Findora Research Synthesis

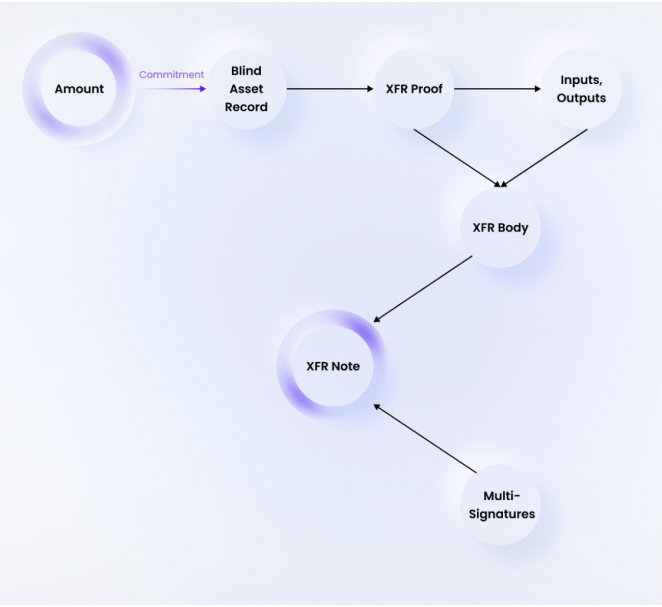
Dylan Kawalec

Layer 1 UTXO Confidential Transfers for decentralized securities.

Bulletproof tx | High Level Overview.

Introduction to Bulletproof | The *Blind Asset Record*

- There's a ***Homomorphic process*** that blinds and hides our tokenized assets. There are **no negative commitments** that ZKP encryption accepts to post to the block: Negative numbers don't work with ZKP. These transactions (tx) are posted as *notes*, called "**XfrNote**". The body of this note "**XfrBody**" contains a list of, [inputs and outputs].
- These records are then **Blinded** using the Cryptographic commitments "**Pedersen commitments**" using the *Ristretto Elliptic curve*.



Technical Procedure

1.1.0 - In the form of data, our Input & Output creates a Blind Asset Record (BAR) called a **XFRproof**,

- In the body of this **BAR ("XFRproof")** contains the { asset | tracing memo | owner memo }

Note | *There's a difference between the **Asset** and the **Blind Asset Record**. Everything committed to the Blind Asset Record runs a **Diffie Hellman Exchange** of numbers that is then placed into the body of the Blind Asset Record (BAR)*

1.2.0 - This is known as the **Pedersen Commitment** — *A commitment scheme is a cryptographic primitive that allows one to commit to a chosen value while keeping it hidden to others, with the ability to reveal the committed value later. Commitment schemes are designed so that a party cannot change the value or statement after they have committed to it : that is,* ***commitment schemes are binding****.*

{...}

Development Question: do I need to know how to commit and store the transaction's contents into the blind asset Record?

Follow up – how do I apply the *Pedersen Commitment* to the transaction recorded in the Blind asset Record?

{...}

1.2.1 - This storage (perseden commitment) is referred to as the **BAR**, "*blind asset record*"

1.2.2 - This applies the "Bullet Proof" Encryption scheme to the blind asset record (BAR), which already has the commitment amount posted.

{...}

Question: **What is the XFR proof**?
{...}

1.2.3 "I commit an amount by blinding it (BAR). when I begin to send the **tx**, we then perform the **XFRProof** - this is committed along with the input and expected output of my transactional commitment → The **xfrBody** is now created! This body, along with the signatures of both parties performing the transfer of the confidential **tx** contract, crest the **xfrNote** !

Procedural step by step review

- **BAR → XFRProof → XRFBody → XRFNote →**

^

←——— Verified Multi-Signature Batch ——-→

V

- Multisig Verifies : XFRBody, this was done to check and see if the asset record was constructed properly.

Three Proofs used in BAR

1.2.4 - The **range proof**, is done first to verify the asset type
1.2.5 - **Chaum Pedersen Equality proof**, is done second to verify the asset type.
1.2.6 - **Asset mixing proof,** is done third to tell whether the asset is confidential or not. This uses the **Schnorr proof of knowledge of discrete logarithms**, to verify the amount.

Verifying the xfr note | citation repository
https://github.com/FindoraNetwork/zei/blob/develop/api/src/xfr/mod.rs#L486

> **Note** | When verifying the XRFbody, it verifies the asset record and tracing proofs.
> When verifying the committed Asset Record: we perform a **bulletproof** (a type of **range proof** and **asset mixing proof**. The Sigma Proof, "**Chaum-Pedersen**" protocol, allows a prover with the keys to prove that an *El-Gamal cipher-text* is valid cipher-text. Chaum-Pedersen is used to denote a sum of multiple terms.

The Bullet Proof | Citation article

https://courses.grainger.illinois.edu/CS498AC3/fa2020/Files/Lecture_15_Scribe.pdf - then completing the bulletproof on the sum of the circuits.

> Note | xfr notes are verified by the Bodies & the Multi-signatures in two separate batches, one after the other. The reason being, it's for compatibility purposes with **WASM** (Web Assembly). Zei library is single-threaded by default. and, due to a natural limitation of **batching**, actually our validators verify each XFR note one after the other.

{...}

Question:

What is the Public Identity Commitment?
Answer - hash(private key / private string): acts like a public key on chain.

Question:
Please explain how XFRNote's are verified without any validator intervention?

{...}

Encryption Methods, Technical Specifications

Diffie-Hellman key exchange

video - Secret Key Exchange (Diffie-Hellman) - Computerphile

Diffie's rules

- You must agree on a *prime modulus* and a *generator*.
- In order to get the shared secret key result which reveals the prime modulus, you must know Alice or Bob's Private key's in order to break the shared secret key. There are two possible keys that can break the prime modulus generator, since both result in the same **Diffie Key**.

