Specification: Configurable Asset Policy

TRI

Table of Contents

1 Introduction

This document defines, specifies and instantiates the notion of *Configurable Asset Policy* (CAP) system in a decentralized setting. A CAP system allows for issuance and transfers of arbitrary asset that are fully private, yet publicly verifiable.

Transactions transferring assets do not reveal the asset, amount, source or destination addresses, making each transaction private and anonymous to verifiers and observers in general.

Each asset is defined via a unique identifier code and a *policy* specifying a set of rules to be complied with when transferring the asset.

While potentially, the asset type can support arbitrary policies, the current document focuses on the following simple built-in policies designed for configurable privacy purposes:

- Asset Tracing. An asset policy may specify an entity allowed to learn source, destination, and amount of a transfer.
- Identity Tracing. An asset policy may specify the obligation of the sender to reveal credential information to an specified entity.
- Freezing/Releasing. An asset policy may specify a freezing entity allowed to *freeze* or *release* certain asset records at will without owning them.

1.1 Public ledger

The system assumes a ledger L publicly maintained where transaction are to be posted. Note that Ledger consensus and maintenance features are not in the scope of *this* document. In contrast, this write-up focuses on the information and data that user can post and observe on the ledger.

1.2 System Entities

The system involves multiple parties playing different roles.
Users: Users are both the "asset issuers" and the "customers" of the system. They can create and issue assets, as well as own and transfer assets.
Identity authorities: Identity authorities are responsible for issuing credentials to users, certifying users' identity attributes (e.g., age > 18).
Validators: Validators receive, multicast, validate transactions, and run consensus on the total ordering of the next batch of transactions to be included in the ledger.
Tracers: Whenever an asset policy allows it, *tracers* are able to extract otherwise hidden information (e.g., sender and receiver's, amounts, asset type, credentials) in published transactions.
Freezers: Freezers are able to freeze/release assets whenever there is a *freezing policy* attached to it. That is, a frozen asset cannot be spent by its owner until the freezer releases the asset.

1.3 System Workflow

At genesis, the ledger state L is initialized and the public parameters are generated. A user can join the system by generating a key pair. Users can define and *mint* assets by posting asset issuance transactions on the ledger. Users can scan the ledger to collect assets they own and can spend their assets by building and posting a transfer transaction. Validators receive transaction and verify them before posting them to the ledger. The ledger state will be updated whenever a (batch of) transaction is published. Tracers can scan the ledger to find tracing memos directed at them on each transaction. Tracers can also have *freezing* capabilities, e.i. they can publish transactions that freeze/release record whose associated asset policy allows for tracing and freezing.

In addition, users can obtain verifiable credentials from credential issuers. Credential attributes can be attached to transaction whenever a policy indicates so. In addition to common attributes such as SSN and DoB, our credentials support expiration dates.

1.4 Organization

Formal definitions of the system and the primitives used to instantiated them are presented in Section 2. In Section 3 we sketch an abstract construction based on generic cryptographic primitives and tools. Finally, Section 4 specifies instantiation details on each tool used in our abstract construction. Additionally, the appendix includes a formal description of all topics covered in this document.

2 Configurable Asset Policy: Definition

This section introduces the notion of a *Configurable Asset Policy scheme* (CAP scheme), extending the notion of a *decentralized anonymous payment* (DAP scheme) in [SCG+14]. We presents a construction of a CAP scheme in Section 3.

2.1 Objects and Data Structures

We start by specifying the data structures used in a CAP scheme.

Distributed Ledger. A *distributed ledger* whose state is denoted as L, maintained by a group of *validators* across a distributed peer-to-peer (P2P) network, keeps track of all *asset records* ever issued by aggregating them into a cryptographic *accumulator*, and all spent asset records via a *nullifier set*.
The ledger is assumed to have a *native asset* with asset type at^* and tracing policy $\mathsf{policy}^* = \perp$. The native asset is defined during ledger setup and is known to the entire network. It is primarily used for transaction fees and other incentivization payouts (e.g. block rewards) to the ledger maintainers (namely the validators).

Public Parameters. Before running the system, a trusted party generates a set of *public parameters* pp available to all users. The public parameters are used by algorithms described in Section 2.2.

Nullifier Sets. To prevent double-spending, the system maintains a *nullifier set* for the list of *spent* asset record commitments (ARCs). Each nullifier $\mathsf{nl} := \mathsf{Nullify}(\mathsf{nk}, \mathsf{elem})$ maps to an accumulated element $\mathsf{elem} := (\mathsf{uid}, \mathsf{arc})$ where uid is an unique identifier and arc is an asset record commitment. The nullifier nl reveals no information about arc to observers, and it is only computable by entities that know both arc and nk. In the current design, only arc's owner and a potential freezer have this capabilities. The nullifier set should support efficient insertion and lookup.

Addresses and Credentials. To join the system, a user generates an *address key pair* $(\mathsf{upk}, \mathsf{usk})$. The user address upk is published to allow others to transfer assets to the user. The secret key usk is used for spending assets.
We allow a user to generate more than one address key pair. In addition, users can obtain an identity credential cred from an identity authority. This credential can be used in a transaction to prove partial identity information of the sender's attributes (e.g., age > 18) without revealing further identity information.

Asset Definitions. An asset definition contains an asset type at (namely an unique identification code) and associated policies policy. The asset type is derived from a secret key known by the asset issuing entity. This derivation process avoid arbitrary users to reissue already posted asset types.

Tracing Policies. Each asset type is associated with a *tracing policy* specifying how the asset type should be traced. For example, a policy may contain a tracer public key, a credential issuer public key, and a reveal map indicating the subset of asset record information and identity attributes to be revealed to the tracer.

