This post is intended to those interested in using Taiga with some understanding of what they are doing.

Introduction

Taiga is a component of Anoma that is used to produce valid state transition proposals. Taiga defines what a valid transaction is and provides the means to produce and verify transactions.

We can think of a partial transaction as a state transition proposal

. And partial transactions are the building blocks to create transactions

(i.e. state changes) in Anoma, a privacy-preserving, distributed, trust-aware operating system

.

This is a detailed explanation of partial transactions by example. In a tutorial-like fashion, we are going to create a partial transaction in Taiga from scratch using the example of a two-party barter in which we want 1 BTC in exchange or either 2 ETH or 5 NAM. The goal is to understand the lifecycle of a transaction and the different parties involved. We will explore the Taiga API along the way.

Background context

Our aim is to submit a partial transaction that looks conceptually like this in a very simplified way:

Input

Ouput

1

NF(Note(5, Token(NAM), …)), {\pi_i}

Com(Note(1, Intent, …)), {\pi_i}

2

NF(Note(2, Token(ETH),…)), {\pi_i}

Com(Note(1, Token(BTC), …)), {\pi_i}

Here Com(...)

means commitment, i.e. a hash with hiding properties using randomness. In this case, it is the commitment of a note. A note contains, among other fields, a value

and a note type

. For example, Note(5, Token(NAM), ...)

refers to a note of type NAM token

and value 5. NF(...)

is the nullifier (or nonce) of a note that we use to consume it. {\pi_i}

represents a set of proofs.

In Taiga, the state of the system is determined by the Unspent Transaction Outputs

(UTxOs), i.e. assets are stored in unspent outputs, rather than in accounts. In the UTXO model, a transaction has inputs and outputs, where the inputs are unspent outputs from previous transactions. In this model, a note encapsulates a subset of the state of the system - e.g. I own 5 ETH (but I may own more in other notes). Taiga's closest state model is Cardano's Extended UTXO model, which generalises the concept of the address

of a note to some arbitrary logic.

Since applications in Taiga are shielded by default (i.e. both the data and the type of application are hidden from a third party), a partial transaction contains only some proofs about the validty of the predicates of the notes involved, and some other data, including the nullifiers of the input notes and the commitments of the output notes. A partial transaction does not contain the notes themselves. The notes in the "input" column are sometimes called "consumed" or "spent" notes, since they need to exist in the first place and we nullify them once the transaction is executed so that they can't be used again (i.e. they are consumed

). These notes exist in a data structure named Merkle Tree, a binary tree of hashes suitable for verifying data integrity and inclusion. We call this instance of a Merkle Tree a note commitment tree

. On the other hand, the "output" notes don't exist yet in the note commitment tree. Only if the transaction is valid and executed, they get inserted in the tree (i.e. they are created

), so that they can be consumed

in a future transaction, and so on.

A partial transaction

is a data structure that contains the proofs that two input notes can be consumed and two output notes can be created, together with some public data (e.g. nullifiers and commitments) and other rules that Taiga enforces. In a partial transaction, the sum of the note values of the same note type (e.g. ETH token is a note type) need not be zero. In other words, the sum of the input note values of every note type doesn't need to be equal to the sum of the values of the output notes of the same note type. We then say that a partial transaction doesn't need to be balanced

.

A transaction in Taiga is just a balanced

set of partial transactions, i.e. for any given note type

, the values

of the consumed notes is equal to the values of the created notes.

Intents

An intent is a proposal about the outcome of a state transition. In our case, the state transition proposal says, informally, "in exchange of 1 BTC, spend either

5 NAM or

1 ETH". How does Taiga do this?

As illustrated in the table above, we input (consume) both NAM and ETH token notes by publicising their nullifiers. We also create a BTC token note commitment of value 1 with us as a receiver. As discussed here, the sender validity predicate needs to be satisfied when creating a token note. We create an intent note commitment, an "or

" relation stating the conditions in which it (the intent note) can be consumed, that is, if a NAM or ETH token note of values 5 or 2, respectively, then both the newly created intent note and BTC note can be consumed in the same transaction.

Since a transaction is a balanced set of partial transactions and a partial transaction is atomic, the NAM and ETH token notes can be consumed if and only if the "or

" relation intent note and the BTC token note that we create are also consumed in a different partial transaction of the same transaction. It is a transaction that needs to be balanced, not a partial transaction. We'll dive into this later.