Follow the Steps of a Diffie-Hellman Key Exchange

1. Agree on a Prime Modulus and a Generator Number set {x,y}
2. **Alice** the sender selects a ("PrivateKey - A") PK-A, and uses {x,y} x modulus y = (*equal's*) a public result, (**z1**)

- The public result is shared to **Eve**, the middleman; **Bob** is the recipient.

3. Bob selects a ("PrivateKey - B") PK-B, and uses {x,y} x mod y = a public result, (**z2**)

- The public result is shared with Eve the middleman and Alice, the Sender.

4. Alice takes Bob's public Result and raises it to the power of her Private key (PK).

- **z2^PK-A** = *shared_secret*

5. Bob takes Alice'spublic Result and raises it to the power of his PK.

- **z1^PK-B** = *shared_secret*

6. The *shared_secret* is the exact same result.

Diffie Hellman | Citation article
https://www.khanacademy.org/computing/computer-science/cryptography/modern-crypt/v/diffie-hellman-key-exchange-part-2

{...}

Question:

The Blind asset decryption between party A & B, how does the validator verify the PK or verify the diffie hellman exchange between the two parties without knowing the PK during the Transaction?

Answer – *There's an asset manager for all committed asset types & amounts. The commitments are encrypted in the form of a text, and then this text is decrypted by the receiver once the exchange is then committed and the note is then verified.*

{...}

Ristretto Elliptic Curve

link - What is the Ristretto Elliptic Curve?

- Ristretto is a technique for constructing prime order elliptic curve groups with non-malleable encodings. It extends Mike Hamburg's Decaf approach to cofactor elimination to support cofactor- 8 curves such as Curve25519.

Homomorphic encryption

- Homomorphic encryption is the conversion of data into ciphertext that can be analyzed and worked with as if it were still in its original form. Homomorphic encryptions allow complex mathematical operations to be performed on encrypted data without compromising the encryption.

Chaum-Pedersen

video - Chaum pedersen zero knowledge protocol

- "Chaum-Pedersen protocol allows a prover with the keys to prove that an El-Gamal ciphertext is valid ciphertext".

Schnorr's Protocol

link -"Schnorr Digital Signature"

video | What is Schnorr and why is it important?

- This is a great video that explains the Schnorr Signature process. From what we can gather, it's a private way to encrypt multiple signatures into a single digital signature. You don't know if it was 1 or 20 people who signed for a single transactional amount to be sent or received. **It's memory efficient and it enables you to create really interesting privacy features –** *This single protocol can be applied to any cryptocurrency and it would be an automatic upgrade for the asset's intended utility. In the case for Findora, we are applying a discrete logarithmic encryption to the FRA asset amount.*

Bulletproofs

video - Bulletproofs: Short Proofs for Confidential Transactions and More

- A type of **range proof** that the sender proves they have a non-negative amount. This relies on elliptic curves, and requires no trusted setup.

How Bulletproof process works, a high Level Overview.

Begin: Amount → Binary bits → Commit each bit → Prove that the *sum* of the bits (in exact order) is the *sum* of the amount committed.

• referred to as the "*recursive inner product argument*."

Range proofs via inner product arguments

Let $C_v = g^v h^r$ be a commitment to a value v. To show that v lies in the range $[0, 2^n - 1]$, it suffices for the Prover to show that he knows a vector $\mathbf{a}_L = (a_0, \cdots, a_{n-1})$ such that:

- $\langle \mathbf{a}_L \, , \, \mathbf{2}^n \rangle = v$, which shows that $v = \sum_{i=0}^{n} a_i \cdot 2^i$
- $\mathbf{a}_L \circ (\mathbf{1}^n - \mathbf{a}_L) = \mathbf{0}^n$, which shows that the entries of $\mathbf{a}_L$ lie in $\{0, 1\}^n$.

In other words, this shows that the n entries of a_L represent the bit decomposition of v.

For randomly generated challenges $y, z \in \mathbb{F}_p$, it suffices to show that,

$$z^2 \cdot \langle \mathbf{a}_L \, , \, \mathbf{2}^n \rangle + z \cdot \langle \mathbf{a}_L - \mathbf{1}^n - \mathbf{a}_R \, , \, \mathbf{y}^n \rangle + \langle \mathbf{a}_L \, , \, \mathbf{a}_R \circ \mathbf{y}^n \rangle = z^2 \cdot v$$

where $\mathbf{a}_R = \mathbf{1}^n - \mathbf{a}_L$

Pedersen Commitments

video - Pedersen Commitment

- **Pedersen Commitments** are *perfectly hiding* – a computationally powerful attacker cannot find out the data in the commitment, no matter how powerful this attacker is. While, “perfect hiding” implies imperfect binding, meaning that a computationally powerful attacker can de-commit it to other values. This is often enforced by cryptography, saying that such an attacker doesn’t exist in the real world. This is used during the BAR process.
- *“A Pedersen commitment is a point C on an elliptic curve that is cryptographically binding to a data message m, but completely hides the message. A Pedersen commitment hides the message in an even stronger way than encryption. The curve point is completely random and contains no information at all about m.” –* ***google search***

Proof Generation

The way findora denominates the XRFProof, it must be equivalent to the sum of the blinded outputs from the BAR (Blind asset record). The XRFProof then equates the sum of the outputs against the sum of the inputs to verify that the committed asset amount has not been tampered

with before transferring the full amount. This is within respect to the perseden commitment algorithm at the start of the transaction's initial commitment to become a Blind asset record.