Freezing Policies. Each asset type may also be associated with a *freezing policy* specifying that records of this asset type may be frozen/released by a freezing entity. Our current design assumes the freezing entity has access to asset record information. Hence, freezing can only be done if the tracing policy reveals all asset record information.

Threshold Policies. Each asset type may also be associated with a *threshold policy* that specifies a threshold parameter. The tracing/freezing policies will only apply to those transactions whose transfer amount exceed the threshold. If an asset type does not enable threshold policy, the default threshold parameter is set to 0.

Asset Record Commitments. The "coin" published in the system is a commitment $\mathsf{arc} := \mathsf{Com}(v, \mathsf{at}, \mathsf{upk}, \mathsf{policy}, b_{\mathsf{frz}}; \gamma)$ to an amount v, an asset type at, a user address upk, a policy policy (that is associated with at), a freezing status flag b_{frz}, and a blinding factor γ.
We denote the commitment as *Asset Record Commitment (ARC)*, and denote the underlying values as an *ARC opening* oar.

Owner Memos. An *owner memo* is used for an asset record owner to derive the actual values, namely the opening, of an asset record commitment. It is typically an encryption (under the owner's public key) of the ARC opening. Owner memos are sent from the sender to the receiver. They do not need to be published on the ledger; however, it is convenient to do so whenever there is no private channel between the parties.

Tracer Memos.

A *tracer memo* is used for a tracer to extract otherwise hidden information about a transaction (e.g., the openings of involved ARCs and the identities of senders/receivers). It is typically an encryption (under the tracer's public key) of the transaction information.

Accumulators. The system maintains an *accumulator* Acc for the list of published ARCs. Acc has a short digest rt and supports following operations:

- $\mathsf{Acc.add}(\mathsf{rt}, \mathsf{elem})$ *inserts* an element elem and outputs an updated digest rt'. In our instantiation, $\mathsf{elem} := (\mathsf{uid}, \mathsf{arc})$ is the concatenation of an ARC arc and a unique identifier uid.
- $\mathsf{Acc.prove}(\mathsf{rt}, \mathsf{elem})$ outputs a *membership proof* π for the element elem.
- $\mathsf{Acc.vfy}(\mathsf{rt}, \pi, \mathsf{elem})$ outputs a bit *verifying* whether π is a valid proof showing that elem was correctly accumulated.
- $\mathsf{Acc.size}()$ outputs current total number of accumulated elements, which is also the uid for the next inserted ARC.

Dummy Asset Record. *Dummy* records are "spendable" asset records that hold default data (no associated policy, and zero amount value), but whose commitments are not stored in the ledger's accumulator. Nullifiers associated with dummy records are indistinguishable from real asset records nullifiers. Furthermore, each dummy record's nullifier is unique as the record holds a unique blinding factor. Users can add dummy records to hide the number of input records in a transaction.

Transactions. We specify the transaction bodies of various transaction types.

Asset Issuance. An asset issuance transaction $\mathsf{tx}_{\mathsf{mint}}$ has the form:

$$\mathsf{tx}_{\mathsf{mint}} := (\mathsf{rt}, \mathsf{nl}_{\mathsf{fee}}, \mathbf{arc}_{\mathsf{out}}, (v, \mathsf{at}, \mathsf{policy}), \mathsf{aux})$$

where rt is an accumulator state, $\mathsf{nl}_{\mathsf{fee}}$ is the nullifier for input fee, $\mathbf{arc}_{\mathbf{out}}$ is a vector of two output ARC, the first of which is the fee change and the second is the minted new asset record, $(v, \mathsf{at}, \mathsf{policy})$ is the exact amount, asset type and policy of the newly minted record, and aux is auxiliary information. In our instantiation, aux contains a proof that the commitment is *honestly* computed, a tracing memo for the minted output commitments, and also transaction fee information. When issuing assets, an honestly computed commitment means the sender knows a secret seed used to derive the asset type code.

Anonymous Transfer. An anonymous transfer $\mathsf{tx}_{\mathsf{axfr}}^{n,m}$ (parameterized by n, m — the number of inputs and outputs) has the form

$$\mathsf{tx}_{\mathsf{axfr}} := (\mathsf{rt}, \mathbf{nl}_{\mathsf{in}}, \mathbf{arc}_{\mathsf{out}}, \mathsf{aux}) \ ,$$

where rt is an accumulator state, $\mathbf{nl}_{\mathsf{in}} = (\mathsf{nl}_i)_{i \in [n]}$ is a list of input nullifiers, $\mathbf{arc}_{\mathsf{out}} = (\mathsf{arc}_i)_{i \in [m]}$ is a list of output ARCs, and aux is auxiliary information. In our instantiation, aux includes a tracer memo, a proof that the transaction is valid and has not been modified, and transaction fee info. In the following, we use the notation $\mathsf{tx}_{\mathsf{axfr}}$ for $\mathsf{tx}_{\mathsf{axfr}}^{n,m}$ when n, m are clear in the context.

Asset freeze. A freezing transaction $\mathsf{tx}^n_{\mathsf{frz}}$ (parameterized by n — the number of inputs) has the form

$$\mathsf{tx}_{\mathsf{frz}} := (\mathsf{rt}, \mathbf{nl}_{\mathsf{in}}, \mathbf{arc}_{\mathsf{out}}, \mathsf{aux}) \ ,$$

where rt is an accumulator state, $\mathbf{nl}_{\mathsf{in}} = (\mathsf{nl}_i)_{i\in[n]}$ is a list of input nullifiers, $\mathbf{arc}_{\mathsf{out}} = (\mathsf{arc}_i)_{i\in[n]}$ is a list of *frozen* output asset record commitments, and aux is auxiliary information. In our instantiation, aux includes transaction fee info, and a proof for the following statements: the input records contain a freezing policy and that their freeze-flag is FREEZABLE, and that the output freeze-flag is FROZEN, and that can be unfrozen only to their original data.

