In Aztec Connect, users had to trust

that the Sequencer would actually submit encrypted note preimages to L1, as calldata. There was no enforcement at the protocol level, to ensure that every note commitment was accompanied by its corresponding encrypted note preimage.

We need to enforce this, when we build Aztec. Otherwise, an app's data might never get broadcast!

The Problem:

Suppose a user has a note_preimage

, and they want a guarantee that the encrypted_note_preimage = enc(note_preimage)

will be definitely

be submitted when the note_hash

is submitted.

Note: we don't care about how enc()

works, in this post.

The Public Inputs ABI for an Aztec function cannot be dynamic in size. It is a fixed-size struct. But we want to allow users to be able to design custom notes, which might have custom data sizes. Therefore, the encrypted_note_preimage

might be variable in size, depending on the app.

Proposal:

tl;dr

Expose, a new field log_hash

as a public input, which can be the hash of any data. Provide a way for that data to be submitted on-chain, re-compute the hash on-chain, and compare it against the public input log_hash

. Reject the rollup block if this check fails.

Essentially, we allow a single field element – compressed_log

– to represent "all of the data I wish to submit as an event"

. Naturally, this single field element will be a sha-256 hash of the unpacked event data.

I'm ignoring EIP-4844 here. I'll talk about it in the current world, where calldata is usually sha-256

-hashed in order to be validated. It's easy ™ to migrate these concepts to EIP-4844, eventually.

We'll add four new fields to the Public Inputs ABI of an Aztec function.

struct PrivateCircuitPublicInputs = { encryped_log_hash: Field, unencrypted_log_hash: Field, encryped_log_preimage_length: u32, unencrypted_log_preimage_length: u32, }

- encryped_log_hash: Field

: is a single field, which MUST be the sha256-hash of the encrypted

data we wish to submit via L1.

- unencrypted_log_hash: Field

: is a single field, which MUST be the sha256-hash of the unencrypted

data we wish to submit via L1.

- encryped_log_preimage_length

: the size of the preimage data, so that the gas cost of this request can be measured by circuits, without actually needing to feed in the variable-length data.

- unencrypted_log_preimage_length

: the size of the preimage data, so that the gas cost of this request can be measured by circuits.

Note: I've kept encryted

and unencrypted

data separate, for now. My thinking being: it'll be easier this way for an aztecRPCServer to identify which data it should attempt to decrypt. Otherwise, it could waste effort. Maybe there's a nice encoding which would allow the preimages to be combined, whilst also distinguishing which 'substrings' of the blob are encrypted vs unencrypted.

When a user submits their Private Kernel Proof to the tx pool, they will also need to provide the xxx_log_preimage

s in that submission.

For every tx which the sequencer includes in their L2 block, they will need to submit the xxx_log_preimage

s to L1.

TBD: this data can either be submitted as part of the 'main' rollup tx; or submitted in advance to a separate function, which would hash the data and store the hash for later querying by the 'main' rollup.

The xxx_log_hash

and xxx_log_preimage_length

fields are percolated all the way up to L1 (unless xxx_log_hash == 0

, in which case we can maybe optimise it away). The xxx_log_preimage

data blob is also submitted to L1, where it is sha256-hashed.

When the sequencer submits this L2 block to L1, the Rollup Contract will be able to decode the calldata to identify each individual xxx_log_hash

field of each individual tx. It can then compare this xxx_log_hash

value against the sha256(xxx_log_preimage)

:

require(encryped_log_preimage_length == encrypted_log_preimage.length); require(encryped_log_hash == sha256(encryped_log_preimage);

require(unencrypted_log_preimage_length == unencrypted_log_preimage.length); require(unencrypted_log_hash == sha256(unencrypted_log_preimage);

Why call it "log"?

Why not name it something specifically about "note encryption"?

Because this field can now be used more generally than just for note encryption!

Any event data can be emitted with this approach!

- Custom event information (encrypted or publicly visible)
- Custom note preimages
- Custom nullifier preimages (see Vitalik's nice blog for a nice example of why an encrypted nullifier preimage might be useful for apps!).

Note: This does not enforce that the encrypted data is correct

.

That's another problem, for another thread. Even for unencrypted blobs of event data, this does not enforce that the data is used

by the app circuit in any way. It's up to the app to ensure this (or to not bother ensuring this) via their Noir Contract logic!

All this proposal ensures is that the Sequencer cannot submit a valid rollup unless they also make all

requested event data available on L1.

Continuing the discussion further

The above is the meat of this proposal, and is hopefully not too controversial.

The following tries to standardise some encodings of ‘event data’, to make it easier for an aztecRPCServer to process that data.

It’s important to note that the below suggestions might be over-fitted to “brute force” syncing, where a user’s aztecRPCServer must trial-decrypt every

log.

We’re hoping to move away from that, eventually, but for now here are some ideas.

Standardising the compressed_log_preimage

data encoding:

Note:

If there’s an encoding (not to be confused with ‘encryption’!) which allows encrypted and unencrypted data to be smooshed together in one blob, that would be nice.

Regardless, we can devise an abi-encoding the data blobs, so that an aztecRPCServer can distinguish between different actual

events.

Here’s a suggestion:

- header = contract_address
- body = encoding of log_preimage
- Data submitted to L1: enc_1(header) || enc_2(body)

The idea here would be: the header

can be trial-decrypted (at least, in our initial protocol) using a user’s global enc/decryption key. It would also be much

faster to only trial-decrypt one field element, rather than the whole blob, because the user will be trial-decrypting millions of times (and so parallelising the decryption of the whole of every message would only waste cores that could otherwise be trial-decrypting the next headers).

