Private token prototype (with Aztec/Noir)

https://github.com/taurusgroup/private-tokens

TODO @Gustave Charles-Saigne

Introduction:

Institutions are finally understanding the power of blockchain and the disruptive potential it has for their financial applications, storage and payments, among others. While they are now focusing on riding the wave, and going "onchain", their next logical question will be: "how can I guarantee the privacy that I had before on this new public and open software?". This is what certain protocols and teams have thrived to achieve for the past few years: a public and open blockchain that maintains the privacy of its users. Thanks to recent breakthroughs in cryptography, blockchains like Aztec are able to bring enshrined privacy into blockchain. The aim of this project is to leverage the Aztec blockchain to create a private version of the CMTAT, pioneering and getting ahead of the trend.

Aztec and Noir concepts

Users addresses and keys

Keys can be pinned to a certain block so that key rotation doesn't impact the validity of an authentication at any given time.

4 keys for each account in Aztec:

- **nullifier key pair:** used for note nullifier computation, comprising the master nullifier secret key (`nsk_m`) and master nullifier public key (`Npk_m`). Rotating nullifier keys requires the nullifier public key, or at least an identifier of it, to be stored as part of the note.
- *nskm* MUST NOT enter an app circuit.
  nskm MAY enter the kernel circuit.
- Npk_m = derive_public_key(nskm)
- **incoming viewing key pair:** used to encrypt a note for the recipient, consisting of the master incoming viewing secret key (`ivsk_m`) and master incoming viewing public key (`Ivpk_m`).
- *ivskm* MUST NOT enter an app circuit.
- Ivpk_m = derive_public_key(ivskm)
- **outgoing viewing key pair:** used to encrypt a note for the sender, includes the master outgoing viewing secret key (`ovsk_m`) and master outgoing viewing public key (`Ovpk_m`).
- *ovskm* MUST NOT enter an app circuit.
  ovskm MAY enter the kernel circuit.
- Ovpk_m = derive_public_key(ovskm)
- **tagging key pair** used to compute tags in a tagging note discovery scheme, comprising the master tagging secret key (`tsk_m`) and master tagging public key (`Tpk_m`).
- *tskm* MUST NOT enter an app circuit.
- Tpk_m = derive_public_key(tskm)
- **signing key pair:** as there is account abstraction, this will/must be implemented by the wallet provider.

Authentication witness: Scheme for authentication actions on Aztec, so users can allow third-parties (eg protocols or other users) to execute an action on their behalf. It is defined for a specific action. For example: allowing an app to transfer funds on your behalf.

In private context, the authwit is created by the user who is making the action. "I want Defi_protocol to send X tokens on my behalf in private" → I will call Defi_protocol transfer function, which will call the transfer function of token, which will check with a private execution oracle call if I have authwited the Defi_protocol to send my tokens on its behalf.

In public context, account contracts will/should store in the account storage the 3rd party authorisations made by a user. When a user makes a call through a 3rd party, it will send in a batch the signature (authwit) to the account contract.

Account contracts typically implement an `entrypoint` function that receives the actions to be carried out and an authentication payload

Transaction flow and execution environment

1. All private functions are executed in an execution trace
2. A proof of correct execution is generated
3. All public functions are executed

Private state and functions, shields, notes

Private state storage management:

- **pxe database storage - physically stored in client :** store encrypted data (= notes). UTXO = enc(data, owner, owner.sk). An entry is available if there is no nullifier linked to this entry in the Nullifier Set. Also stores authentication witnesses, deferred notes and capsules.
- **One append-only hash notes tree - physically stored in global state:** store hashes of commitments (= hashes of the encrypted data) in an append only tree. **(sometimes referred to as data tree in Aztec doc who knows why?)**
- **One append-only nullifier tree (nullifier set) - physically stored in global state:** To delete notes that have been spent, a matching nullifier is created in the nullifier tree. To create a nullifier for the specific entry, one has to have a nullifier secret key that corresponds to the owner of this specific entry. No nullifier key – no nullifier! Nullifiers are deterministically generated from UTXO inputs and can't be forged. Nullifier = enc(UTXO, owner.sk(= nskm?))
- **Deleting a private value** = emitting the corresponding nullifier
- **Modifying a private value** = emitting a nullifier for current state variable, generating a new private state and append it to the tree
- **reading private values:** reading data from data tree and proving that this data is active is done by reading and creating a new nullifier, so that no one knows if data has been read, or new data has been written.

Private function execution:

Note: Since a user PxE doesn't have an up-to-date view of the latest public state, private functions are always executed on some historical snapshot of the network's state. Private functions do not execute on the latest up to date public state.

Private functions rely on historical state due to concurrency issues. Indeed, execution of private functions is done on the user's device, far away from the sequencer and the network. Picture Alice and Bob, and let's say that private functions rely on the latest public state (PS0). Bob executes a private function and ends up with a new PS1. In the meantime, Alice wrote publicly to PS0, and PS0 became PS1. Bob's PS1 cannot be proven as the new state, because his PS1 is not based on the latests state anymore, as Alice has overwritten it. Bob's transactions is thus aborted. Private function reading from latest state creates cascading aborts, this is why private function read from historical state. Values read from historical state are checked as not nullified.