The same transaction is used to unfreeze frozen records by swapping the freeze-flag states in the statement above.

A Note On Transaction Non-malleability. Typically, to prevent adversaries from mauling the transaction content, a transaction also contains a *signature* over the transaction body, and a *proof* showing that the corresponding signature verification key is honestly generated from the sender. Fortunately, we can remove the use of signatures because the SNARK proof system (i.e., Plonk) we are using is conjectured to be non-malleable in the first place (more formally, it satisfies a property called *simulation extractability*), hence the adversary cannot create a valid SNARK proof after modifying any part of the transaction. We stress, however, that we need to add the signature-key binding mechanism as in ZCash [SCG+14] when we replace Plonk with a SNARK scheme that does not satisfy simulation extractability (e.g., [Gro16]).

2.2 Scheme Syntax and System Workflow

Definition 1. *A **Configurable Asset Policy** (CAP) scheme is a tuple of polynomial-time algorithms*

$$\begin{aligned}\Pi = (&\mathsf{Setup}, \mathsf{CredKeyGen}, \mathsf{UserKeyGen}, \mathsf{TracerKeyGen}, \mathsf{FreezerKeyGen},\\ &\mathsf{RequestCred}, \mathsf{VerifyCredReq}, \mathsf{IssueCred},\\ &\mathsf{IssueAsset}, \mathsf{AnonXfr}, \mathsf{Freeze}, \mathsf{Unfreeze}, \mathsf{VfyTx}, \mathsf{Receive}, \mathsf{Trace})\end{aligned}$$

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$*: Given a security parameter λ, the algorithm outputs a list of public parameters* pp.
- $\mathsf{CredKeyGen}(\mathsf{pp}) \to (\mathsf{isk}, \mathsf{ipk})$*: Given public parameters* pp*, the algorithm outputs a credential issuing key pair* $(\mathsf{isk}, \mathsf{ipk})$.
- $\mathsf{UserKeyGen}(\mathsf{pp}) \to (\mathsf{usk}, \mathsf{upk})$*: Given public parameters* pp*, the algorithm outputs a user key pair* $(\mathsf{usk}, \mathsf{upk})$.
- $\mathsf{TracerKeyGen}(\mathsf{pp}) \to (\mathsf{tsk}, \mathsf{tpk})$*: Given public parameters* pp*, the algorithm outputs a tracer key pair* $(\mathsf{tsk}, \mathsf{tpk})$.
- $\mathsf{FreezerKeyGen}(\mathsf{pp}) \to (\mathsf{fsk}, \mathsf{fpk})$*: Given public parameters* pp*, the algorithm outputs a freezer key pair* $(\mathsf{fsk}, \mathsf{fpk})$.
- $\mathsf{RequestCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{usk}, \mathsf{upk}, \mathsf{msg}) \to \mathsf{request}$*: Given public parameters* pp*, a credential issuer public key* ipk*, a user key pair* $(\mathsf{usk}, \mathsf{upk})$*, and a message* msg *to the credential issuer, output a request of credentials to be issued by* ipk *for* upk.
- $\mathsf{VerifyCredReq}(\mathsf{pp}, \mathsf{upk}, \mathsf{request}) \to b$*: Given public parameters* pp*, a user public pair* upk*, and the user request, return accept/reject bit* b.
- $\mathsf{IssueCred}(\mathsf{pp}, \mathsf{isk}, \mathsf{upk}, \mathbf{attrs}, \mathsf{aux}) \to \mathsf{cred}$*: Given public parameters* pp*, a credential issuer secret key* isk*, a user public key* upk*, a list of identity attributes* $\mathbf{attrs}$ *of the user, and auxiliary data, output a credential* cred *signed under* isk.
- $\mathsf{IssueAsset}(\mathsf{pp}, \mathbf{oar}_{\mathsf{out}}, \mathsf{aux}_{\mathsf{at}}, \mathsf{aux}_{\mathsf{fee}}, \mathsf{rt}) \to (\mathsf{tx}_{\mathsf{mint}}, \mathsf{memo}_{\mathsf{own}}, \sigma_{\mathsf{memo}})$*: Given public parameters* pp*, two output ARC openings* $\mathbf{oar}_{\mathsf{out}}$ *(the first is fee change, the second is the new asset record minted), auxiliary information about minted asset type* $\mathsf{aux}_{\mathsf{at}}$*, information about input ARC (namely fee)* $\mathsf{aux}_{\mathsf{fee}}$*, and current accumulator root* rt*, the algorithm outputs an asset issuance transaction* $\mathsf{tx}_{\mathsf{mint}}$*, an owner memo for the new asset record* $\mathsf{memo}_{\mathsf{own}}$ *and a signature over the owner memo* σ_{memo} *under a fresh random public key of the sender.*
- $\mathsf{AnonXfr}_{n,m}(\mathsf{pp}, \mathbf{usk}_{\mathsf{in}}, \mathbf{arc}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{in}}, \boldsymbol{\pi}_{\mathsf{in}}, \mathsf{aux}) \to (\mathsf{tx}_{\mathsf{axfr}}, \mathbf{memo}_{\mathbf{own}}, \sigma_{\mathsf{memo}})$*: This algorithm transfers asset values from input ARCs to new output ARCs, marking the input ARCs as spent. More precisely, denote by n, m the number of input and output ARCs. Given public parameters* pp*, a list of sender secret keys* $\mathbf{usk}_{\mathsf{in}}$*, a list of input ARCs* $\mathbf{arc}_{\mathsf{in}}$ *and their openings* $\mathbf{oar}_{\mathsf{in}}$*, a list of membership proofs* $\boldsymbol{\pi}_{\mathsf{in}}$*, and auxiliary information* aux*, the algorithm outputs an anonymous transfer transaction* $\mathsf{tx}_{\mathsf{axfr}}$*, a list of owner memos for output records*[1] $\mathbf{memo}_{\mathbf{own}}$ *and a signature over all owner memos* σ_{memo} *under a fresh random public key of the sender.*