Then once they have a match for the header

, we need to decrypt the body.

Option 1

- Use the same decryption key as was used for the header. I.e. one decryption key for all contracts.

Option 2

- Use a different decryption key per contract. Something like BIP-32 can be used to allow users to derive each-others’ encryption/decryption keys on a per-contract basis.

Option 3

- Allow Noir Contract developers to write custom encryption/decryption logic for how the body

of their contract’s encrypted logs can be decrypted. If a function_selector

for the Noir Contract’s custom decryption function is included in the header, the aztecRPCServer could pass the body of each successfully-decrypted header to the appropriate decryption function.

Regardless of which option we choose, I think we will need to include in the body

the function_selector

(s) of the Noir Contract functions which can serialise custom notes and compute custom nullifiers.

Rolling with Option 3

, for example's sake:

- header = contract_address || body_decryption_function_selector
- Here's a suggestion for the encoding of the body

:

number_of_events || first_event_processing_function_selector || first_event_length || first_event_data || etc...

The body_decryption_function_selector

is just a suggestion to allow custom decryption algorithms to be written in Noir Contracts. To be discussed more.

The first_event_processing_function_selector

is some function selector which can 'process' this log data. It might be a note serialisation (and storage) function. It might compute the nullifier of a note. It might process some app-specific data.

Note: standardising an encoding is really only for the benefit of the aztecRPCServer

software, so that it can process blob after blob of event data. Maybe there's a better encoding. Maybe various standards of encodings will be developed over time by the community. Noir Contracts themselves could dictate the encoding and be fed the opaque blobs to decrypt.

Example

Emitting data required for a simple transfer

, but not performing the encryption calculation within the circuit.

Suppose, outside of the circuit, the following data has been encoded, and then encrypted:

- header

: * contract_address: 0x...

- body_decryption_function_selector

(suggestion under option 3 above)

- contract_address: 0x...
- body_decryption_function_selector

(suggestion under option 3 above)

- body

: * Number of events coming from the transfer

function: 1

- first_event_processing_function_selector

: process_new_note_preimage.function_selector

- Length of the first event: 4 * Field.length
- First event arg (sender): 0x...
- Second event arg (receiver): 0x...
- Third event arg (value): 10
- Fourth event arg (randomness): 0x...
- Number of events coming from the transfer

function: 1

- first_event_processing_function_selector

: process_new_note_preimage.function_selector

- Length of the first event: 4 * Field.length
- First event arg (sender): 0x...
- Second event arg (receiver): 0x...
- Third event arg (value): 10
- Fourth event arg (randomness): 0x...

Noir Pseudocode:

/// File: zk_token.nr

use aztec::events::emit_encrypted_log_hash; use aztec::notes::ValueNote; use aztec::notes::ValueNoteLength; // other imports not shown

contract ExampleToken { balances: Map>;

```
// Note:
// In this example, we won't go down the rabbit hole
// of _proving_ correct encryption, since we want to
// focus on the topic at hand.
// So we just pass-in the sha256 hash of some encrypted
// data blob. The circuit doesn't isn't checking it's
// correct, for the sake of keeping the example brief!
//
fn transfer(
    amount: Field,
    sender: Field,
    recipient: Field,
    hash_of_encrypted_recipient_note: u256
    preimage_length: u32
) {

    // Not shown:
    //   - Do the logic of a transfer, as we've seen many times.

    emit_encrypted_log_hash(
        hash_of_encrypted_recipient_note,
        preimage_length
    );
}

// Provides an example of an unconstrained function which
// could process a note for the aztecRPCServer,
// provided its function_selector is included in the
// encrypted log!
unconstrained fn process_new_note_preimage(
    note_data: [Fields, ValueNoteLength]
) {
    let contract_address = note_data[0];
    let storage_slot = note_data[1];
    let note_fields = note_data.slice(2);

    // The oracle will keep notes in a special db collection.
    oracle.store_as_note(
        contract_address,
        storage_slot,
        note_fields
    );
}

}
```

/// File: events.nr

// Provides a way to emit an opaque hash value, without // actually validating or computing that hash value from // any underlying data. // Useful for applications where proving correct encryption // isn't needed, and all we need is to ensure data availability // on L1. fn emit_encrypted_log_hash(encrypted_log_hash, encrypted_log_preimage_length) { // Ensure it hasn't already been pushed-to. assert(context.public_inputs.encrypted_log_hash == 0);

```
context.public_inputs.encrypted_log_hash = encrypted_log_hash;
context.public_inputs.encrypted_log_preimage_length = encrypted_log_preimage_length;
```

}

global encrypted_log_preimage: [Field]; // Would be cool, if possible, // to derive its length at // compile-time.

// Not used in this example, but interesting to see:

// Pushes more event data to an in-memory array. // // Once all events have been pushed (at the end // of the main function), we can compute the // hash through a separate function. fn push_encrypted_log_data(encrypted_log_data: [Field; N]) { // E.g. push the length of the next event first: encrypted_log_preimage.push(N); // Then push the actual data: encrypted_log_preimage.push_array(encrypted_log_data); }

// Hash whatever event data is in memory: // Pseudocode: because there's som tricky 'length' concepts here. fn compute_and_emit_encrypted_log_hash() { log_hash: Field = pedersen(encrypted_log_preimage);

emit_encrypted_log_hash(log_hash, log_hash.length());

}

Edit: significant edits, following Santiago's message.

Edit: rename: aztec-node → aztecRPCServer

Edit: rename: compressed_xxx_log

to xxx_log_hash