Proof of execution and correctness is generated client-side before reaching the mempool:

1. Private functions are compiled down to ACIR byte code
2. The PxE simulates and executes on the ACVM the private function client side (creating a **private function circuit**), and generates needed data, particularly, the witnesses needed for proving. The proving is done, after execution, by the backend prover, called barretenberg in Aztec's L2.
3. The private kernel circuit aggregates and verifies functions from the private call stack one by one until there is none anymore. We build a proof of transaction execution correctness (= verify).
4. Private function execution proofs, nullifiers, commitments and logs are sent to the sequencer (the p2p pool from which transactions are picked by the sequencer) as a **transactions object.**

5. Sequencer executes, proves and verifies public functions with the help of a prover and the public kernel circuit, and constructs a block that it passes into the rollup circuit, which creates a final proof. Sequencer updates state trees, UTXO and notes/nullifiers.
6. The proof is verified by smart contract on L1

Private functions can:

- privately read from, and insert into the private UTXO tree
- insert into the Nullifier Set
- create proofs from historical data (coprocessor functionality)
- shield data (move data from public state to private state)
- call public functions (but without any return values)

Private execution environment:

- **PxE** (Client): Library for private execution of functions. The client runs the **ACIR**, the **KeyStore**, and the **LMDB (key-value store) database**. The PXE generates proofs of private function execution using the private kernel circuit. Private inputs never leave the client-side PXE.
- **PxE Service** (Server): API for interacting with the network from PxE
- **ACIR**: simulates Aztec smart contract function execution and generates the partial **witness** and the public inputs of the function, as well as collecting all the data (such as created notes or nullifiers, or state changes).
- **LMDB database:** stores the data in a key-vale store.
- The **keystore** is a secure storage for private and public keys.
- **Private kernel circuit:** on user's device

Note: private variables that hold data. = UTXOs.

Note hash/commitment: public commitment to some note whose value is hidden by the commitment hash property. The notes or UTXOs in Aztec need to be compressed before they are added to the trees. To do so, we need to hash all the data inside a note using a collision-resistant hash function. Currently, Pedersen hash is used.

Note transmission: A note which is created and nullified during the very same transaction is called transient. Such a note is chopped by the private kernel circuit and is never stored in any persistent data tree. encrypted logs : communication channel to transmit notes.

Nullifier: The nullifier is generated such that, without knowing the decryption key of the owner, an observer cannot link a state record with a nullifier. There is a pattern to disassociate notes and nullifiers, we should always use it.

Shield data: move data from public state to private state (= public balance to private balance). Not necessarily the same address.

Unshield data: move data from private state to public state (= private balance to public balance). Public functions can do that if the call was initiated by private function earlier. Not necessarily the same address.

Public state and functions

Public Data Storage Tree: The key-value store for public contract state is an updatable merkle tree. (Public data also consists of the note hash tree and the nullifier tree)

Archive tree allows us to prove statements about the state at any given block

Public function execution:

- Public function code is compiled down to AVM byte code. (once)
- Two ways of executing and proving:
  - The sequencer picks up the transaction in the mempool and executes and proves public functions on the AVM. The sequencer must run the AVM
  - the proof is done by a third party prover
- Proof is verified by the public kernel circuit.
- Proof is then rolled up in the rollup circuit with other proofs and sent to L1 for verification

Public functions

- Can read and write public state
- Can insert into the UTXO tree for use in private functions
- Can broadcast information to everyone (similar to msg.data on Ethereum)
- Can unshield data (move data from private state to public state), if the call was initiated by private function earlier

Unconstrained functions

Generally we want to use unconstrained functions whenever there's something that's easy to verify but hard to compute within the circuit.

They are not constrained by the proving circuit, so if you pass an input that should be private in an unconstrained function, then you won't know and won't be sure that the input has not been leaked. These functions are not part of the proof, however you can verify the computation inside a constrained function.

For example, if you wanted to calculate a square root of a number it'll be a much better idea to calculate this in unconstrained and then assert that if you square the result you get back your number.

```
fn main (in: Field) {
  out = un_sqrt(in)
  reconstructed_num = out^2
  assert(in == reconstructed_num);
  out
}

unconstrained fn un_sqrt(in: Field) → Field{
  out = sqrt(in)
  out
}
```

Unconstrained functions are compiled down to Brilling byte code and executed on the user device. The byte code is executed by the PxE on the ACVM.

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.

Fees market and paymaster and legal

Fee payment is public. Thus, if someone wants to pay fees privately, they will go through a paymaster.

There will be a gas/fee token that is locked on L2 and will not be transferable. This is due for legal purposes

Compliance should happen at an application layer, and not at the protocol layer. Aztec will provide credibly neutral infrastructure, and it's up to the people to build good stuff.

Shared Mutable and Proxy

This variable type is used when you want to create a public variable that can be privately modified and read. Private function read/write is different from public read/write because, as said in the private function concepts part, public functions rely on the latest public state, whereas private function do not.

mutable public state that can be accessed with no contention is hard

In order to support contract upgrades, we need a way to store what the current implementation is for a given contract such that it can be accessed from a private execution and doesn't introduce contention between multiple txs.

If the public state is only changed infrequently, and it is acceptable to have delays when doing so, then shared state is a good solution to this problem.

Aztec Limitations

Note encryption and decryption burden:

One of the functions of the PXE is constantly loading encrypted logs from the `AztecNode` and decrypting them. When new encrypted logs are obtained, the PXE will try to decrypt them using the private encryption key of all the accounts registered inside PXE. If the decryption is successful, the PXE will store the decrypted note inside a database. If the decryption fails, the specific log will be discarded.