What about fee's? Don't they mess up the commitment's total sum during the transaction?
- The answer is, NO – "*the fees are denominated in the FRA token", the fee is* ***Fixed.***

$$\sum_{i \in Input[FRA]} \alpha_i = \sum_{j \in Output[FRA]} \beta_j + Fees$$

← *fee's are added after ;)*

The text that's sent as XRFNote is sent to the receiver's public key, and decrypted with the receiver's Private key. **None of the key exchange / decryption happens outside of the Sender or Receiver's hand**. The note's cannot be decrypted by Validators.
"The security of this scheme hinges on the hardness of the Discrete logarithm problem (DLP). The proof of the amount-sum equality relies on the homomorphic property of Pedersen commitments." – wiki.findora.org
…

Proving Commitment Equality

– A Perseden commitment comes with the Asset Amount & Type – this is the beginning of your **Bulletproof**. We must prove for equality in order for the preseden commitment to complete the Bulletproof.
– Using an aggregation trick, this proof can be kept constant-sized when the Prover needs to show that the asset type committed in the Asset record is the same size as the Blind Asset Record. Using bits and fancy math …

$C_1 = g_1^{a_1} g_2^{b_1} h^{r_1}, C_2 = g_1^{a_2} g_2^{b_2} h^{r_2}$ are such that $a_1 = a_2$,

We are able to prove that the size of the constants-size are, "consistent".

Proving the Amount-Sum Equality

– when we verify that the commitment is equal to itself (the transparent record is the same size as the blind asset record's amount & type) we move on to check if the receiver received the same encrypted sum. This part checks the outputs against the original inputs.

– the prover, who is the sender, does a zero knowledge trick that the prover knows some integer (r) and uses the bit size of type (a) committed in the input/output verification method...

More math...

$\widetilde{C} := (\prod_{i=1}^{m} C_i^{\text{in}}) \cdot (\prod_{j=1}^{n} C_j^{\text{out}})^{-1}$ is of the form $g_1^{a(m-n)} h^r$ (or $g_1^{a(m-n)} g_2^{\text{fees}} h^r$

– In order to prevent double spends, the sender must perform the initial Zero Knowledge Proof that the original amounts committed are all non-negative. Luckily, the Commitment scheme prevents all non-negative input values from being committed, using the range proof. These proofs are small in size and take up less memory to store on the chain (<=1KB), with a clock time of .34 milliseconds per proof; Fast and effective.

Proof Verification

- The verification process actually happens in batches for increasing the efficiency

Batch verification Overview - the empire batch strikes back #ZK

Batch Part 1 "Summarize the constructed size" : We verify the multisignature XRFNote → Which verifies the XRFBody of the note in a batch, checking to see if the multisig is the sum of the committed inputs. How? We take the XRFBody and serialize it to get a ZKP size in bytes and check for any non-negative errors → we verify the body of the note with the receiver's public key and check for a *consistent* constructed size.

Batch Part 2 "Check the contents and path" : Now we check to see if the XRFNote's body is constructed properly in the batch → step 1. verify the

amount and asset types using the bulletproof method → step 2. verify the trace of the proofs.

- Note : Simultaneously, we checked the XRFbody within two additional steps in "batch part 2", as described above.

The Final Batch

- "the three proofs" : these proofs are done in a batched manner, one after the other.

First : "Confidential Amounts"→ A batch range proof which is using the zei library performs this execution: this is known as your BULLETPROOF , and it checks for errors in the amount being processed.

Second : "Confidential Types…clam-Chaum-der" → ignoring my cheesy naming conventions as I continue to repeat the same things over and over, this step assess the asset type with the *Chaum Pedersen Equality proof*.

Third : " the snore proof.." → verification! This consists of both confidential asset and non-confidential assets; "*Schnorr Signature process*"

Why is ZK so Complex!?

The proofs are batched so that the communication complexity and the verification time stay constant. The verifiers are performing crazy discrete logarithms, hashing algorithms and inner product checks to make sure everything is sound. If you care to see what I mean by complex, look no further → Link - we had to simplify the batching technique in order to make sense of the code, which can only be explained using classy star war's references and soup recipes.