[1] except the 1st output as it is fee change

- $\mathsf{Freeze}_n(\mathsf{pp}, \mathbf{fsk}_{\mathsf{in}}, \mathbf{arc}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{in}}, \boldsymbol{\pi}_{\mathsf{in}}, \mathsf{aux}) \to (\mathsf{tx}_{\mathsf{frz}}, \mathbf{memo}_{\mathbf{own}}, \sigma_{\mathsf{memo}})$*: This algorithm freezes n input asset records by consuming them and minting new records of the same opening except the freezing status flag $b_{\mathsf{frz}} = 1$ (i.e. being flipped to "frozen" state). Given public parameters* pp*, a list of freezer secret keys* $\mathbf{fsk}_{\mathsf{in}}$*, a list of input ARCs* $\mathbf{arc}_{\mathsf{in}}$ *and their openings* $\mathbf{oar}_{\mathsf{in}}$*, a list of membership proofs* $\boldsymbol{\pi}_{\mathsf{in}}$*, and auxiliary information* aux*, the algorithm outputs an anonymous transfer transaction* $\mathsf{tx}_{\mathsf{frz}}$*, a list of owner memos for output records* $\mathbf{memo}_{\mathbf{own}}$ *and a signature over all owner memos* σ_{memo} *under a fresh random public key of the freezer.*
- $\mathsf{Unfreeze}_n(\mathsf{pp}, \mathbf{fsk}_{\mathsf{in}}, \mathbf{arc}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{in}}, \boldsymbol{\pi}_{\mathsf{in}}, \mathsf{aux}) \to (\mathsf{tx}_{\mathsf{frz}}, \mathbf{memo}_{\mathbf{own}}, \sigma_{\mathsf{memo}})$*: This algorithm unfreezes n input asset records and shares identical syntax and interface with* Freeze*. The only difference is that all input records need to have $b_{\mathsf{frz}} = 1$ and all output records need to have those flags $b_{\mathsf{frz}} = 0$ (i.e. flipped back to "unfrozen"). Note that the output transaction* $\mathsf{tx}_{\mathsf{frz}}$ *is of the same transaction type as that of* Freeze *and it's infeasible to distinguish transactions that freezes records from ones that unfreeze records.*
- $\mathsf{VfyTx}(\mathsf{pp}, \mathsf{tx}, \mathsf{info}_L) \to b$*: Given public parameters* pp*, a transaction* tx*, and information about the ledger state* info_L*, the algorithm outputs a bit $b \in \{0, 1\}$ indicating whether the transaction* tx *is valid. In our instantiation,* info_L *consists of the current accumulator digest and block timestamp.*
- $\mathsf{Receive}(\mathsf{pp}, \mathsf{upk}, \mathsf{usk}, L) \to ((\mathsf{arc}_1, \mathsf{oar}_1), \ldots, (\mathsf{arc}_\ell, \mathsf{oar}_\ell))$*: Given public parameters* pp*, a user key pair* $(\mathsf{usk}, \mathsf{upk})$*, and the current ledger state L, the algorithm outputs the list of unspent ARCs (and the corresponding openings) received by* upk*.*
- $\mathsf{Trace}(\mathsf{pp}, \mathsf{tpk}, \mathsf{tsk}, \mathsf{tx}) \to \mathsf{list}_{\mathsf{trc}}$*: Given public parameters* pp*, a tracer key pair* $(\mathsf{tsk}, \mathsf{tpk})$*, and a transfer transaction* $\mathsf{tx}_{\mathsf{axfr}}$*, the algorithm outputs a list* $\mathsf{list}_{\mathsf{trc}}$ *— the information to be traced underlying* tx*.*

Remark 1. Tracer memos contain encryptions of the openings of both input and output ARCs, together with selectively revealed identity attributes of the sender. The correctness of a tracer memo is enforced via the SNARK circuit, and it is part of the anonymous transfer transaction body – namely part of $\mathsf{tx}_{\mathsf{axfr}}$. Whereas *owner memos* contain encryptions of the opening of only the output ARCs. Being separated from the transaction body, owner memos are not verified by network validators – in practice, they are most likely not even stored on-chain, but either being sent directly to the receiver via authenticated secure channel or being sent to a public bulletin board waiting to be pulled asynchronously by the receiver later. To prevent anyone from sending garbage to the public bulletin board, we further require a signature over all owner memos in a transaction from the sender so that the bulletin board service won't be flooded with unauthenticated memos. Moreover, since the signature over owner memos is a publicly viewable value, it has to be dissociated from the real identity of the sender to avoid linking of transactions by the same sender. To achieve that, we allow the sender to specify a freshly generated public key in the public inputs of the transaction SNARK circuit to bind the public key with that particular transaction and consequently sign the list of owner memos under this random key pair. Noted that the fresh public key is optional (can be a dummy key if the sender choose to deliver the owner memo via authenticated channel instead), and it is independent of the circuit logic – meaning it is not checked inside the logic, thus incurring no additional cost in proving or in verification or in the size of the proof.