For the PXE to successfully process the decrypted note we need to compute the note's 'note hash' and 'nullifier'. GitHub - AztecProtocol/aztec-nr enables smart contract developers to design custom notes, meaning developers can also customize how a note's note hash and nullifier should be computed. Because of this customizability, and because there will be a potentially-unlimited number of smart contracts deployed to Aztec, an PXE needs to be 'taught' how to compute the custom note hashes and nullifiers for a particular contract. This is done by a function called `compute_note_hash_and_optionally_a_nullifier`, which is automatically injected into every contract when compiled.

Crosschain communication

- L1 has bridge contract to lock/unlock funds from L1 to L2.
- L1 has a inbox contract that manages L1-->L2 pending messages
- L1 has an outbox contract that manages L2-->L1 ready messages
- The developers create portal contracts on L1 and L2 that interact with the inbox and outbox contracts.
- L2 has a structure that holds L1-->L2 and L2-->L1 messages.

Message passing:

Since any data that is moving from one chain to the other at some point will live on L1, it will be public. While this is fine for L1 consumption (which is always public), we want to ensure that the L2 consumption can be private. To support this, we use a nullifier scheme similar to what we are doing for the other notes. As part of the nullifier computation we use a `secret` which hashes to a `secretHash`, which ensures that only actors with knowledge of the `secret` will be able to see when it is spent on L2.

Message from L1 to L2:

- message logic is written in portal contract function. Message should be an Aztec function call, ABI encoded with parameters.
- portal contracts sends message to inbox.
- inbox receives and sends message to L2.
- Message is held on the L1-->L2 append only tree
- When message is consumed and user has the right secret for that message, a nullifier is emitted.

Message from L2 to L1:

Link to Aztec's Thursday's presentations: Protocol Webinar Series - Google Drive

Aztec developement notes: GitHub - AztecProtocol/engineering-designs: Internal engineering designs

Protocol limitations: aztec-packages/noir-projects/noir-protocol-circuits/crates/types/src/constants.nr at aztec-packages-v0.49.1 · AztecProtocol/aztec-packages

10 tps/second on testnet. Testnet coming in December/January. Noir compiler audit in January 2025.

Private token implementation

Assumptions and requirements of CMTAT Private - V1

- Assumption: TotalSupply should stay public and be updated according to the mint and burn.
- Assumption: issuer address can be publicly known
- Assumption: admin address can be publicly known
- Assumption: we want to allow third parties to execute transactions on behalf of our users, so we use auth witness when transferring
- Assumption: there is no auth witness in “mint” and “burn“ function, as a third party is not allowed to mint/burn in any case (will not have the authorisation), only the issuer can do it.
- Assumption: admin cannot be changed. issuers can be added or removed by admin
- **TotalSupply - Public**: For a particular CMTAT token, any person may know the total number

  of tokens in circulation at any point it time.
- **BalanceOf - Private:** For a particular CMTAT token and a particular user, no one, apart from the issuer, should know the number of tokens currently recorded on the user's ledger address.
- **Transfer - Private**: Users may transfer some or all of their tokens to some other ledger address

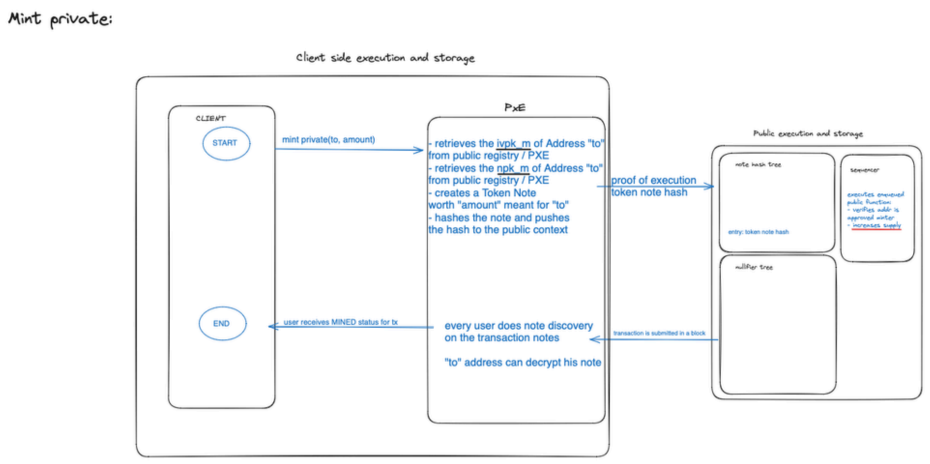
  (that the transferor does not necessarily control). According to the above functionality, a transfer must be private, such that no one apart from the parties involved and the issuer should know that a transfer has occurred, the transfer participants, and that a certain amount has been transferred.
- **Mint - Private:** Issue a given number of tokens to a given ledger address. The issuer and the given address should be the only ones that know that a transaction is happening. Only the issuer and the receiving address should know the amount minted to the receiver.
  - Note - **Public**: according to the assumption, the total supply will increase accordingly in a public function, and thus the new total supply will be visible to everyone. The supply change amount will be traceable to that particular private proof.
- **Burn - Private:** Issuer burns (destroy) a given number of tokens from a given ledger address. The issuer and the given address should be the only ones that know that a transaction is happening.
  - Note - **Public**: according to the assumption, the total supply will decrease accordingly in a public function, and thus the new total supply will be visible to everyone. The supply change amount will be traceable to that particular private proof.