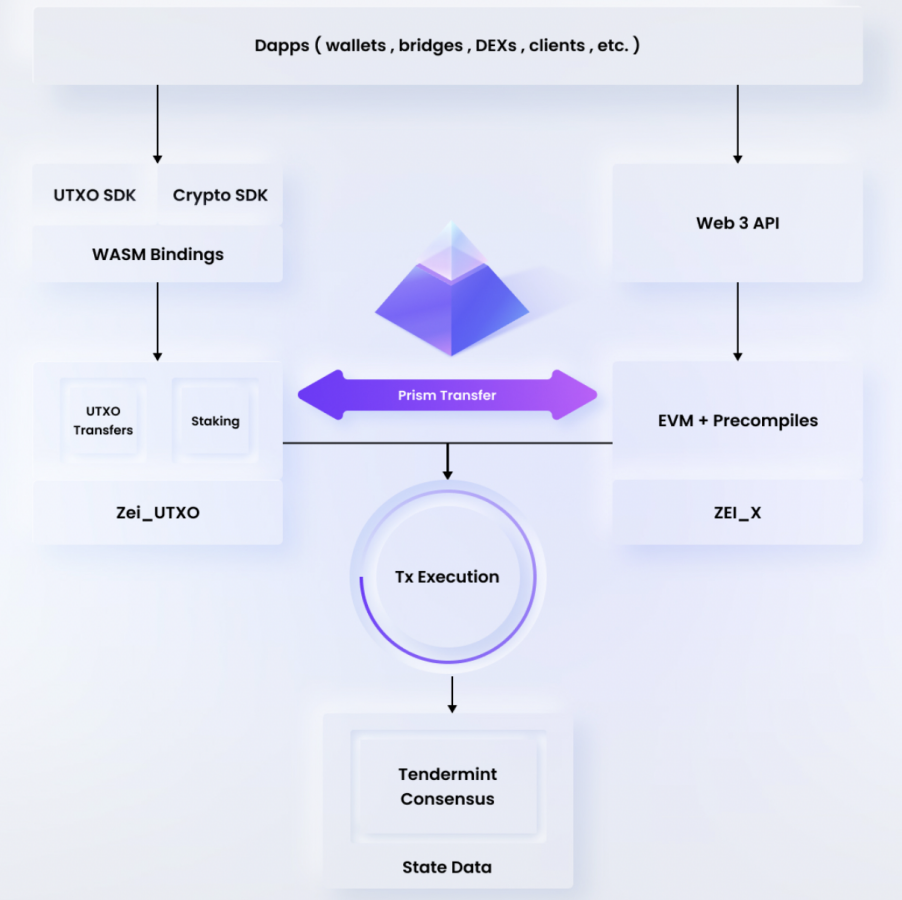
{...}

Question:

Does the Validator behave as party C and hold onto the Public results of the Prime Modulus Calculation?

Answer – the validator is NOT involved at all in the Key exchange process.

{...}



---

www.Findora.org/#

[BAR <> ABAR]

Findora Triple Masking

Confidential Decentralized Financial Services

Introduction to Private Transactions

Findora supports Multi-Asset transfers and gives their users the ability to audit their transaction's while still being able to perform fully anonymous / confidential — "pseudonymous transfers". This gives developers a level of custom programmability when it comes to the level of privacy they intend to give users; the three categories privacy is applied to are – [Asset type, Asset amount and the Sender/Receiver's 0X/FRA address]. Developers have full control over these interchangeable privacy features while building their Decentralized Privacy Fincianal Applications (PriFi) —

The three types of Ethereum transactions

- Asset Transfer - fully transparent transaction → "*public knowledge transfer*".
- Confidential Asset Transfer - Asset type and amount are hidden → "Bulletproof".
- Anonymous Asset Transfer - Identity of the Sender & Receiver + Amount & Type of the asset is hidden.

— Anonymous transfers are made possible by the use of **zk-SNARKs** - "*Zero-Knowledge Succinct Non-Interactive Argument of Knowledge*". The user provides a cryptographic proof that validates the authenticity and validity of the transfer to assure that the transfer was conducted by the sender of the anonymous transfer. The reference the Findora foundation uses for this kind of transfer is called **Triple Masking.**

If you have been following along this far, you would have read that Findora's **bulletproof** schema requires no trusted set up or provider to initiate a full confidential transfer.

> "Bulletproofs rely solely on the discrete logarithm problem in elliptic curves; which is one of the oldest and most battle-tested hardness assumptions in cryptography."
>
> ...
>
> "While the same scheme can, in theory, also support an anonymous transfer, the verification time is linear in the size of the circuit, which is far too expensive for the complex statements that constitute an anonymous transfer. To this end, we use ***TurboPlonk***, a pairing-based Snark which has a constant-sized proof and a constant verification time. While this scheme **does** require a trusted setup, it is far better suited for more complex statements. Furthermore, the trusted setup is universal and updateable."
>
> — Findora.org

{...}

Question:

What sort of trusted set up is required to perform a triple masking transfer? Is a validator involved at all in this process?

{...}

BAR to ABAR

- BAR – Blind asset record.
    - Hidden : Asset Type and Asset Amount
    - Public : Public Key
- ABAR – Anonymous blind asset record.
    - Hidden : Asset Type + Asset Amount, and Public Key

Note : In order to perform the **ABAR**, we need a new anon data structure and a trusted setup. The public key lies on a dense **Merkle Tree**, which acts as the *version verifiable accumulator*.

Converting tokens from a BAR to an ABAR

CHECK : 1. Identify that the BAR has been unspent using the **TxoSID**; this also validates the true possession & ownership of public keys → 2. Identity the holder of the **Anonymous keys.**

- *Once checked, only then is the ABAR ownership assigned*

How does the check work? : The unspent status and possession of the public key of a BAR is inside of a **set**, which replays the transactions within this particular set.

- "Imagine rewinding a film back to the beginning of the film – this is essentially checking to see if the film is starting at the very beginning as a fresh new set or a watched film, which would be a 'spent set' ."

NOTE : The equality check to compare the BAR & ABAR's asset Amount & Type is done separately from this 'spending status check'.

Review

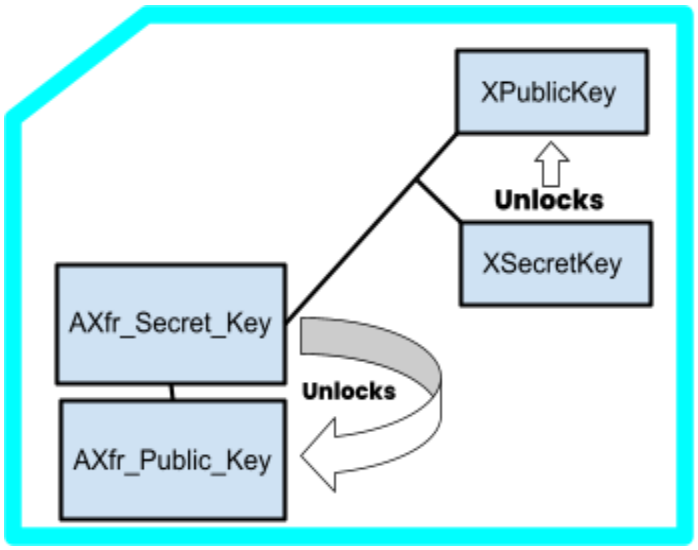
- A BAR is spent to make an ABAR → We perform a spending check, by giving the *integer ID* (**TxoSID**) a Single Signature, which corresponds to the public key on the ledger.