System Workflow: At genesis, the ledger state L is initialized and the Setup algorithm is executed to generate the system's public parameters. A user can join the system by invoking $\mathsf{UserKeyGen}$ to generate a user key pair $(\mathsf{usk}, \mathsf{upk})$, A user (asset issuer) can define and issue an asset by invoking the $\mathsf{IssueAsset}$ algorithm and broadcasting the asset issuance transaction. A user can collect the assets she owned by invoking the $\mathsf{Receive}$ algorithm, then she can spend her assets by invoking the $\mathsf{AnonXfr}$ algorithm and broadcasting the anonymous transfer transaction. Upon receiving a transaction, a validator invokes the VfyTx algorithm to ensure that the transaction is valid and has not been mauled. The ledger state will be updated whenever a transaction is published. For each transfer transaction $\mathsf{tx}_{\mathsf{axfr}}$ published on the ledger, a tracer T is able to extract the hidden information of tx if the asset type underlying tx is associated with a policy containing the tracer public key of T.

Additionally, the $\mathsf{CredKeyGen}$ algorithms are used to setup credential issuers that provide credential to users. Users can request a credential from a provider by creating a credential request using $\mathsf{RequestCred}$ algorithm. Credential issuers can verify the request by calling $\mathsf{VerifyCredReq}$, and issue a credential on a set of attributes attrs by executing $\mathsf{IssueCred}$. When transferring assets with tracing policies, users use the credential to selectively reveal traced attributes.

2.3 Completeness and Security

We define completness and the security properties of an Configurable Asset Policy scheme in Appendix B.

3 Configurable Asset Policy: Construction

We now construct a concrete CAP scheme using building blocks including cryptographic primitives listed in Section 3.1 and zk-SNARK proof system in Section 3.2.

Our construction shown in Section 3.3 in pseudocode describes the internal of all algorithms whose syntax are defined in Section 2.2. It simultaneously serves as a formal API reference of the overall payment system.

3.1 Cryptographic Building Blocks

Hash Functions: Let $\mathcal{S}$, $\mathcal{M}$ and $\mathcal{T}$ be sets such that $\mathcal{S}$ is a seed space of exponential size in the security parameter λ, $\mathcal{M}$ is the message space, $\mathcal{T}$ is the digest space, and that $|\mathcal{M}| > |\mathcal{T}|$. A hash function $\mathcal{H} = (\mathsf{Gen}_{\mathsf{hash}}, \mathsf{H})$ is defined by the following efficiently computable algorithms:

- $s \leftarrow \mathsf{Gen}_{\mathsf{hash}}(1^\lambda)$: Instance generation algorithm that samples a random instance $s \in \mathcal{S}$.
- $h \leftarrow \mathsf{H}^{(s)}(m)$: Hashing algorithms that take an instance description s and a message $m \in \mathcal{M}$ and output a digest $h \in \mathcal{T}$.

The hash function satisfies the **collision resistant** property (formally defined in Appendix A Definition 2).

Pseudo-random Function and Permutation: A **pseudo-random function (PRF)** is an efficiently computable function F_k from a PRF family $\{F_k : \mathcal{X} \to \mathcal{Y}\}_{k \in \mathcal{K}}$ where $\mathcal{K}$ is the key space, $\mathcal{X}$ is the input space, $\mathcal{Y}$ is the output space. Informally, it satisfies *pseudo-randomness* if it is indistinguishable from a function uniformly sampled from the set of all functions with the same input and output spaces(formal definition see Definition 3).

A **pseudo-random permutation (PRP)** is defined similarly to PRF only with its input space equal to its output space($\mathcal{X} = \mathcal{Y}$).

Signature Schemes: A **signature scheme** $\mathcal{S} = (\mathsf{Gen}_{\mathsf{sig}}, \mathsf{Sign}, \mathsf{Vfy})$ is a triple of efficient algorithms, Specifically,

- $(pk_{\mathsf{sig}}, sk_{\mathsf{sig}}) \xleftarrow{\$} \mathsf{Gen}_{\mathsf{sig}}(1^\lambda)$ is a probabilistic **key generation algorithm** that outputs a key pair consists of a **verification key** pk_{sig} and a **signing key** sk_{sig}.
- $\sigma \leftarrow \mathsf{Sign}(sk_{\mathsf{sig}}, m)$ is a **signing algorithm** that given a signing key sk_{sig} and a message from a message space$m \in \mathcal{M}$, outputs a **signature** denoted as σ.
- $b \leftarrow \mathsf{Vfy}(pk_{\mathsf{sig}}, m, \sigma)$ is a **verification algorithm** that given the verification key, the original message and the proclaimed signature, outputs an accepting bit $b \in \{0, 1\}$ representing accept if $b = 1$, and reject otherwise.

We require the digital signature scheme to have *existential unforgeability against chosen message attack* $\mathsf{UF\text{-}CMA}$ **security**, which captures the notion that it should be computationally infeasible for an efficient adversary $\mathcal{A}$ to forge a signature of a new message under a given verification key, even if the adversary were given many signatures over messages of its choices (formal definition at Definition 6).

