This is Part 2

of a series to understand the lifecycle of a partial transaction in Taiga, using a two-party barter as an example. We'll continue mixing high-level concepts with some implementation details for clarity.

In Part 1, we managed to:

- retrieve and decrypt our XAN and ETH notes together with their note commitments and their authentication path in the note commitment tree

.

- create an intent note and a dummy note and therefore we can compute their commitments

Recall that a partial transaction looks roughly like:

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}

Building a partial transaction

Next, to create a partial transaction we need to run the validity predicates in each note and output the proofs

that confirm they were satisfied. Recall that in a partial transaction the validity predicates of every note involved need to be satisfied.

We can compute a proof that the validity predicate is satisfied as follows. The get_verifying_info

method belongs to the ValidityPredicateVerifyingInfo

trait which every validity predicate is required to implement:

fn get_verifying_info(&self) -> VPVerifyingInfo { let mut rng = OsRng; let params = SETUP_PARAMS_MAP.get(&12).unwrap(); let vk = keygen_vk(params, self).expect("keygen_vk should not fail"); let pk = keygen_pk(params, vk.clone(), self).expect("keygen_pk should not fail"); let instance = self.get_instances(); let proof = Proof::create(&pk, params, self.clone(), &[&instance], &mut rng).unwrap(); VPVerifyingInfo { vk, proof, instance, } }

It seems quite reasonable to ask: how many validity predicates does a note have?

As we have seen, a note has one

validity predicate, the one whose vk

is in the note type. However, this one validity predicate (call it application

) may require other validity predicates to be also satisfied. For example, a token application

may enforce a signature check, which is also a validity predicate. This separation of concerns is a key design choice in Taiga. So, how do we achieve this?

For one, the application

validity predicate of a note can read the other note fields, including app_data_dynamic

. This field contains arbitrary encoded data. It can contain the verifying keys vk

of many different validity predicates. In the case of a token application, we can hash the signature check verifying key in app_data_dynamic

. In the proving phase, we will need to satisfy these non-primary validity predicates as well. We sometimes call them subVPs

. This is what we're doing next.

We have now all the necessary data to create a partial transaction. To summarise, for each output note, we provide:

- The (decrypted) note
- The proof and the instances used in the validity predicate of the note. As the name VPVerifyingInfo

suggests, these three fields (vk

, proof

and instance

) are exactly what any verifier will need to verify that the validity predicate of the note was satisfied.

- The proof and instances of the possibly many validity predicates that are encoded in app_data_dynamic

.

In code, this translates to providing an instance of OutputNoteProvingInfo

:

pub struct OutputNoteProvingInfo { pub note: Note, app_vp_verifying_info: Box, app_vp_verifying_info_dynamic: Vec>, }

For the proving information of the input

note, need to also check that the notes already

exist in the note commitment tree. That is, we additionally provide:

- The authentication path of the note in the note commitment tree
- The root of the note commitment tree, which gives a compact and unique representation of the entire note commitment tree

pub struct InputNoteProvingInfo { pub note: Note, pub auth_path: [(pallas::Base, LR); TAIGA_COMMITMENT_TREE_DEPTH], pub root: pallas::Base, app_vp_verifying_info: Box, app_vp_verifying_info_dynamic: Vec>, }

Great! We are now ready to construct a partial transaction:

pub fn create_partial_transaction(input_info: [InputNoteProvingInfo; 2], output_info: [OutputNoteProvingInfo; 2],) -> ShieldedPartialTransaction

What happens next? Once a partial transaction is created, it is sent to the gossip network

, i.e. a network of (imbalanced) partial transactions that are waiting to be solved

. Before we go there, let's dive into what really is partial transaction.

Analysing a partial transaction

How does ShieldedPartialTransaction

look like? What is it that we share in the gossip network?

pub struct ShieldedPartialTransaction { actions: [ActionVerifyingInfo; 2], inputs: [NoteVPVerifyingInfoSet; 2], outputs: [NoteVPVerifyingInfoSet; 2], }

A (shielded) partial transaction only needs to hold the information to prove that the validity predicate of each input/output note was satisfied. That is, only the verifying key, the proof and the public instances of the validity predicate are necessary. In the code above, it means that input and output notes are an instance of the struct NoteVPVerifyingInfoSet

, which keeps only the data from app_vp_verifying_info

and app_vp_verifying_info_dynamic

that we had in OutputNoteProvingInfo

and InputNoteProvingInfo

. What is then that actions

field about?

```
pub struct NoteVPVerifyingInfoSet { app_vp_verifying_info: VPVerifyingInfo, app_dynamic_vp_verifying_info: Vec, }
```

Action circuit

Given the verifying key, the proof and the public instances of the validity predicates of the input and output notes, how do we know that the verifying key is the right verifying key of each consumed or created note and not an arbitrary one? How do we make sure the input notes can be consumed (i.e. they exist in the note commitment tree and their nullifier is not published)? We need extra logic and data. Here's where the action circuit comes.

