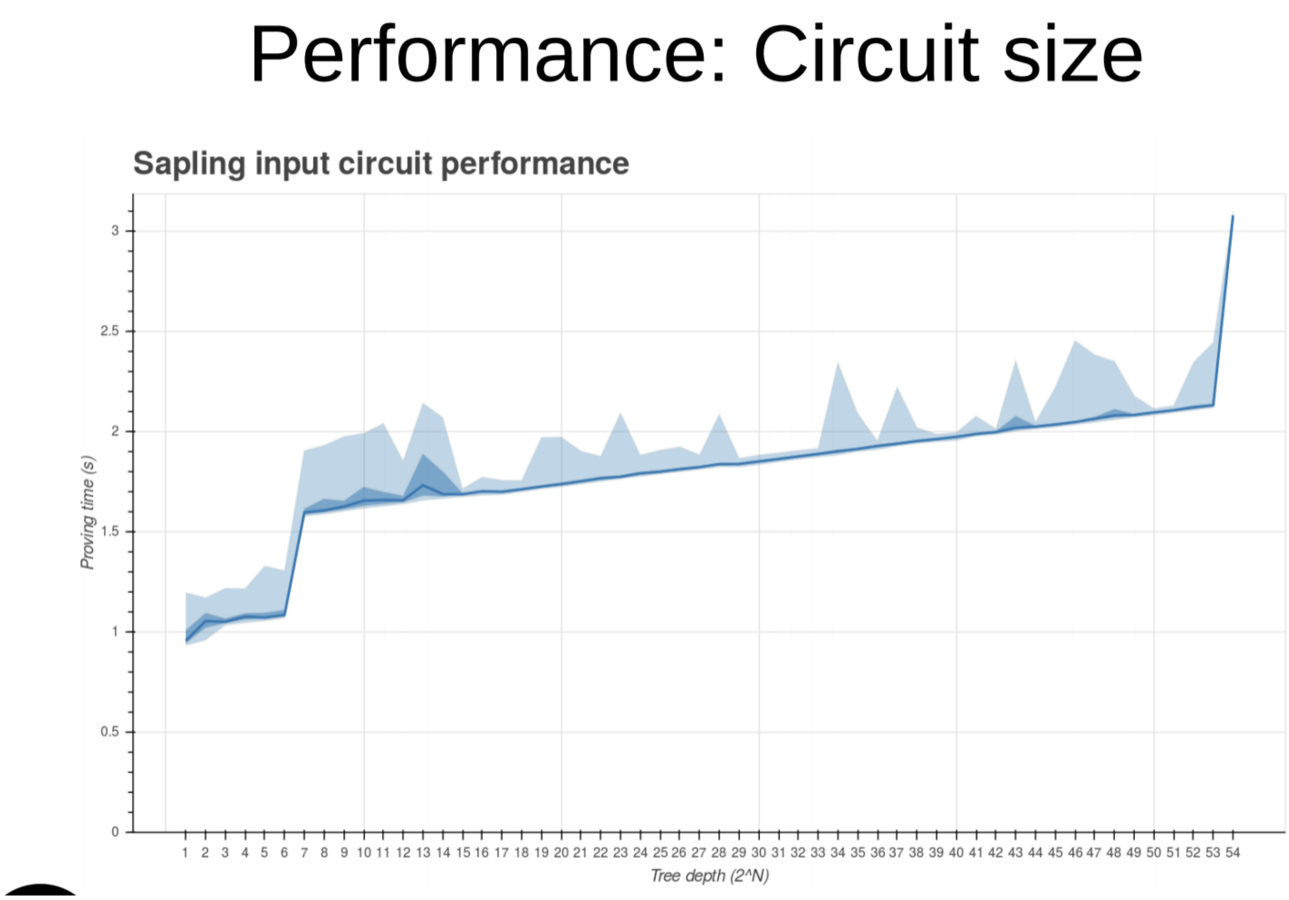
For hash functions, ZCash improved on the Pedersen hash function creating the Bowe-Hopwood Pedersen Hash

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](#)

Overview

Bulletproofs are short non-interactive zero-knowledge proofs that require no trusted setup.

See paper : Bulletproofs

- 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 generalise this idea to general arithmetic circuits.