- **compliance/audibility**: give users the option of sharing private transaction details with a trusted 3rd party. A user can optionally share "shareable" secret keys, to enable a 3rd party to decrypt the following data:
  - Outgoing data, across all apps,
  - Outgoing data, siloed for a single app,
  - Incoming internal data, across all apps,
  - Incoming internal data, siloed for a single app
  - Incoming data, across all apps

Storage

- issuer_address: `SharedMutable<AztecAddress,>` the address of the issuer, which serves as a base reference to encrypt user’s notes. As it is a SharedMutable, it can be changed if compromised.
- balances: `BalanceMap<TokenNote>` - token balance of every user inside its PXE. Address → PrivateSet<TokenNote>. The balance of a user is the sum of the amount of all of his private TokenNote.

Mint private


Mint private:
Client side execution and storage
CLIENT
START
mint private(to, amount)
PxE
- retrieves the ivpk_m of Address "to" from public registry / PXE
- retrieves the npk_m of Address "to" from public registry / PXE
- creates a Token Note worth "amount" meant for "to"
- hashes the note and pushes the hash to the public context
proof of execution
token note hash
Public execution and storage
note hash tree
entry: token note hash
sequencer
executes enqueued public function:
- verifies addr is approved minter
- increases supply
nullifier tree
transaction is submitted in a block
every user does note discovery on the transaction notes
"to" address can decrypt his note
user receives MINED status for tx
END

Issuer :

the new notes of the recipient are encoded and broadcasted to the issuer.

Failure cases:

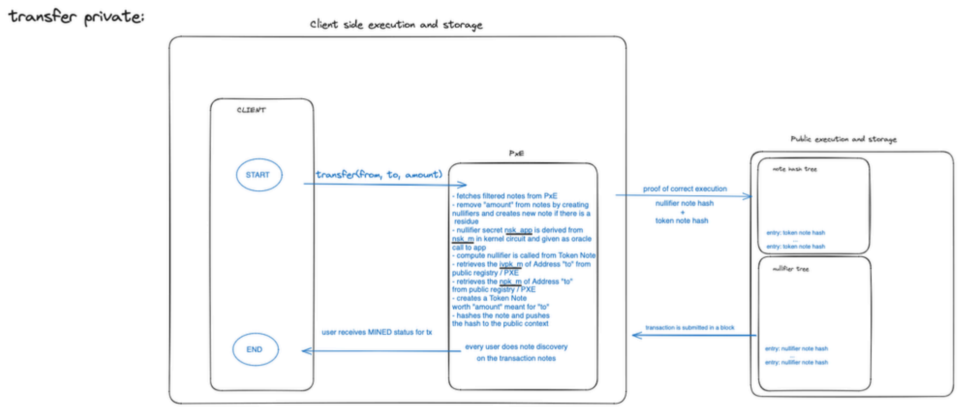
- Enforcement module: if the “to“ address is frozen, the mint will fail.
- Authorisation module: if the caller doesn’t have the minter role, the mint will fail
- Pause module: if the contract is paused, the mint will fail

Limitations:

According to protocol limitations, only 4 encrypted logs can be emitted in a function call and only 4 private functions can be called from a function call. As we have 2 encrypted logs emitted in the mint function, our bottleneck is the encrypted logs, which means that we can only batch 2 mint functions at the same time.

Transfer private

- As nsk_m cannot be inserted in an app circuit, it is hardened to create the nsk_app, which is used in the app circuit as the secret of the nullifier computation.


transfer private:
Client side execution and storage
CLIENT
START
transfer(from, to, amount)
PxE
- fetches filtered notes from PxE
- remove "amount" from notes by creating nullifiers and creates new note if there is a residue
- nullifier secret nsk_app is derived from nsk_m in kernel circuit and given as oracle call to app
- compute nullifier is called from Token Note
- retrieves the ivpk_m of Address "to" from public registry / PXE
- retrieves the npk_m of Address "to" from public registry / PXE
- creates a Token Note worth "amount" meant for "to"
- hashes the note and pushes the hash to the public context
proof of correct execution
nullifier note hash
+
token note hash
Public execution and storage
note hash tree
entry: token note hash
...
entry: token note hash
nullifier tree
entry: nullifier note hash
...
entry: nullifier note hash
transaction is submitted in a block
every user does note discovery on the transaction notes
user receives MINED status for tx
END

Discussion between JP, Ryan and Gustave: The issuer cannot do a force transfer on behalf of the user, as he would do in the CMTAT. The solution is that in the case where we want to have the same behaviour as a force transfer, we freeze the account.

Issuer :

the added notes from sender and recipient are encoded and broadcasted to the issuer.

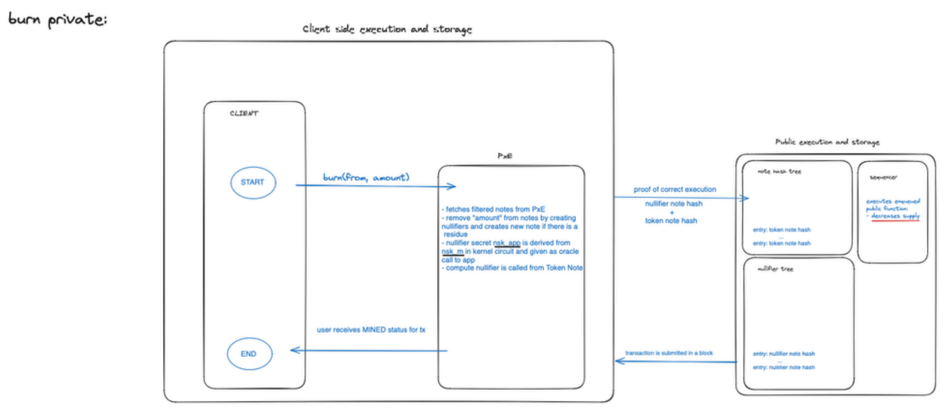
Failure cases:

