In this post, we will focus specifically on creating a token validity predicate. This will take us to understand how authorisation in Taiga works and, more generally, how dynamic validity predicates are used to extend an application validity predicate.

Token validity predicate

The only difference between different tokens, say “ETH” and “BTC”, is their unique three-letter identifier or token name

, but their application logic is likely be the same. In circuit parlance, this means that different token applications will have the same circuit structure or verifying key

. This token name must be encoded in what’s called the static data

of an application. The static data and the verifying key determine uniquely an application.

lazy_static! { pub static ref TOKEN_VK: ValidityPredicateVerifyingKey = TokenValidityPredicateCircuit::default().get_vp_vk(); pub static ref COMPRESSED_TOKEN_VK: pallas::Base = TOKEN_VK.get_compressed(); }

Validity predicates are intimately related with notes, in the sense that every note encapsulates a part of the state of the system, defined by an application. And an application is uniquely determined by the circuit structure of a validity predicate (its verifying key) and some static data (app_data_static

). Since state transitions (i.e. transactions) in Taiga are shielded by default, the input and output notes or the name of the token can’t be public data and are therefore private inputs. Implementation-wise, the private inputs of a circuit correspond to the fields of the circuit struct.

pub struct TokenValidityPredicateCircuit { pub owned_note_pub_id: pallas::Base, pub input_notes: [Note; NUM_NOTE], pub output_notes: [Note; NUM_NOTE], pub token_name: String, pub auth: TokenAuthorization, pub receiver_vp_vk: pallas::Base, }

The token name

of the token application is passed dynamically in the proving phase as a witness or private input. The four notes involved in the partial transaction also become witnesses or private inputs to each validity predicate. In particular, the token name will be taken from decoding the app_data_static

field in the note that contains this validity predicate.

Authorisation

In our token application, we want to make sure only the owners of a token note can consume it. While this is obvious for this type of application, it may not be for other types. In systems like Zexe or Zcash, each record (their equivalente of a note) has an address. These protocols enforce that the spender of the note signs it, i.e. he proves knowledge of the private key of this address for a given “signature” circuit in zero knowledge. Taiga is more expressible in the sense. As mentioned in the Zexe vs. Verizexe vs. Taiga post:

Taiga also introduces programmable authorisation, in which validity predicates control spending (instead of public keys). In short, Taiga extends the authorisation method to a validity predicate, instead of the more restrictive signature check of Zexe. The consequences of these changes are vast, enabling the surge of new applications which are not possible in the Zexe model such as subscriptions or automatic transactions in which the user doesn’t have to be online.

In particular, we’ll have another validity predicate that checks the signature. We call this validity predicate dynamic

since its private and public inputs depend on the owner of the token, and not on the token itself; they are “runtime” values provided by the spender. These dynamic validity predicates are encoded in the app_data_dynamic

field of a note and it is the role of the application validity predicate to decide what to use with it. So, how do we do this? How do we check that a party has the right (i.e. the private key) to consume a token in Taiga?

pub struct TokenAuthorization { pub pk: pallas::Point, pub vk: pallas::Base, }

As seen in the snippet above, an instance of this TokenAuthorization

struct is passed as a witness to the token validity predicate via the auth

field. It consists of the public key of the owner and the circuit structure of the authorisation validity predicate, i.e. the rules for consuming and creating the token note (vk

) and some public inputs to it (pk

).

In our custom constraints

, we assign these pk

and vk

private values (or advices) to some cells in the circuit (as mentioned in Part 1, a Halo2 circuit is commonly conceptualised as a matrix):

let pk = NonIdentityPoint::new(ecc_chip, layouter.namespace(|| "witness pk"), Value::known(self.auth.pk.to_affine()),)?;

let sender_vp_vk = assign_free_advice(layouter.namespace(|| "witness auth vp vk"), config.advices[0], Value::known(self.auth.vk),)?;

In our token application, we want the dynamic data field of the application (app_data_dynamic

) to encode the sender validity predicate (i.e. the authorisation validity predicate), the receiver validity predicate and the public key of the owner.

What is this receiver validity predicate? When receiving a token, the receiver wants the sender to encrypt the note so that the receiver can decrypt it later. That is, the receiver validity predicate validates that the given encrypted note is in fact the note encrypted with the receiver's public key.

The constraint that the app_data_dynamic

field encodes these validity predicates and not any others is part of our application logic (i.e. part of the custom_constraints

in the token validity predicate).

let app_data_dynamic = get_owned_note_variable(config.get_owned_note_variable_config, layouter.namespace(|| "get owned note app_data_dynamic"), &owned_note_pub_id, &basic_variables.get_app_data_dynamic_searchable_pairs(),)?;

let receiver_vp_vk = assign_free_advice(layouter.namespace(|| "witness receiver vp vk"), config.advices[0], Value::known(self.receiver_vp_vk),)?;

let encoded_app_data_dynamic = poseidon_hash_gadget(config.get_note_config().poseidon_config, layouter.namespace(|| "app_data_dynamic encoding"), [pk.inner().x(), pk.inner().y(), sender_vp_vk, receiver_vp_vk],)?;

layouter.assign_region(|| "check app_data_dynamic encoding", |mut region| { region.constrain_equal(encoded_app_data_dynamic.cell(), app_data_dynamic.cell()) },)?;

That is, the application validity predicate just validates in zero knowledge that the encoded dynamic data (app_data_dynamic

) of a note is a hash of the authentication and receiver validity predicates, and the public key of the owner of the token note.

When constructing a partial transaction, the dynamic application validity predicates for input and output token notes are different. If a validity predicate owns an input note, the application validity predicate requires the satisfaction of the sender_vp

, i.e. that the receiver can consume the token note by proving ownership.

let app_vp_verifying_info_dynamic = vec![Box::new(sender_vp)] InputNoteProvingInfo::new(input_note, merkle_path, Box::new(token_vp), app_vp_verifying_info_dynamic,)

If a validity predicate owns an output note, the application validity predicate requires the satisfaction of the receiver_vp

, i.e. that the sender has encrypted the note to the receiver's public key.

let app_vp_verifying_info_dynamic = vec![Box::new(receiver_vp)] OutputNoteProvingInfo::new(output_note, Box::new(token_vp), app_vp_verifying_info_dynamic,)

We use this proving information for the input and output notes to construct a partial transaction:

ShieldedPartialTransaction::build([input_note_proving_info_1, input_note_proving_info_2], [output_note_proving_info_1, output_note_proving_info_2],)

How do we do this? How does an application validity predicate require a different dynamic validity predicate depending on whether it owns an input or an output note? For this the application validity predicate commits to the relevant dynamic validity predicate and publicises it, i.e. it constrains that one of the public inputs is the right dynamic validity predicate commitment depending whether it owns an input or an output note. Since an application validity predicate knows which note it owns, it also knows whether it is an input or an output note.

```
let dynamic_cm = blake2_hash(sender_vp * is_input_note_flag + receiver_vp * (1 - is_input_note_flag), rcm);
layouter.constrain_instance(dynamic_cm.x.cell(), instances, cm_idx)?;
```

At the moment, Taiga doesn't have function privacy. Once it's implemented, the verifier circuit will check that the dynamic validity predicates in InputNoteProvingInfo

and OutputNoteProvingInfo

are the ones we previously committed.

For now, the action circuit, i.e. the circuit that encodes the rules of the Taiga protocol, checks whether this dynamic commitment corresponds to the dynamic validity predicate that a prover proves when constructing a partial transaction. The action circuit achieves this by opening the commitment.

Since this part is still under discussion, more details will come shortly