Some use cases for bulletproofs

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

Comparison

Comparison of the most popular zkp systems

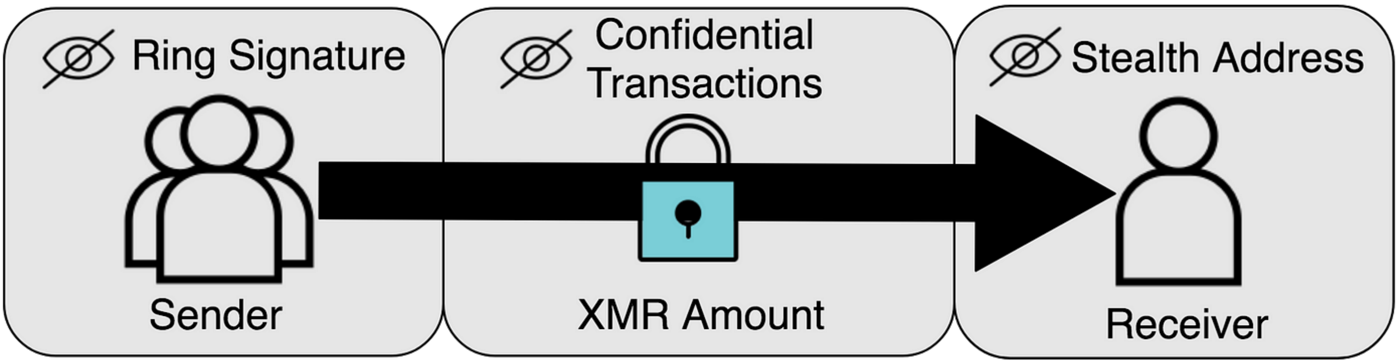
	SNARKs	STARKs	Bulletproofs
Algorithmic complexity: prover	O(N * log(N))	O(N * poly-log(N))	O(N * log(N))
Algorithmic complexity: verifier	~O(1)	O(poly-log(N))	O(N)
Communication complexity (proof size)	~O(1)	O(poly-log(N))	O(log(N))
- size estimate for 1 TX	Tx: 200 bytes, Key: 50 MB	45 kB	1.5 kb
- size estimate for 10.000 TX	Tx: 200 bytes, Key: 500 GB	135 kb	2.5 kb
Ethereum/EVM verification gas cost	~600k (Groth16)	~2.5M (estimate, no impl.)	N/A
Trusted setup required?	YES 😒	NO 😄	NO 😄
Post-quantum secure	NO 😒	YES 😄	NO 😒
Crypto assumptions	DLP + secure bilinear pairing 😒	Collision resistant hashes 😄	Discrete log 😏

Framework	**Arithmetization**	**Commitment Scheme**	**Field**	**Other Configs**
Circom + snarkjs / rapidsnark	**R1CS**	**Groth16**	**BN254 scalar**	
gnark	**R1CS**	**Groth16**	**BN254 scalar**	
Arkworks	**R1CS**	**Groth16**	**BN254 scalar**	
Halo2 (KZG)	**Plonkish**	**KZG**	**BN254 scalar**	
Plonky2	**Plonk**	**FRI**	**Goldilocks**	**blowup factor = 8 proof of work bits = 16 query rounds = 28 num_of_wires = 60 num_routed_wires = 60**
Starky	**AIR**	**FRI**	**Goldilocks**	**blowup factor = 2 proof of work bits = 10 query rounds = 90**
Boojum	**Plonk**	**FRI**	**Goldilocks**	