- Enforcement module: if the “to“ address is frozen, the transfer will fail.
- Enforcement module: if the “from“ address is frozen, the transfer will fail.
- Validation module: if operations are enabled, the module checks if “from“ or “to“ should be restricted.
- Pause module: if the contract is paused, the transfer will fail

Limitations:

According to the following document, only 4 encrypted logs can be emitted in a function call. As the mint already emits 4 (2 for user, 2 for issuer), we can only have 1 transfer in the transfer batch.

Burn private


burn private:
Client side execution and storage
CLIENT
START
burn(from, amount)
PxE
- fetches filtered notes from PxE
- remove "amount" from notes by creating nullifiers and creates new note if there is a residue
- nullifier secret nsk_app is derived from nsk_m in kernel circuit and given as oracle call to app
- compute nullifier is called from Token Note
user receives MINED status for tx
END
proof of correct execution
nullifier note hash
+
token note hash
transaction is submitted in a block
Public execution and storage
note hash tree
entry: token note hash
...
entry: token note hash
sequencer
executes enqueued public function:
- decreases supply
nullifier tree
entry: nullifier note hash
...
entry: nullifier note hash

Issuer :

the new notes of the recipient (if any remaining) are encoded and broadcasted to the issuer.

Failure cases:

- Enforcement module: if the “from“ address is frozen, the burn will fail.
- Authorisation module: if the caller doesn’t have the burner role, the burn will fail
- Pause module: if the contract is paused, the burn will fail

Limitations:

According to protocol limitations, only 4 encrypted logs can be emitted in a function call and only 4 private functions can be called from a function call. As we have 2 encrypted logs emitted in the burn function, our bottleneck is the encrypted logs, which means that we can only batch 2 burn functions at the same time.

Security & confidentiality concerns

- Private mint call to public function:
  - will reveal minter addr: as it is a parameter in the public function call. It is the issuer, which address is already known, but still. Private to public function calls pose a problem, as they also reveal that the contract was called.
  - An out-of-protocol option to randomizing `msg.sender` (as a user) would be to deploy a diversified account contract and route transactions through this contract. Application developers might also be able to do something similar, to randomize the `msg.sender` of their app contract's address.
  - In the case of our token, when an issuer mints tokens, it is publicly known how much tokens he mints. This means that if the issuer mints “on-demand“ (every time a user wants to mint some tokens, the issuer mints) then there is a leak of information. This can be mitigated by the issuer minting a fixed amount of tokens at a certain point in time (= circulating supply), and then privately distributing to the users, thus revealing absolutely no information.
  - will leak the amount being minted, as the amount is passed to the public function from the private one
  - public transaction will be traceable back to the private proof
  - will leak that a private function (private_mint) has been called
  - will NOT leak the address to which this amount is being sent.
- We must add randomness to the note hash when creating it
- We must add a secret, for example nsk_app when calculating the nullifier of a note so that no link can be done between a note and its nullifier
- Algorithm to harden nsk_m into nsk_app is optimized, and not guaranteed to be secure as of now. See: Default Keys Specification | Privacy-first zkRollup | Aztec Documentation
- Note encryption should be constrained: We could make note encryption and tagging unconstrained, as this is allowed, but we don’t want to. Unconstrained note encryption is done when the sender has an incentive to send correct information to the receiver, as no one proves and verifies it. However, in our case the sender is in no way incentivised to do the right thing. For optimisation purposes, unconstrained might be good in some places.

Modules

Abstract contracts do not exist in Aztec Noir, thus the modules were separated in the form of interfaces and implementations.

Inheritance does not exist neither, which means that every function that can/should be called from a user needs to be exposed in the main contract.

As a consequence, not everything can be displaced from the main contract (i.e: Mint/Burn/transfer are all in the main contract), and most functions are exposed in the main contract. There are 34 functions in the main contract.

Authorisation Module (= access control) - Public:

This module is used by the other modules and by the mint and burn functions. Modules only need to call the `only_role` function, that tells **publicly** if an address has the sufficient roles for the action, otherwise reverts.

The default role is the `DEFAULT_ADMIN_ROLE`. This default role can grant other roles.

This module implementation is quite cumbersome, as in the main contract, an instance of this module is passed to each function call. This is done because the object is unique, and we cannot pass it as a context (for now at least, until a working implementation is found).

Validation Module - Shared:

This module is called only when doing transfers. The `operateOnTransfer` function, which is used in a private context, is call by the transfer function. Each user flag update will be delayed by `CHANGE_ROLES_DELAY_BLOCKS`.

If no operations are enabled, no checks are done, but the function is still called. Operations can be enabled or disabled, and there also is a delay. For now no operations can be added, there is only blacklist/whitelist and sanction list. Sanction list is not implemented.

This delay is caused by the fact that the roles are stored in a Shared Mutable variable type. This is actually needed to preserve privacy when doing a private transfer between two users, all while maintaining the strict rule that no tokens should be transferred from/to a blacklisted address. Indeed, as seen in the concepts part, SharedMutable allows a shared state between public and private user's executions contexts.