Notes

As mentioned, a note represents a subset of the state of the system. Every note is characterised by the verifying key vk

(i.e. a succint representation of the validity predicate) and the encoded data in app_data_static

. These two fields constitute the type

of the note. The note type is a unique reference to what we commonly call application

. For example:

- In the case of the "ETH" and "XAN" token notes,
- vk

represents the token circuit (i.e. the structure of a circuit that can be shared among different tokens)

- app_data_static

encodes whether it is an “ETH” or a “XAN” token.

- vk

represents the token circuit (i.e. the structure of a circuit that can be shared among different tokens)

- app_data_static

encodes whether it is an “ETH” or a “XAN” token.

- In the case of an intent

that expresses an or relation

of two conditions, * vk

is the circuit description of this relation

- app_data_static

encodes the two conditions

plus the address of the receiver, i.e. the public key of the party who will be able to decode and consume the encrypted note in the future.

pub fn create_intent_note(condition1: &Condition, condition2: &Condition, receiver_address: pallas::Base, rho: Nullifier, nk_com: NullifierKeyCom, rseed: RandomSeed) -> Note { let app_data_static = OrRelationIntentValidityPredicateCircuit::encode_app_data_static(condition1, condition2, receiver_address,); Note::new(app_vk: *COMPRESSED_OR_RELATION_INTENT_VK, app_data_static, app_data_dynamic: pallas::Base::zero(), value: 1u64, nk_com, rho, is_merkle_checked: false, rseed,) }

- vk

is the circuit description of this relation

- app_data_static

encodes the two conditions

plus the address of the receiver, i.e. the public key of the party who will be able to decode and consume the encrypted note in the future.

The other parameters we need for creating a note are:

- app_data_dynamic

encodes additional data that doesn’t affect the note type (e.g. additional authorisation mechanisms)

- value

is used to ensure the balance in a transaction. It represents the quantity of a particular note type. Our partial transaction is not balanced because, among other conditions, we are not consuming

an intent note of the same value

as the intent note we are creating

. In the case of an intent, we can choose any unsigned integer.

- nk_com

is the commitment to the nullifier key.

- rseed

is a random finite field element used to compute \psi=PRF_{\psi}(0,rseed,\rho)

. \psi

is then involved in computing the nullifier of the note nf

when it is consumed.

- rho

(\rho

) is another finite field element that we need in order to compute the nullifier nf

. Formally, nf =Extract_P([PRF_{nk}(\rho) + \psi \ mod \ q] * G + cm)

. Like in Orchard, the nf

of an input note is the rho

of an output note

. This constraint guarantees the uniqueness

of the note. In our example, we can choose the nf

of either the “XAN” or “ETH” note as our rho

value.

Unchecked and dummy notes

If the is_merkle_checked

flag of a note is set to true

, the merkle path authorization (membership) of the input note will be checked (this and other Taiga-specific checks that ensure the validity of a state transtion are done in a circuit called the action circuit - we’ll eventually come to the action circuit). In the case of an intent, we set this flag to false

. This is because we are creating an intent note and spending it in the same transaction. In other words, we can’t insert the created intent note in this partial transaction into the note commitment tree since the partial transaction is unbalanced until we consume the intent note, and an unbalanced transaction can’t be executed. And we can’t consume a note if it doesn’t exist in the note commitment tree. So we set the flag is_merkle_checked

to false

to bypass the inclusion check of the created intent note in the note commitment tree when we consume the intent. Remember that a transaction is just a set of partial transactions that are balanced. A note with the is_merkle_checked

flag set to false

is called an unchecked note

.

We can create a dummy note

in a similar way. A dummy note is a note of value

zero, that is, it doesn’t affect the balance of a transaction. We inherit this terminology from Zcash. The validity predicate of a dummy note is the trivial one, yielding always true

. The rest of the fields can be of any value, since they will not be checked. The commitment of a dummy note still goes to the note commitment tree

to preserve the privacy properties of the system.

Encrypted notes

Since the commitment of a note that we store in the public note commitment tree

is a hash of the note (and we can’t retrieve a note from its hash), we also need to store the encrypted notes for future decryption somewhere else. For our purposes, we can assume the encrypted notes live in a key-value store where the key

is the note commitment and the value

is the encrypted note.