Public-key Encryption Schemes: A **public-key encryption scheme** $\mathcal{E} = (\mathsf{Gen}_{\mathsf{enc}}, \mathsf{Enc}, \mathsf{Dec})$ is a triple of efficient algorithms, Specifically,

- $(pk_{\mathsf{enc}}, sk_{\mathsf{enc}}) \xleftarrow{\$} \mathsf{Gen}_{\mathsf{enc}}(1^\lambda)$ is a probabilistic **key generation algorithm** that outputs a key pair consists of a **public key** pk_{enc} and a **private key** sk_{enc}.
  The public key defines a message space $\mathcal{M}_{pk}$.
- $\mathsf{ct} \xleftarrow{\$} \mathsf{Enc}(pk_{\mathsf{enc}}, m)$ is a probabilistic **encryption algorithm** that given a public key and a message $m \in \mathcal{M}_{pk}$, outputs a ciphertext ct.
- $m \leftarrow \mathsf{Dec}(sk_{\mathsf{enc}}, \mathsf{ct})$ is a deterministic **decryption algorithm** that given a ciphertext and the secret key whose corresponding public key were used to generate the ciphertext, outputs the encrypted message in plaintext. m is either a message or a special reject value if decryption failed.
- For all possible key pairs from $\mathsf{Gen}_{\mathsf{enc}}$, and all messages m, we have

$$\Pr[\mathsf{Dec}(sk_{\mathsf{enc}}, \mathsf{Enc}(pk_{\mathsf{enc}}, m)) = m] = 1$$

There are two security properties we require from the encryption scheme for our construction, informally defined as follows:

- IND-CPA security: for all polynomial-time adversaries, the ciphertexts for any two lists of adversary-chosen plaintext are indistinguishable from another, provided that the length of the list is poly-bounded. (see formal definition at Definition 7)
- IK-CCA security [BBDP01]: given a challenging ciphertext, all polynomial-time adversaries could not determine which particular key from a group of known public keys is used to produces the ciphertext. Similarly, the adversaries are given a decryption oracle to query the plaintext of any non-challenging ciphertext. This property ensures the anonymity of the keys being used during encryption, thus protecting the identity of the encrypting party. (see formal definition at Definition 8)

Credentials: A credential is a signature from an identity authority over a set of arbitrary attributes including the credential owner public key.

- $(\mathsf{isk}, \mathsf{ipk}) \leftarrow \mathsf{Gen}_{\mathsf{cred\_issuer}}(1^\lambda)$: is the key generation algorithm for credential issuer.
- $\mathsf{request} \leftarrow \mathsf{RequestCred}(\mathsf{ipk}, \mathsf{upk}, \mathsf{usk}, \mathsf{msg})$: Computes a credential request for user upk.
- $0/1 \leftarrow \mathsf{VerifyCredReq}(\mathsf{request}, \mathsf{isk}, \mathsf{upk}, \mathsf{msg})$: Verifies a credential request.
- $\mathsf{cred} \leftarrow \mathsf{Issue}_{\mathsf{cred}}(\mathsf{isk}, \mathsf{upk}, \mathsf{attrs})$: is a credential issuance algorithm invoked by an issuer, on input the issuer's secret key isk, a user public key, and attributes attrs, outputs a credential cred.
- $b \leftarrow \mathsf{Vfy}_{\mathsf{cred}}(\mathsf{ipk}, \mathsf{cred}, \mathsf{attrs})$: is a public credential verification algorithm, on input the issuer's public key ipk, a credential cred and the attributes attrs outputs an accepting bit $b \in \{0, 1\}$, $b = 1$ indicating valid credential.
- $\pi \leftarrow \mathsf{Reveal}_{\mathsf{cred}}(\mathsf{usk}, \mathsf{cred}, \mathsf{attrs}, \mathsf{bitmap}, \mathsf{ipk})$: is a selective reveal randomized algorithm invoked by the user. On input the originally issued credential cred and corresponding opening of the committed attributes attrs, a bitmap bitmap indicating which subset of the attributes should be revealed, the issuer's public key ipk and outputs a zero-knowledge proof of knowledge π of a valid signature (i.e the original credential).
- $b \leftarrow \mathsf{VfyAttr}_{\mathsf{cred}}(\mathsf{cred}, \pi, \mathsf{attrs}', \mathsf{bitmap}, \mathsf{ipk})$: is the verification algorithm for credential cred'. On input the credential cred, the proof π of the validity of the credential given the revealed attributes $\mathsf{attrs}' := \mathsf{attrs} \odot \mathsf{bitmap}$ for some bit vector bitmap, it will output an accepting bit $b \in \{0, 1\}$, indicating accept if $b = 1$.

In our system, this simple credentials issuance and verification can be constructed via a regular signature scheme. Revealing and verifying attributes can be implemented with zero-knwoledge proof systems.

Commitment Schemes: A *commitment scheme* allows one party to "commit" to a secret value m by publishing a *commitment* cm, and then to provably reveal m (called "opening" the commitment).

More precisely, a **commitment scheme** $\mathcal{C} = (\mathsf{Setup}_{\mathsf{com}}, \mathsf{Com}, \mathsf{Vfy}_{\mathsf{com}})$ is a triple of efficient algorithms where:

- $\mathsf{pp} \xleftarrow{\$} \mathsf{Setup}_{\mathsf{com}}(1^\lambda)$ is a setup algorithm that outputs a public parameter given the security parameter;
- $\mathsf{cm} \leftarrow \mathsf{Com}(\mathsf{pp}, m; r)$ is a commit algorithm that produces a commitment cm given the message from a message space to be committed ($m \in \mathcal{M}_{\mathsf{pp}}$), and an explicit randomness $r \xleftarrow{\$} \mathcal{R}_{\mathsf{pp}}$ from some random space;
- $b \leftarrow \mathsf{Vfy}_{\mathsf{com}}(\mathsf{pp}, \mathsf{cm}, m, r)$ is a verification algorithm that checks whether (m, r) is the correct opening of the commitment cm, and outputs a bit $b \in \{0, 1\}$ representing accept if $b = 1$, and reject otherwise.