The delay also causes some problems, as we might imagine that a user that sees that he is going to be blacklisted in a certain number of blocks sends his funds to an address that is not blacklisted. This problem has no solution for now.

We also need to think about: is the shared state going to be changed often or not? If it is not, then shared mutable is an okay solution, otherwise it is a problem.

- **Theoretical** (i.e not applicable) **solution 1:** Using a SharedMutable is essential, because otherwise you would use a PublicMutable, which means that the user calling the transfer function needs to call a public function to read the PublicMutable variable. This leaks the sender's address. One possible solution to the leak problem might be to hide the caller address, which can be done using: Diversified and Stealth Accounts | Privacy-first zkRollup | Aztec Documentation . If reading publicMutable did not leak user address, then sharedMutable would be useless.
- **theoretical solution 2:** have a counter that is set when the SharedMutable is changed. For the COUNTER amount of time, token contract is paused, to prevent any blacklisted address to be able to retrieve funds. This solution is horrible UX and Devex as the issuer needs to manually unpause the contract (the contract cannot unpause itself (it could with Chainlink automation but out of scope))
- **Practical solution 3:** if we whitelist instead of blacklisting, the problem that we will have is that a new whitelisted address will not be able to transfer funds directly, which is not really a big issue.

Pause Module - Public:

The pause module is a public mutable. The functions to set and unset the pausable flag are protected under Access Control.

The pause check is done in public state for mint/transfer/burn.

Enforcement Module - Shared:

This module is called in mint/transfer/burn to check if an address has been freezed or not. Contrary to the validation module, this module is mandatory. Changing and address to freezed has a delay, as the value is a Shared Mutable.

"Freeze address" note: The enforcement has a delay, and thus has the same problem as the validation module. What can be done in this module is that before freezing some accounts, we pause the contract for the delay time. Then we unpause it. This requires manual pause/unpause.

How is issuer able to view all transactions and notes?

- find a way so that the issuer can see all transaction. Multiple options:
  - Emit events that can be viewed by a 3rd party by inputing his ivpk_m (ivpk_m_issuer)
    - Questions: here, we are not able to see exactly his actual token holdings, but we are able to see all of his transfers.
    - we can see his token holdings at a certain point in time by using the history library
  - When emitting a note, we can emit a duplicate with the issuers ivpk_m_issuer
- What is the best way to have a third party be able to look at a wallet balance of someone's private tokens balance on demand? An issuer has a ivpk_m_issuer. Multiple options:
  - let it see all the transactions with an emit encrypted with the issuers viewing key, the issuer needs to reconstruct and keep track of the state. However, anyone can emit events, and it is cumbersome for the issuer.
  - Have note encrypted twice on the user's pxe, one for himself and one for the issuer. Problem is that we double the notes needed, and we also need to keep track of them in our own pxe.
  - Have an app siloed key that the issuer can use for decrypting any note in the note hash tree of this app.

    - make a shared encryption/decryption key. Encrypt note once but two people can decrypt it.
    - when hardcoding the viewing key, make it rotatable and updatable
    - pas trop de réponses encore a la question, ils sont en train de travailler sur tout ce qui est encryption

L2 <> L1 token

- address is private
- amount is private
- Shield/unshield pattern

L1 to L2 private token transfer:

- L1 token is sent to the inbox contract from portal contract.
    - The portal contract computes the L2 message selector that calls the mint_private function on L2.
    - The portal contract transfers token amount from user to portal contract
- Inbox contract sends message to L2 append only tree
-
-

Miscellaneous / other concerns

- The wallet should implement note discovery and tagging mechanisms, not the application.
- Mint function: should we restrict the “to“ address to not be the issuer, in order to prevent a “malicious“ issuer from not revealing the ”real” supply of the token, ex: by minting 100% of tokens to himself, he can hide the supply held by users.
- Can a user modify a token contract function? No, it is not possible as each function is committed on the public state.
- Encryption of note emission is done with AES128, we do not know as of now if the encryption with AES is constrained at the protocol circuit.
- Notes are linked to their transaction hash (to verify), because they are in the same transaction object when waiting in the mempool.
- The transaction object cannot be modified between the point when it has been locally proven to the point when it reaches the sequencer, because the output of the private kernel circuit (the protocol circuit) is the input to the public kernel circuits, which also verify.
- For now, we cannot know who sent us a private note. When receiving a note, we can link it to a certain transaction (a transaction object), but we cannot know who is the sender. One solution would be to enshrine the sender address inside a note field. But we need to see confidentiality concerns.
- The transaction hash is always emitted during local execution as the first nullifier of the transaction, in order to prevent replay attacks. This is enforced by the private kernel circuit.

Issues and (maybe) solutions of doing a token contract on Aztec

- We’d love for the issuer to be able to see the private balance of the user at any time. Maybe this could be done with a special note to which the user and the issuer share a special viewing key. This however raises some other questions:
    - Will the issuer need to have one of those keys for each user? Key management becomes hard
    - Will these notes be differentiable from other more standard notes?
    - We will need to come up with a new encryption/decryption/viewing of our special notes
- The issuer should be able to force the transfer of notes, according to Swiss law. This is not possible in Aztec as it would require that the issuer can nullify user’s notes without user’s consent. This can be possible if issuer has Nullifier keys, but it is quite a huge concern.
    - The solution is that in the case where we want to have the same behaviour as a force transfer, we freeze the account.
    - If the account is frozen indefinitely, then we should decrease the circulating supply. However we don’t know the amount of tokens that that user holds. This is an issue
    - By keeping track of user’s token amount, we can actually know how much the frozen address has.
- The freezing and blacklisting of addresses takes a certain number of blocks, due to the nature of the sharedMutable type in Aztec.