- Before ownership is assigned, the ABAR is associated with what's known as the *Spending Key* (aka : **anonymous key** ; also referred to as your "private key"), which is used to spend ABAR's previously sent to its corresponding public key owner. An Anon key is required for a **triple masking** transfer ("anonymous transaction").

BAR <> ABAR Overview

- The **anon key data structure** is like a *black box* that only the spender of the ABAR has access to. It contains the **spending**

key, which is a private key correlated to the owner's public key. It's also known as the AXfr_Secret_Key, which grants access to the AXfr_Public_Key (this is the public key associated with the **anon key data structure**). The AXfr_Secret_Key is in of itself, a keypair! The key pair within the AXfr_Secret_Key unlocks another Secret key which unlocks another public key!! The **Spending key pair** contains the **Decryption X Key Pair** : 1 / The **Decryption Key**, named the XSecretKey, which unlocks 2/ the **Encryption Key**, the XPublicKey. The XPublicKey is used to encrypt the **HMAC**, known as the *owner's memo* ; and the XSecretKey is used to Decrypt HMAC.



Overview of the previous overview…

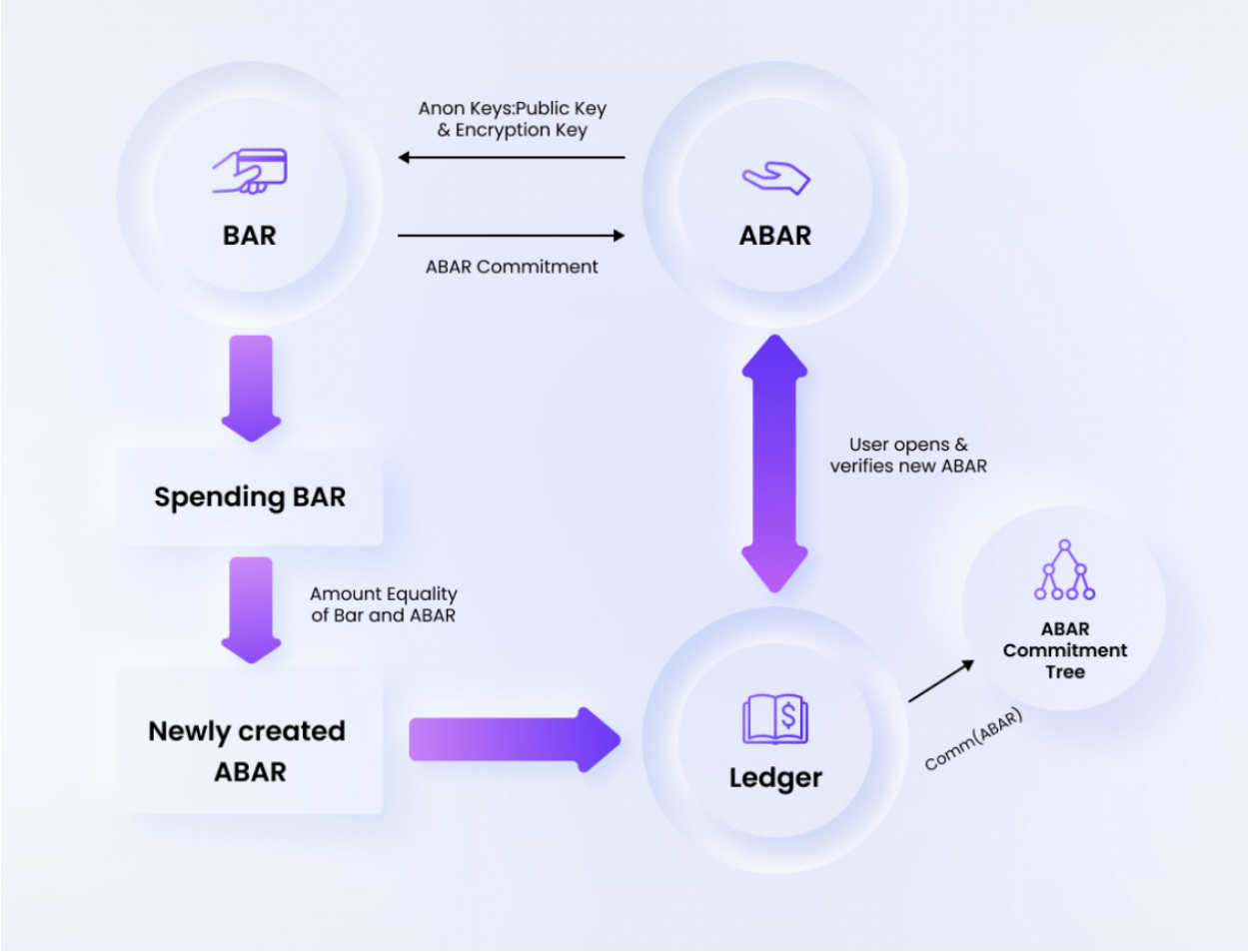
1. **The Spending key** is received through the spending of the BAR.
   - **ABAR** lets you use the **Spending Key =** **AXfr Secret key**, which is correlated to the **Decryption Key** and **AXfr Public Key**.
2. **"Decryption key" =** **XSecretKey** which lets you Decrypt the Owner memo ("**HMAC**"), or Encrypt the memo with the **"**pub enc_key", ie: the **XPublicKey**.

Note : Our **Anon Key Data Structure** has the red key "Spending key" and Pink key "Decryption key".

– The **Red key** gives you access to the **Orange key**, and the **Pink key** gives you access to encrypt with the **green key**.