The action circuit can be seen as the core Taiga circuit

, ensuring that the proposed state transitions follow the Taiga rules. The action circuit is unique in the sense that its checks don't depend on the applications involved and are the same for all transactions in Taiga. Concretely, it checks that

- For input notes:

- that they exist (in the note commitment tree)

- that they haven't been consumed (the nullifier is not in the nullifier set)

- that the proofs in app_vp_verifying_info

correspond to the notes being consumed

- that they exist (in the note commitment tree)

- that they haven't been consumed (the nullifier is not in the nullifier set)

- that the proofs in app_vp_verifying_info

correspond to the notes being consumed

- For output notes:
- that the proofs in app_vp_verifying_info

correspond to the notes being created

- that the note commitment is derived correctly

- that the proofs in app_vp_verifying_info

correspond to the notes being created

- that the note commitment is derived correctly

```
pub struct ActionCircuit { /// Input note pub input_note: Note, /// The authorization path of input note pub auth_path:
[(pallas::Base, LR); TAIGA_COMMITMENT_TREE_DEPTH], /// Output note pub output_note: Note, /// random scalar for net
value commitment pub rcv: pallas::Scalar, }
```

There is one action circuit per input/output pair. As we saw earlier, the nullifier of an input note is the random value ρ

of an output note, needed to guarantee uniqueness. This is also a check that the action circuit does.

The action public inputs consist of the output note commitments , the nullifiers

of the input notes, the root of the note commitment tree before the transaction occurs and the value commitment. The importance of exposing the nullifiers of the input notes and the commitments of the output notes is clear: to consume and create transactions by inserting the nullifiers in the nullifier set and the commitments in the note commitment tree, respectively. Each action circuit computes the balance

each input/output pair of the partial transaction and commits it into what is called value commitment

. Formally, a value commitment is defined as $cv = [v^{in}]NT^{in} - [v^{out}]NT^{out} + [rcv]R$

, where NT^{in}

, NT^{out}

are the note types of the input and output notes, and v^{in}

, v^{out}

are the values of each note. [rcv]R

achieves the hiding properties of the commitment, since rcv

is random (also called trapdoor). Implementation-wise, the public inputs of the action circuit look like this:

pub struct ActionInstance { /// The root of the note commitment Merkle tree. pub anchor: pallas::Base, /// The nullifier of input note. pub nf: Nullifier, /// The projection on the "x" coordinate of the commitment of the output note. pub cm_x: pallas::Base, /// net value commitment pub cv_net: ValueCommitment, }

As we see in the code above, the rcv

is a witness (private) and the commitment cv

is public. Also, the input and output notes are witnesses and the nullifier and commitment are public. And the authentication path of the input note is private and the root of the note commitment tree is public (to verify inclusion). This gives us an idea of the zero-knowledge checks that the action circuit does.

In short, the partial transaction also contains the verifying data for two action circuits.

pub struct ActionVerifyingInfo { action_proof: Proof, action_instance: ActionInstance, }

This completes our understanding of a partial transaction. We can now show what exactly a partial transaction is, that is, the verifying information (i.e. instances or public inputs) for the validity predicates of the applications of each input and output notes, and for the action circuit.

Input

Ouput

Action

1

vk_{Token(NAM)}

, \pi_{Token(NAM)}

, public_data

vk_{Intent}

, \pi_{Intent}

, public_data

NF(Note(5, Token(NAM), …)), Com(Note(1, Intent, …)), cv_1

, note_cm_root

2

vk_{Token(ETH)}

, \pi_{Token(ETH)}

, public_data

vk_{Token(BTC)}

, \pi_{Token(BTC)}

, public_data

NF(Note(2, Token(ETH),…)), Com(Note(1, Token(BTC), …)), cv_2

, note_cm_root

The tuple (vk

, \pi

, public_data

) constitutes the verifying information of the application validity predicate. We have omitted in the table the verifying information of the optional dynamic validity predicates.

Summary

We have created a partial transaction and understood its components. We'll look next at the role of a solver

and how a solver is able to produce a (balanced) transaction from some partial transactions and execute it. We'll also study the privacy properties of the system.