    - If this is okay, do nothing
    - A solution might be to encrypt who is blacklisted with a key. But idk how it would work
- According to protocol limitations, only 4 private calls can be made from a private function. Which means that batch functions are limited. Also, only 4 encrypted notes can be emitted in a function call, which further limits batching. See more in mint/burn/transfer spoiler tabs.
- We may need to expose in the main contract the `schedule_delay_change` function for every SharedMutable, which is cumbersome.

Upgradability / proxy

See contract instance and contract classes to understand this better. Contract instances | Privacy-first zkRollup | Aztec Documentation

Comparison between CMTAT and Aztec Token

Version CMTAT

What can we actually do ?

- **Mint/transfer:** they behave the same way as in CMTAT. There is for now no knowledge of who sent you notes however
- **Burn:** We can do burn_from with allowance.
- **Validation Module:** whitelisting and blacklisting is enabled on demand. The rule engine has been merged into the validation module. Thus we only have one interface that manages both and is always deployed along the main contract. The functionalities are private. Storage can be read in public.
- **Pause Module:** same functionalities as CMTAT. Pause is public and instantaneous.
- **Enforcement Module:** Freeze and unfreeze supported. The functionalities are private. Storage can be read in public.
- **Access Control (Authorization) Module:** same functionalities as CMTAT. Admin has the default role, which he can use to grant roles to himself and/or to other addresses.
- **Credit Events Module:** same functionalities as CMTAT.
- **Debt Base Module:** same functionalities as CMTAT.

What will we be able to do in the future?

- **Batched mint/transfer/burn:**
    - Now, the protocol limitations are fairly high. We can only have 4 private calls in a function call, and we can only have 4 encrypted events emitted per function call. Thus batching is very limited. In the long run these protocol limitations will be lifted, and will enable us to do batched transactions. The logic is already implemented in the contracts.
    - These functions are not separated into their own “abstract contract” as it does not exist in Aztec. We could put them in a library but this would mean much more boilerplate code. They may improve composition/abstraction in the future.
- **Validation module:**
    - there is a limitation regarding shared mutable delay. This means that when modifying the whitelist/blacklist, there is a delay of a certain number of blocks (it could translate to a range from minutes to a few hours) until the new values are reflected in the blockchain.
    - sanction list not yet enabled, as there is no onchain list such as Chainanalysis on Ethereum.
- **Enforcement Module:** there is a limitation regarding shared mutable delay. When modifying freezed accounts, there is a delay of a certain number of blocks.
- **We cannot know who sent us the encrypted notes.** But we can link notes to a transaction hash. This is a problem regarding accounting. What can be done is to create a new field in a note with the address of the account that has created the note.
- Better (maybe) audit from the issuer: User’s may, in the future, be able to share a shareable key for audit purposes.
- **Events:** they are not yet enabled because they are cumbersome, as they can only be in the main contract for now, and make the code very long.

What will we never be able to do by design?

- We will never (unless Aztec want to have a scandal as big as Qedit) be able to burn someone else's tokens without some approval from his side.
- We will never (at least until a breaking discovery) be able to have a shared state (public and private) that has no delay when changed. This is a problem by construction.

Setup

Development environment

Setup:

Get the sandbox, aztec-cli and other tooling with this command:

```
bash -i <(curl -s install.aztec.network)
```

Install dependencies for Sandbox deployment:

https://docs.aztec.network/getting_started/aztecjs-getting-started

```
yarn init -yp
yarn add @aztec/aztec.js @aztec/accounts @aztec/noir-contracts.js typescript @types/node

downgrade node to 20.14.0 (until fixed with Aztec team)

aztec-nargo compile

//create the corresponding typescript artifacts
aztec-builder codegen -o src/artifacts target

Note: check that dependencies on Nargo.toml and dependencies on package.json are the same version

```

Sandbox testing:

cheat codes for sandbox: https://docs.aztec.network/reference/sandbox_reference/cheat_codes

Questions:

- est ce qu'on peut avoir notre propre verifier qui est propre a nos preuves et le deployer sur la sandbox? Generate a Solidity Verifier | Noir Documentation aztec-packages/yarn-project/end-to-end/src/fixtures/setup_l1_contracts.ts at 10048da5ce7edfe850d03ee9750 5ed72552c1dca · AztecProtocol/aztec-packages. This will only be useful with token bridges, there is an implementation on how to do that: https://docs.aztec.network/tutorials/contract_tutorials/advanced/token_bridge
- What happens when I fetch public master keys getters. Is it based on historical state so I can access it from a private function? If so, what happens if the keys are not in the historical storage, I register them on behalf of a user (see if this is possible too), but the user has created them between: transaction_timestamp - historical_fetch. keys | Privacy-first zkRollup | Aztec Documentation Do the keys that I fetch from other users go to my private keystore? Is thus the keystore a "key buffer"?
- Can I prevent against key rotation if I have the npk_m_hash as a value in my note instead of only npk_m?
- Why do we have this: balances: BalanceMap<TokenNote> ? Is it because we can have multiple accounts in a same pxe? Is it normal that a user is able to see notes of another user in the pxe? Yes, we can have multiple accounts in the same PxE. Notes will be encrypted in the PxE, but as the keystone will have all the decryption keys of all the accounts of that PxE, it will be able to decrypt them anyways.