ANON key Data Structure (use color guide to track which key is which in the overview)

```
pub struct AnonKeys {
   pub axfr_secret_key: AXfrKeyPair,
   pub axfr_public_key: AXfrPubKey, // can be obtained from AXfrKeyPair
   pub enc_key: XPublicKey, // can be obtained from XSecretKey
   pub dec_key: XSecretKey, // 4 keys in total.
}
```



Similar to the **TxoSID** for BAR, when you generate a BAR to ABAR, you create a **ATxoSID.** This lets you spend the ABAR in the future. You may have assets inside the Anonymous wallet after the BAR<>ABAR transition; but you can't see what's inside. That's exactly why we explained the process above –
To Open the ABAR, you commit the **AXfr Secret key** and **Decryption X key**

What is inside the ABAR?

- **Asset Amount** - unsigned integer ‘64
- **Asset Type**
- **Public Commitment** – “BLSScalar”
- **AXfr Public Key -** “secretive public key”
- **Owner Memo -** ABAR opening & closing mechanism : XSecretKey (decryption) / XPublicKey (encryption, obtained by the XSecretKey)
- **Merkle root value**

{...}

Question :

Can we go backwards from BAR to ABAR , and then Back to a BAR again (?)

Are all ABAR’s forever ABAR’s after the BAR<>ABAR is conducted?

{...}

ABAR to BAR

For each ABAR that is validated, there is an associated ABAR commitment with a fixed position in a **Merkle tree** of ABAR commitments. Findora uses a *3-ary Merkle tree* built with the ”Snark-friendly” ***Rescue* hashing algorithm** for the future spending of ABAR’s. All ABAR’s are associated with their own unique **nullifier hash** within a **nullifier set**, this prevents ABAR’s to forge or double spend other ABAR’s out there. To prove the Nullifier, you must have the **spending key**, which means you must also have the **private key** associated with it. The way in which nullifier sets are stored is done using a **binary sparse merkle tree** built using the **SHA-256 hashing algorithm**. These Nullifiers are the only thing shown to prevent them from being spent again. In order to know their position on the **3-ary Merkle tree**, you must know the proof that proves against a dense discrete logarithm problem, which is not possible for any known hacker to do. The way the commitment is stored in the form of the

nullifier set is done using a method called **Turbo plonk**, which is instantiated using "***Kate/KZG polynomial commitment scheme*** *as the zk-Snark that enables this mechanism*". Turbo plonk is a customized, gated optimization of the existing Plonk method. In our instance, the turbo plonk is the Zero Knowledge proof that provides the proof of ownership of the ABAR. Once an ABAR is spent or transferred, a new one is addressed to the new owner.

High Level Overview of the ABAR <> BAR Process

Check For Key Ownership

- Spending the **ATxoSID** of the ABAR which exposes the unique commitment of the BAR
- Use Anon Keys : Spending & Decryption X keys.
- We Spend the ABAR to prove the existence of the ABAR.

Merkle Proof

- Prove the correctness of the correctness in the Merkle Tree proof, originally used to store the public key. Using ZKP, we can verify the Secret Spending Key is owned by the owner in question.

Check Merkle Root

- Using the owner's XSecretKey, The Root of the merkle proof is checked by the network
    - Still, The spending of the ABAR remains a secret.

Hash it

- Spender provides a new HMAC (owner memo) and creates a unique **nullified hash (Tree #1)** which is derived from three fields (asset Type, Amount and Owner's Public Key).
    - Fact Check : this comes from the **XPublicKey** that decrypted the original *Owner Memo* (HMAC) from the BAR<>ABAR process.

Plant it

- The **Nullified hash** is then committed to a nullifier set, also referred to as the **nullifier tree (Tree #2).**

- This tracks the spending of all ABAR's to avoid double spends and forked transactions.

Prove it

- Prove the correctness of the HASH using ZK to avoid forging of the ABAR.