Noted that some definitions in the literature make the commit algorithm Com probabilistic and instead of taking an explicit randomness r, they return some "opening hint" (possibly r used, sometimes with more information) – both are valid definitions.

Informally, a commitment scheme is called **binding** if once a message is committed, it is infeasible to later open to a different message; and it is called **hiding** if the commitments of any two messages are indistinguishable from one another. See formal definition in Appendix at Definition 9 and Definition 10.

Polynomial Commitment Scheme (PCS): Now we proceeds to define a particular commitment scheme of interest called *polynomial commitment scheme* (PCS), which was first introduced by [KZG10]. Informally, a polynomial commitment scheme allows a prover to commit to a univariate polynomial $p \in \mathbb{F}[X]$ by producing a commitment c, and then later on "open" c at any point $x \in \mathbb{F}$, producing an *evaluation proof* π, testifying that the opened value is consistent with the polynomial committed evaluated at value x – i.e. $p(x)$.

More formally, a **polynomial commitment scheme** $PCS = (\mathsf{Setup}_{\mathsf{pc}}, \mathsf{Com}_{\mathsf{pc}}, \mathsf{Vfy}_{\mathsf{pc}}, \mathcal{P}_{\mathsf{Eval}}, \mathcal{V}_{\mathsf{Eval}})$ is a tuple of five efficient algorithms that run in time polynomial in λ, d. [2] Specifically,

- $\mathsf{pp} \xleftarrow{\$} \mathsf{Setup}_{\mathsf{pc}}(1^\lambda, d)$: is a probabilistic setup algorithm that outputs public parameter pp for committing to polynomial of maximum degree d.
- $(\mathsf{cm}_{\mathsf{pc}}, o_{\mathsf{pc}}) \leftarrow \mathsf{Com}_{\mathsf{pc}}(\mathsf{pp}, f)$: outputs a commitment $\mathsf{cm}_{\mathsf{pc}} \in \mathcal{C}$ (a commitment space $\mathcal{C}$) to the polynomial $f \in \mathbb{F}^{\leq d}[X]$ and an opening "hint" $o_{\mathsf{pc}} \in \{0,1\}^*$.
- $b \leftarrow \mathsf{Vfy}_{\mathsf{pc}}(\mathsf{pp}, f, o_{\mathsf{pc}}, \mathsf{cm}_{\mathsf{pc}})$: checks the validity of an opening hint o_{pc} for a commitment $\mathsf{cm}_{\mathsf{pc}}$ to a polynomial f and outputs $b \in \{0,1\}$ where $b = 1$ represents accept, $b = 0$ represents reject.
- $\pi \leftarrow \mathcal{P}_{\mathsf{Eval}}(\mathsf{pp}, f, o_{\mathsf{pc}}, \mathsf{cm}_{\mathsf{pc}}, z, y)$: outputs a proof π which prove non-interactively that $f(z) = y$ and $\deg(f) < d$. This algorithm will be run by the prover.
- $b \leftarrow \mathcal{V}_{\mathsf{Eval}}(\mathsf{pp}, \pi, \mathsf{cm}_{\mathsf{pc}}, z, y)$: verifies the proof π of an proclaimed evaluation value $y \in \mathcal{C}$ of a committed polynomial f at point $z \in \mathbb{F}$ outputs an accepting bit $b \in \{0,1\}$. This algorithm will be run by the verifier.

A PCS needs to satisfy security properties including **correctness**, which informally states that any honestly computed polynomial evaluation at any point and its respective proof would pass verification; **binding** which is analogous to that of a standard commitment scheme; **knowledge soundness** which informally states that anyone who can produce a convincing proof during non-interactive point evaluation must *know* the original polynomial. Formal definitions of the foregoing properties are specified in Definition 11, 12 and 13.

Moreover, a PCS is called **efficient** if the $\mathcal{V}_{\mathsf{Eval}}$ verifier runs in time $o(d \cdot log|\mathbb{F}|)$, i.e. sublinear in the size of committed polynomial. The scheme is called **succinct** if both the size of commitments and proof size of $\mathcal{P}_{\mathsf{Eval}}$ protocol are $o(d \cdot log|\mathbb{F}|)$.

Universal SNARK: A **pre-processing zk-SNARK with universal SRS** is a tuple of efficient algorithms $(\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ where:

- $\mathsf{srs} \xleftarrow{\$} \mathcal{G}(1^\lambda, N)$: is a probabilistic, polynomial-time generator that samples an SRS srs that supports indices of size up to N. This is the universal setup phase to be carried out once and used across all future circuits of size up to N.
- $(\sigma, \tau) \leftarrow \mathcal{I}^{\mathsf{srs}}(i)$: is a deterministic, polynomial-time indexing algorithm that on input an index i of size at most N and with oracle access to the SRS, outputs the proving key σ and verification key τ. This is the pre-processing phase for each circuit where i is description of the circuit, and the computed keys can be used for proving and verifying membership of all instance-witness pairs of the same relation.
- $\pi \xleftarrow{\$} \mathcal{P}(\sigma, x, w)$: is a probabilistic, polynomial-time proving algorithm that given the proving key σ, the public instance x, the secret witness w, produces a proof π.
- $b \leftarrow \mathcal{V}(\tau, x, \pi)$: is a deterministic, polynomial-time verification algorithm that given the verification key τ, the public instance x and a proof π, outputs an accepting bit $b \in \{0,1\}$ indicating accept if $b = 0$ or reject otherwise.

For formal definitions, please see Definition 14.

3.2 zk-SNARKs for Transactions Validity

3.2.1 Building Blocks: Syntax We first recall the syntax of asset record commitments and defer the the concrete specification of asset record commitments (i.e., the instantiation of the commitment scheme and the concrete definitions of the data fields) to Section 4.2.5.