- see if the nonce is useful in the contracts, try to make it useful. Ask why it is there
- When testing, private amounts seem to be accessed from external users who should not be able to. Is it because notes are stored in the same pxe? But shouldn't they be encrypted with ivpk_m? There is no logic in the library for decryption, is it done at the kernel level? they are not (yet?) encrypted in the user's PxE. There should be one PxE by user, that can have multiple accounts. Notes are
- run different pxes on my computer: this is broken
- public transaction will be traceable back to the private proof in my private to public calls? Yes, as the public transaction will need to prove the private one
- How do unconstrained functions behave? Is unconstrained same as view attribute? Why is the balance_of_private function still unconstrained in the token contract? Couldn't it be with the private and view macros? Unconstrained functions are good to be used when we do not change the state of the network, or when we are doing an off chain call (for example when reading your own notes, you are doing it completely off chain, so you cannot constrain it). View functions need to be used from reading from public state, you cannot use unconstrained.
- "When executing a private function, its ACIR bytecode will be executed by the PXE using the ACVM. The ACVM will generate the witness of the execution. The proving system can be used to generate a proof of the correctness of the witness." Are the witnesses the private functions inputs that need to be kept private? The proving backend (Barretenberg) creates the proof and then the private kernel circuit verifies it?
-
- Each PxE has an instance of the ACVM right? Yes
- Will the user that is just blacklisted be able to make one more transaction before being publicly flagged because of the delay constant? Cat takes a look into it and tells me when she knows.
- I have a statement: private/public function are executed by their respective VM, a proof of this execution is then generated. Then this proof is fed to the private kernel circuits, which proves that it has verified that the proof respectes the constraints. Only then the proof is sent to the L1, which does the verification. Does the private kernel circuit also uses the backend barretenberg prover? Does the private kernel actually proves anything or does it only validates the ACVM execution proof? Does it generate a new proof that states that the proof is actually correct and well constrained?
- How can I move the blocks of Aztec? Not implemented yet, they may implement it following the question.
- when emitting events, the ivkp_m is used to encrypt an event to the receiver. We need to be sure that this key has been registered in the PxE. Is it the wallet's job to do that?
- Can I decrypt a Note with my PxE and then decide if I want that note inside my PxE or not? For example, I get memecoins airdropped to me, but they are worth nothing and I don't want them in my pxe database. Will this database get full at some point? They don't have an answer
- what's the difference between the note encryption constrained/unconstrained?
- I can burn other person's tokens if my burn function doesn't have an authwit protection? Clearly not, this is happening in my tests because there is only 1 PXE in the tests.
- When someone sends me a note, do I know the address of the sender? No, this is up to the dev to put in a Note field the sender of the note.
- When testing, private amounts seem to be accessed from external users who should not be able to they are not (yet?) encrypted in the user's PxE. There should be one PxE by user, that can have multiple accounts. All of these accounts can see the unencrypted PxE
- Ask about changes in TokenNote interface from version 0.48.0. What does the `TokenNoteHidingPoint` mean? What is the `compute_note_hiding_point` function? Where is it used?
- If I emit twice the same note, one for the user and one for the issuer, what do I put instead of the oak_m of the second encryption? I don't necessarily want the user to end up with two notes that are the same
- what happens if I change computer and my PXE is stored on my device?

TODO:

- Key rotation: https://docs.aztec.network/guides/smart_contracts/writing_contracts/common_patterns/key_rotation
- implement a proof for proving that you own less/more than a certain amount.

- look into overflow and test
- shareable key
- explore the new `[contract_library_method]` macro on Aztec

Examples:

aztec's example of private token contract with public and private transfers:
https://docs.aztec.network/tutorials/contract_tutorials/token_contract

DEX built on Aztec: GitHub - porco-rosso-j/aztec-dex-build

Homomorphic encryption? GitHub - jat9292/noir-elgamal: A Noir library for Exponential ElGamal Encryption on the Baby Jubjub curve GitHub - defi-wonderland/aztec-coin-toss-pvp we don't support FHE on Aztec yet as per: Sharing private data with Authentication Wi tness Based on recent discussions, it seems it'd be viable to have shared private state using FHE. But FHE is already fairly expensive on its own, and creating a ZKP over FHE becomes prohibitive, in particular if we expect end-users to be generating these proofs.

Token transfers flows: Transferring someone else's notes

Private token using Aztec: zkSNARKs & Homomorphic Encryption : Ethereum's Privacy New Frontier

Aztec explorer: GitHub - olehmisar/shieldswap: A Decentralized Exchange on Aztec Network

Other ways of doing encryption:

Zcash Shielded Assets (ZSA):

- implemented by qedit Secure data collaboration using Privacy Enhancing Technology | QEDIT
- In this video: https://www.youtube.com/watch?v=L_ZtZCUvDqw the following question is asked with the following answer: **Question**: Does the ZSA issuer has any way to track assets and transactions once owners have included the ZSAs in shielded transactions? **Answer**: Basically, no.
- **~~Thus, it is not possible for us to build a private CMTAT with their tech stack~~ Their proposal has been reject by Zcash community**

Polygon Miden:

Encrypted notes are not live yet on Miden. There are only private and public notes for now. "Support for `encrypted` notes will be added later on." from their Discord server.

ZAMA:

Zama - Fully Homomorphic Encryption

Aleo | Zero-knowledge by design

Home | Ergo Platform

Silent data:

We cannot use it at this stage. There is no public docs. Product is not there.

•