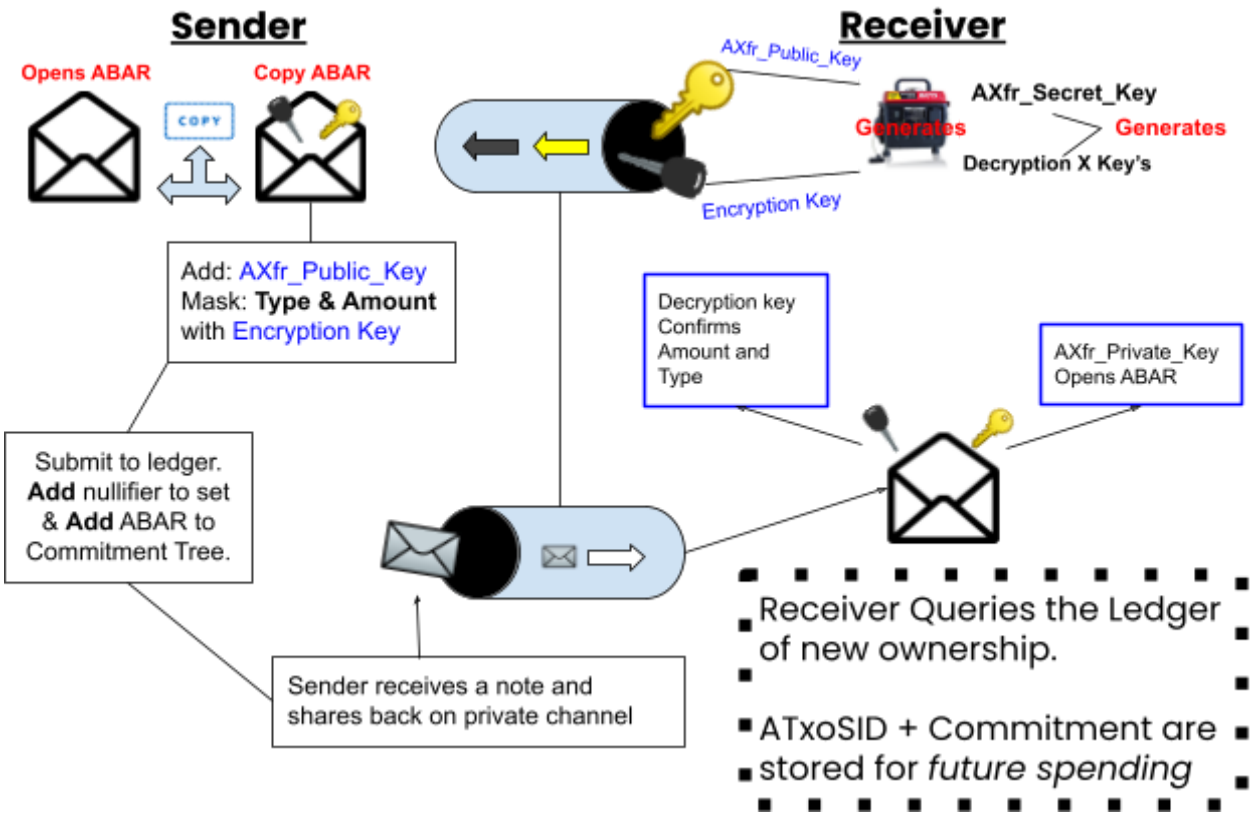
Talk about it

- Returns the new BAR with an equal amount as the ABAR!

Reminder : the public key of the owner of ABAR<>BAR transaction will no longer be anonymous. All nullifiers cannot be replicated to forge or double spend any other ABAR's presently on the commitment tree, and each nullifier is unique.

ABAR Transfer Sequence of steps - Graph

link - **Sequence of steps in ABAR Transfer**



Merkle Tree

video - Merkle Tree | Merkle Root | Blockchain
link - Merkle proof's in Ethereum

Nullifier Tree

- A Nullifier is **a hash of a secret nullifier_seed and the UTXO leaf index in the UTXO tree**. The user sends a transaction and attaches a Nullifier with it. The coordinator runs the circuit off chain and tries to update each Nullifier leaf. A successful update should change the tree root.

XfrProofs structure : it must do a double check summary of the inputs and outputs of the Zero Knowledge Proof it's conducting. The second instance denominates the fee that is then applied before the BAR has been verified.

"Fee's are dynamic, depending on the network".

Once the asset is determined to be an FRA asset, then the fee's are added to the Xfr proof.

Question: Are all UTXO's FRA's?

Discrete logarithm problem ? ; since it determines the randomness given by the Pedersen commitment once the BAR is

instantiated in the Blind asset Body

Anonymous Transfers

zk-SNARKs ; Zero-Knowledge Succinct Non-Interactive Argument of Knowledge

TxoSID, which an integer ID of the BAR and a signature corresponding to the public key on the ledger.

- proving the amount of the BAR is the same as the ABAR is a different process.

- Anon Key data structure uses a “spending key” that is linked to the public key for the transactions to be sent using the BAR<> method.

- AXfr Secret Key allows you to spend the ABAR

- decryption key can encrypt or decrypt the HMAC of the owner,

---

{...}

Question : **How hardened is the Turbo plonk** ?
{...}

Conceptually __ How to convert a BAR to an ABAR?

- we use the “owner memo”

- “**ATxoSID**” : BAR to ABAR.

- ABAR consists of : asset amount, type, axfr public key, HMAC and merkle root value.

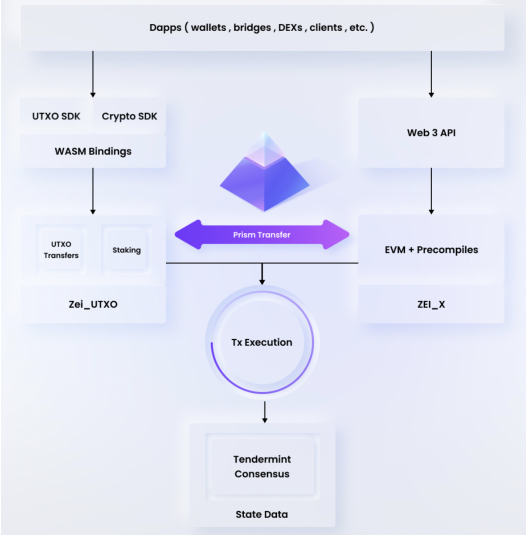
- *Anonymous transactions require a trusted setup that has a constraint verification time. This is the Anon Wallet and transactionBuilder to make a trusted channel to do a full BAR <>ABAR <> BAR Transaction*.

Question: Can you explain the passing of the keys during the BAR<> ABAR routing so we understand where the keys are during the tx-encryption ?

Overview, review the terminology

Multiparty Computation : Multi-party computation (MPC) is a cryptographic tool that allows multiple parties to make calculations using their combined data, without revealing their individual input. Invented by Chinese computer scientist Andrew Yao, MPC works by using complex encryption to distribute computation between multiple parties.

Findora Primitives : In computer programming, a primitive (pronounced PRIH-muh-teev) is **a basic interface or segment of code that can be used to build more sophisticated program elements or interfaces**.
a security identifier (SID) is **a unique value that is used to identify any security entity that the Windows operating system (OS) can authenticate**.



SDK - > Wasm Bindings -> Zei = Full BAR<>ABAR/ABAR<>BAR

Yellow Submarine Implementation

Link to repository Information

---

sidebar_position: 1

Crypto SDK Guide

This guide will show you how `Yellow Submarine` allows users to convert an asset to a private asset and transfer it to a brand new wallet address by `Findora Triple Masking`.

It conducts two ZKP operations:
- transfer asset to `anonymous wallet` from `wallet A`. **(bar to abar)**
- transfer asset to `wallet B` from `anonymous wallet`. **(abar to bar)**

The combination of these two operations is the key to removing the trace.