In fact, the first action that we need to do is retrieving some notes we own from the note commitment tree. A brute-force approach consists on using every leaf in the note commitment tree and aiming to decrypt the note with our private key. If successful, that means that we own the note commitment. An efficient algorithm for finding notes in a tree of hashes is an open research question. We further want to retrieve only certain notes, in this case only our “ETH” and “XAN” notes.

fn retrieve_owned_notes(&self, note_identifiers: Vec, sk: pallas::Base, pk: pallas::Affine) -> Result<Vec<(NoteType, Vec)>, APIError>

(Optional)

Roughly, this algorithm works as follows:

- Iterate over the leaves (i.e. the note commitments) of the Note Commitment Tree
- Look up the encrypted note that corresponds to each note commitment
- Attempt to decrypt each encrypted note with our private key
- If successful, check that the note type of the decrypted note is in the set of note types
- Return the decrypted note, its commitment and the path in the note commitment tree that as a proof of inclusion.pub struct NoteInTree { note: Note, note_cm: pallas::Base, merkle_path: MerklePath, }
- If successful, check that the note type of the decrypted note is in the set of note types
- Return the decrypted note, its commitment and the path in the note commitment tree that as a proof of inclusion.pub struct NoteInTree { note: Note, note_cm: pallas::Base, merkle_path: MerklePath, }

So we are able to retrieve our “ETH” and “XAN” notes, that is, the notes that we want to consume

. What do we need to create the output

notes of our partial transaction?

Validity predicates

A validity predicate (VP) is a boolean function that represents the logic of an application in a note. A transaction is considered valid when the VP for each note involved in the transaction is satisfied.

A validity predicate contains, among other fields, input notes

and output notes

as witnesses. This means that a validity predicate requires the notes involved in a partial transaction as inputs. At the same time, we require a verifying key

vk

to construct a note, which is a succint representation of a validity predicate, as we saw earlier. How can this circular dependency be?

The key realisation is that the vk

is encapsulating the fundamental structure

of a validity predicate and it doesn’t depend on neither the public inputs (instances) nor the private ones (witnesses). At a later stage, the prover inputs the instances and witness to an existing circuit to create a proof, and the verifier only needs this proof, the instances and the vk

to verify the validity of the computation. In other words, the structure of the circuit (i.e. gate configurations) is fixed, while the runtime values (i.e. witnesses and public instances) are specific to each execution of the circuit. In short:

1. We first create a vk

without knowledge of any witnesses or instances

1. We then create a note using the already computed vk
2. The prover (may be us or a solver) constructs an instance of the circuit using valid witnesses and instances.

Implementation-wise this is a bit inconvenient and somehow counterintuitive. We define the witnesses of a circuit as fields of a struct

. In our example:

[derive(Clone, Debug, Default)]

pub struct OrRelationIntentValidityPredicateCircuit { pub owned_note_pub_id: pallas::Base, pub input_notes: [Note; NUM_NOTE], pub output_notes: [Note; NUM_NOTE], pub condition1: Condition, pub condition2: Condition, pub receiver_address: pallas::Base, }

To compute the verifying key we need to create an instance of OrRelationIntentValidityPredicateCircuit

, but we don’t know the values of input_notes

, output_notes

or owned_note_pub_id

at this point. We need to mock them since the values of these fields here do not affect the value of the verifying key. The values we use for these fields are irrelevant for now. One way to do it is as follows:

lazy_static! { pub static ref OR_RELATION_INTENT_VK: ValidityPredicateVerifyingKey = OrRelationIntentValidityPredicateCircuit::default().get_vp_vk(); }

let vk = plonk::keygen_vk(&params, &OR_RELATION_INTENT_VK).unwrap();

As mentioned, when the Prover runs the circuit to create a proof, he will create another instance of OrRelationIntentValidityPredicateCircuit

with the correct witnesses.

Summary

Let’s summarise what we have achieved until now:

- We have shown how to retrieve and decrypt our XAN and ETH notes together with their note commitments and their authentication path in the note commitment tree

.

- We have created an intent note and a BTC token note and therefore we can also compute their commitments

In Part 2, we’ll show how we can then construct a partial transaction from the data gathered here.