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++

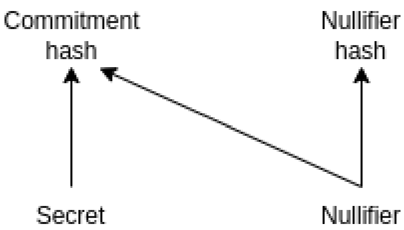
Comparisons between these currrencies

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

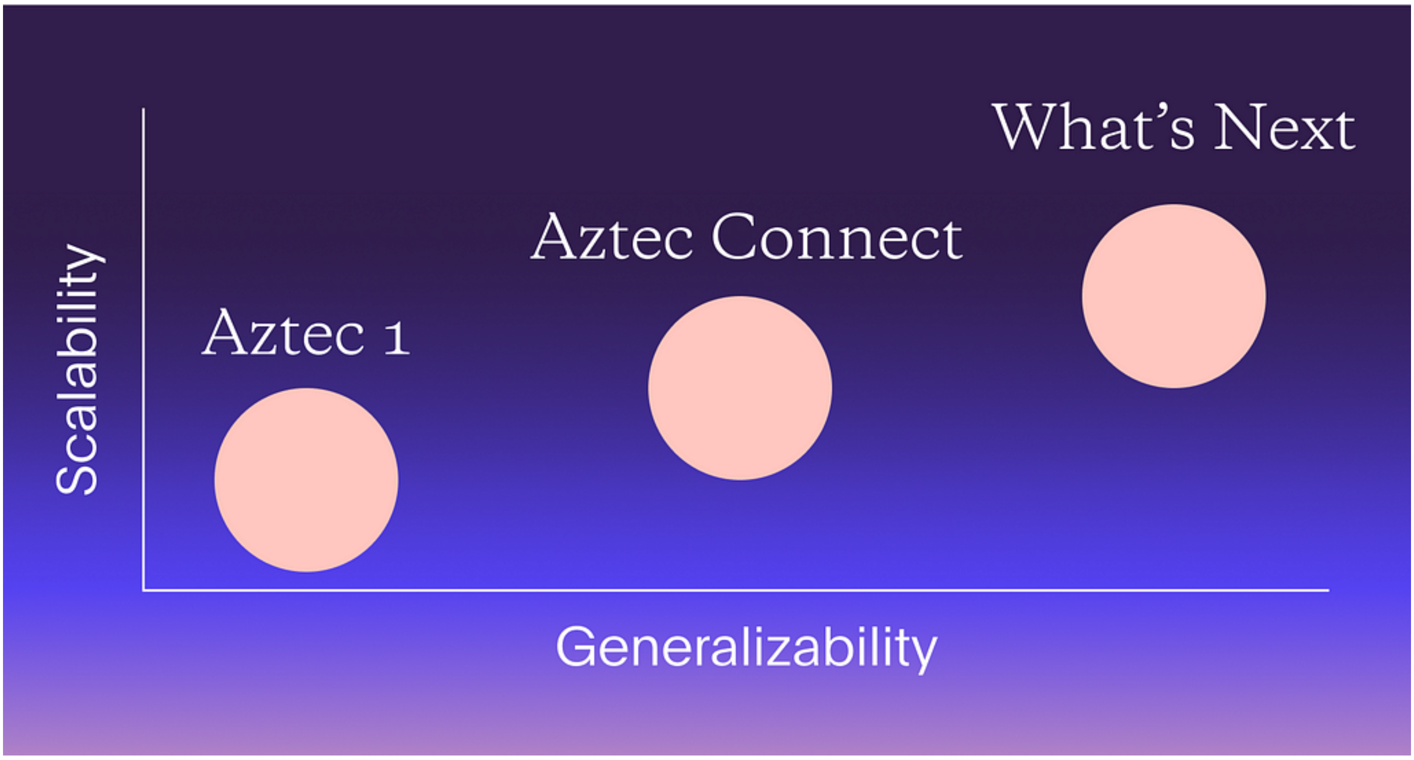
Introduction to Aztec

See Blog
Aztec has recently increased their scope to be a more general privacy solution.
Previous networks are being phased out

Aztec Connect and zk.money

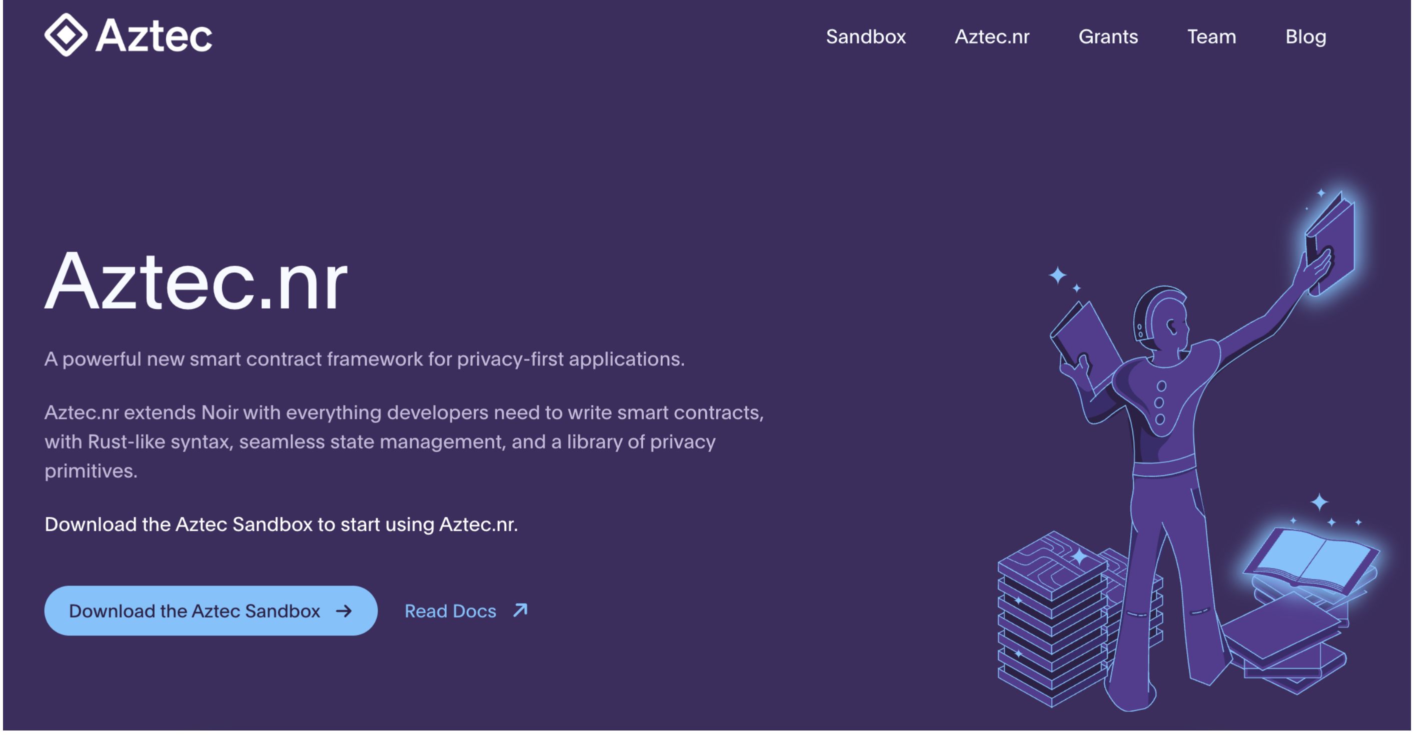
See Annnouncement and zk.money details
This stopped taking deposits in early 2023
Their timetable is