Installing the `Findora SDK`

To install the `Findora SDK` we only need to run one single command:

```bash
yarn add @findora-network/findora-sdk.js
```

1. Setup the Findora SDK

```ts
  // Top-level await
  const findoraSdk = await import('@findora-network/findora-sdk.js');
  const findoraWasm = await findoraSdk.getWebLedger();

  const { Network: NetworkApi, Transaction: TransactionApi } = findoraSdk.Api;
```

2. Create two Findora Wallet
```ts
  // create a Findora Wallet, this wallet will be the source wallet
  const keypairA = findoraWasm.new_keypair();
  const walletA = {
    keypair: keypairA,
    keyStore: findoraWasm.keypair_to_str(keypairA),
    privateKey: findoraWasm.get_priv_key_str(keypairA).replace(/\"/g, ''),
    publickey: findoraWasm.get_pub_key_str(keypairA).replace(/\"/g, ''),
    address: findoraWasm.public_key_to_bech32(findoraWasm.get_pk_from_keypair(keypairA)),
  };
```

```ts
  // create another Findora Wallet, this wallet will be the destination wallet
  const keypairB = findoraWasm.new_keypair();
  const walletB = {
    keypair: keypairB,
    keyStore: findoraWasm.keypair_to_str(keypairB),
    privateKey: findoraWasm.get_priv_key_str(keypairB).replace(/\"/g, ''),
    publickey: findoraWasm.get_pub_key_str(keypairB).replace(/\"/g, ''),
    address: findoraWasm.public_key_to_bech32(findoraWasm.get_pk_from_keypair(keypairB)),
  };
```

3. Create a Findora Anonymous Wallet

```ts
  // create a Findora Anonymous Wallet
  const anonKeys = findoraWasm.gen_anon_keys();
  const anonWallet = {
    axfrPublicKey: anonKeys.axfr_public_key,
    axfrSecretKey: anonKeys.axfr_secret_key,
    decKey: anonKeys.dec_key,
    encKey: anonKeys.enc_key,
  };

  // release the anonymous keys instance
  anonKeys.free();
```

4. Bar to Abar

```ts
  // create an instance of the transaction builder
  const transactionBuilder = await TransactionApi.getTransactionBuilder();

  // get the informance of the UTXO by specific sid
  const { response: [sid] } = await NetworkApi.getOwnedSids(walletA.address);
  const { response: utxo } = await NetworkApi.getUtxo(sid);
  const { response: ownerMemoData } = await NetworkApi.getOwnerMemo(sid);

  const ownerMemo = ownerMemoData ?
findoraWasm.OwnerMemo.from_json(ownerMemoData) : null;
  const assetRecord = findoraWasm.ClientAssetRecord.from_json(utxo);

  // the destination anonymous wallet
```

```
  const axfrPublicKey = findoraWasm.axfr_pubkey_from_string(anonWallet.axfrPublicKey);
  const encKey = findoraWasm.x_pubkey_from_string(anonWallet.encKey);

  const keypair = findoraWasm.keypair_from_str(walletA.keyStore as string);

  // add_operation_bar_to_abar will return a instance of the transactionBuilder, which would be
used to submit the generated tx to the network
  transactionBuilder = transactionBuilder.add_operation_bar_to_abar(
    keypair,
    axfrPublicKey,
    BigInt(sid),
    assetRecord,
    ownerMemo?.clone(),
    encKey,
  );

  // The only way to get access to the funds from the `abar` is to ensure that commitment is
saved
  // after the operation is completed and transaction is broadcasted.
  // `commitments` MUST be saved in order to get access to the funds later.
  const commitments = transactionBuilder?.get_commitments();

  // Finally, broadcast this transaction to the network
  await TransactionApi.submitTransaction(transactionBuilder.transaction());
```

5. Abar to Bar
```ts
  // create an instance of the transaction builder
  const transactionBuilder = await TransactionApi.getTransactionBuilder();

  // the destination wallet
  const receiverXfrPublicKey = findoraWasm.public_key_from_base64(walletB.publickey);

  // the source anonymous wallet
  const aXfrKeyPairSender = findoraWasm.axfr_keypair_from_string(anonWallet.axfrSecretKey);
  const secretDecKeySender = findoraWasm.x_secretkey_from_string(anonWallet.decKey);

  // we need to provide a commitment string which we would `transfer` to the destination wallet
  const commitment = 'YOUR_COMMITMENT';

  // `Abar to Bar` operation would require instances of abar, which will be created (restored)
  // using the commitment strings
  const { response: ownedAbarsResponse } = await NetworkApi.getOwnedAbars(commitment);
```

```
  const [atxoSid, _ownedAbar] = ownedAbarsResponse;
  // instances of abar
  const ownedAbar = findoraWasm.abar_from_json(_ownedAbar);

  // get the informance of the abar
  const { response: abarOwnerMemoData } = await NetworkApi.getAbarOwnerMemo(atxoSid);
  const { response: mtLeafInfoData } = await NetworkApi.getMTLeafInfo(atxoSid);
  const abarOwnerMemo = findoraWasm.OwnerMemo.from_json(abarOwnerMemoData);
  const mTLeafInfo = findoraWasm.MTLeafInfo.from_json(mtLeafInfoData);

  // add_operation_abar_to_bar will return a instance of the transactionBuilder, which would be
used to submit the generated tx to the network
  transactionBuilder = transactionBuilder.add_operation_abar_to_bar(
    ownedAbar,
    abarOwnerMemo,
    mTLeafInfo,
    aXfrKeyPairSender,
    secretDecKeySender,
    receiverXfrPublicKey,
    false,
    false,
  );

  // Finally, broadcast this transaction to the network
  await TransactionApi.submitTransaction(transactionBuilder.transaction());
```