- **13th—31st March:** Aztec Connect normal functionality
- **31st March 2023 at 2PM UTC:** Deposits disabled
- One-year withdrawal window (March 13th, 2023—March 31st, 2024)
- **March 31st, 2024:** Last Aztec rollup sent from Aztec sequencer
- **March 31st, 2024 onwards:** users will have to run withdrawal software or rely on community-run sequencers



"The first attempt at on-chain privacy using a zkRollup was Aztec 1. Aztec 1 was slow, inefficient, costly, and limited in its functionality to basic private transfers."

The 'next step' has started with Aztec.nr



This framework is based on the Noir language.

Taken from Docs and an introduction from Zac

Private Smart Contracts on Aztec

A smart contract on Aztec is a collection of functions, written as ZK-SNARK circuits. These circuits can have different modes of execution:

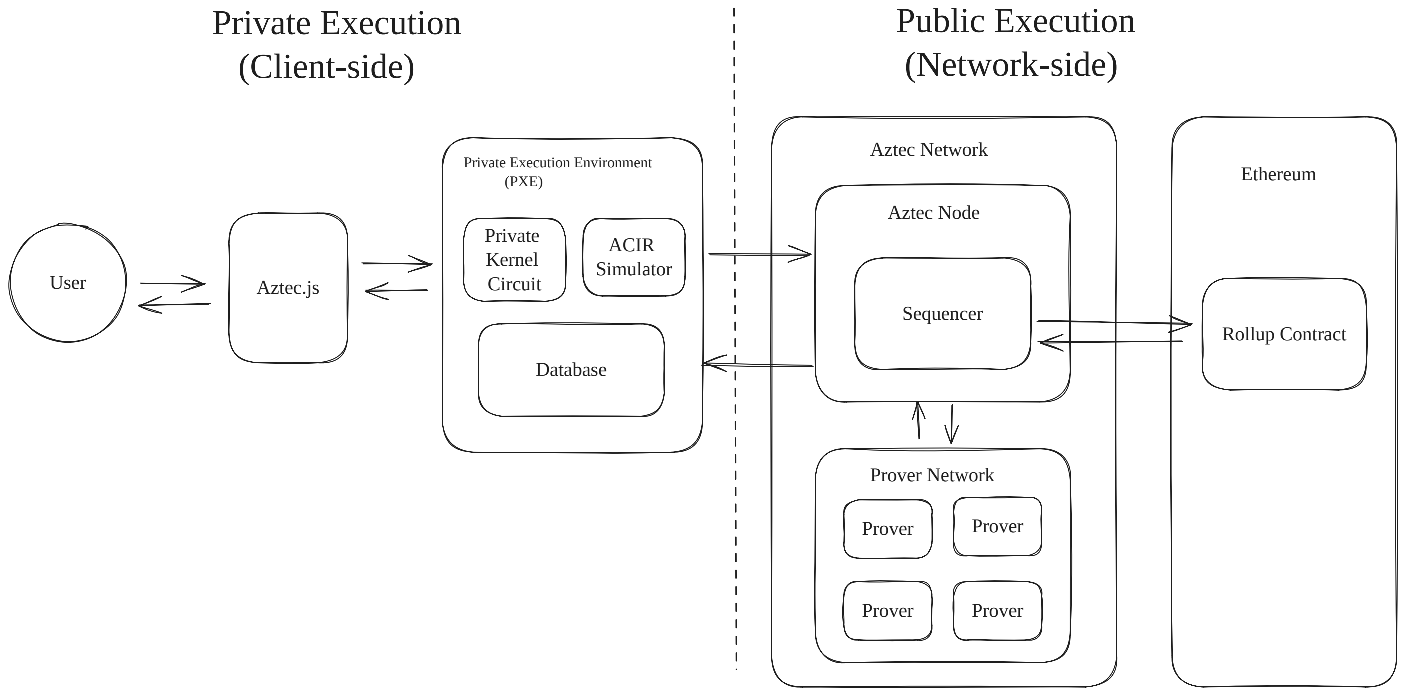
1. Secret Functions -- can read and write private state, read historic public state, consume or send messages to / from L1, and read Ethereum state. Can call other secret functions in the same contract, or other contracts. They can call public functions.
2. Public Functions -- can read and write public state, write private state, consume or send messages to / from L1 and read Ethereum state. Can call other public functions on the same or other contracts.
3. Portal Contracts -- these are contracts on L1 that can receive messages from L2 or allow messages to be sent to L2 from L1 contracts.

Using these different modes of execution, developers can build applications with user privacy, data privacy and code privacy.

User privacy -- transactions may not reveal information about the sender or the recipient.

Data privacy -- transactions may not reveal information about the payload of the transaction, e.g the asset or value being transacted.

Code privacy -- transactions may not reveal the program logic.



Private Execution Environment

The PXE provides a secure environment for the execution of sensitive operations, ensuring private information and decrypted data are not accessible to unauthorized applications. It hides the details of the state model from end users, but the state model is important for Aztec developers to understand as it has implications for private/public execution and L1/L2 communication. The PXE also includes the ACIR Simulator for private executions and the KeyStore for secure key management.

Procedurally, the PXE sends results of private function execution and requests for public function executions to the sequencer, which will update the state of the rollup.

Sequencer

The sequencer aggregates transactions into a block, generates proofs of the state updates (or delegates proof generate to the prover network) and posts it to the rollup contract on Ethereum, along with any required public data for data availability.

UTXO Model

The goal of the Aztec.nr smart contract library is to abstract the UTXO model away

This is achieved with two main features:

1. Users sign over transactions, not over specific UTXO's
2. Aztec.nr contracts support developer defined `unconstrained` getter functions to help dApp's make sense of UTXO's. e.g `getBalance()`. These functions can be called outside of a transaction context to read private state.

State Model

Aztec has a hybrid public / private state model.
From the Docs

Private state must be treated differently from public state and this must be expressed in the semantics of Aztec.nr.

Private state is encrypted and therefore is "owned" by a user or a set of users (via shared secrets) that are able to decrypt the state.

Private state is represented in an append-only database since updating a record would leak information about the transaction graph.

The act of "deleting" a private state variable can be represented by adding an associated nullifier to a nullifier set. The nullifier is generated such that, without knowing the decryption key of the owner, an observer cannot link a state record with a nullifier.

Modification of state variables can be emulated by nullifying the a state record and creating a new record to represent the variable. Private state has an intrinsic UTXO structure and this must be represented in the language semantics of manipulating private state.

Sandbox

https://docs.aztec.network/dev_docs/getting_started/main

sandbox node at http://aztecnode.extropy.io

Execution contexts

Transactions are initiated in the private context, then move to the L2 public context, then to the Ethereum L1 context.

Step 1. Private Execution

Users provide inputs and execute locally on a their device for privacy reasons. Outputs of the private execution are commitment and nullifier updates, a proof of correct execution and any return data to pass to the public execution context.

Step 2. Public Execution

This happens remotely by the sequencer, which takes inputs from the private execution and runs the public code in the network virtual machine, similar to any other public blockchain.

Step 3. Ethereum execution

Aztec transactions can pass data to Ethereum contracts through the rollup via the outbox. The data can consumed by Ethereum contracts at a later time, but this is not part of the transaction flow for an Aztec transaction. The technical details of this are beyond the scope of this tutorial, but we will cover them in an upcoming piece.

Noir notes

When using sandbox make sure you have compatible noir version
you can install a different noir version with

> `noirup --version 0.11.1-aztec.0`

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
);

}
```