[2] While there are numerous definitions for a PCS, we use the one proposed in [BDFG20b] Section 2.2, please refer to the paper for more details.

Asset Record Commitments. An *asset record commitment* (ARC) $\mathsf{arc} := \mathsf{Com}(v, \mathsf{at}, \mathsf{upk}, \mathsf{policy}, b_{\mathsf{frz}}; \gamma)$ is a commitment (with blinding factor γ) to an amount v, an asset type at, an user address upk, an asset policy policy, and a b_{frz} indicating whether the record is spendable or frozen. The asset policy $\mathsf{policy} := (\mathsf{threshold}, \mathsf{tpk}, \mathsf{ipk}, \mathsf{fpk}, \mathsf{reveal})$ consists of an auditing threshold $\mathsf{threshold}$, a tracer public key tpk, a credential issuer public key ipk, a freezer public key, and a reveal map reveal indicating whether to reveal at, v, upk, and each identity attribute respectively.

Building Block Algorithms. The circuits for transactions $\mathsf{tx}_{\mathsf{mint}}$, $\mathsf{tx}_{\mathsf{axfr}}$ and $\mathsf{tx}_{\mathsf{frz}}$ make use of sub-circuits for a few building block algorithms. We explain below the syntax of the building block algorithms and defer the concrete instantiations of the sub-circuits (as well as the data types of inputs and outputs) to Section 4.2.5.

- $(b_0, \ldots, b_{\ell-1}) \leftarrow \mathsf{range}(v, \ell)$ denotes a range-check algorithm. If v is in the range $[0, 2^\ell)$, it outputs the binary representation of v in the least significant bit form; otherwise it outputs $\perp$.
- $\mathsf{at} \leftarrow \mathsf{derive}_{\mathsf{at}}(s_{\mathsf{at}}, \mathsf{aux})$ denotes an algorithm for deriving asset type at from secret seed s_{at} and auxiliary information aux (e.g. asset description digest).
- $\mathsf{pk} \leftarrow \mathsf{derive}^B_{\mathsf{pk}}(\mathsf{sk})$ denotes an algorithm for computing a user/freezer public key pk from a user/freezer secret key sk. Here B is a hardcoded base point, and we omit it when clear in the context.
- $\mathsf{nk} \leftarrow \mathsf{derive}_{\mathsf{nk}}(\mathsf{sk}_1, \mathsf{pk}_2)$ denotes an algorithm to derive the nullifier key from a secret key sk_1 and a public key pk_2.
- $\mathsf{nl} \leftarrow \mathsf{Nullify}_{\mathsf{nk}}(\boldsymbol{x})$ denotes an algorithm for computing a nullifier nl from a derived nullifying key nk and an input $\boldsymbol{x}$. The size of $\boldsymbol{x}$ is fixed in the scheme (i.e., we do not support variable-length input).
- $y \leftarrow \mathsf{Com}(\boldsymbol{x}; \gamma)$ denotes an algorithm for computing a commitment y from a blinding factor γ and an input $\boldsymbol{x}$. The size of $\boldsymbol{x}$ is fixed.
- $b \leftarrow \mathsf{Acc.vfy}(\mathsf{rt}, \pi, \mathsf{elem})$ denotes a Merkle membership proof verification algorithm. Here rt is a Merkle root value, π is an inclusion proof, and $(\mathsf{uid}, \mathsf{arc})$ is an accumulated element where uid is a unique identifier and arc is an asset record commitment.
- $b \leftarrow \mathsf{PoB}_B(\mathsf{rt}, \mathsf{nl}, \pi, \mathsf{uid}, \mathsf{oar}, \mathsf{sk}_1, \mathsf{uorf})$ denotes a proof-of-burn algorithm that checks the proof-of-knowledge of a spendable asset record, where $\mathsf{elem} := (\mathsf{uid}, \mathsf{arc})$ is the accumulated element (if record is not dummy). Here B is a hardcoded base point and we omit it when clear in the context. The algorithm has two modes: when in the user mode (i.e. $\mathsf{uorf} = 1$), we set two public keys $\mathsf{pk}_1 := \mathsf{upk}$ and $\mathsf{pk}_2 := \mathsf{fpk}$. Here upk is the user address (included in oar) and fpk is the freezer public key (included in the policy inside oar); when in the freezer mode (i.e. $\mathsf{uorf} = 0$), we set $\mathsf{pk}_1 := \mathsf{fpk}$ and $\mathsf{pk}_2 := \mathsf{upk}$. The algorithm checks that
  1. $\mathsf{pk}_1 = \mathsf{derive}^B_{\mathsf{pk}}(\mathsf{sk}_1)$. Proof-of-knowledge of secret key.
  2. $\mathsf{nk} \leftarrow \mathsf{derive}_{\mathsf{nk}}(\mathsf{sk}_1, \mathsf{pk}_2)$ (i.e., derive nullifying key).
  3. $\mathsf{nl} = \mathsf{Nullify}_{\mathsf{nk}}(\mathsf{elem})$ (i.e., nullifier is correctly computed).
  4. $\mathsf{arc} \leftarrow \mathsf{Com}(\mathsf{oar})$ (i.e., derive asset record commitment).
  5. $\mathsf{oar.at} = \mathsf{at}_{\mathsf{dummy}} \vee \mathsf{Acc.vfy}(\mathsf{rt}, \pi, (\mathsf{uid}, \mathsf{arc})) = 1$ (i.e., ARC was accumulated if non-dummy).
- $\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{pk}}(\boldsymbol{m}; s)$ denotes an El-Gamal hybrid encryption algorithm that computes ciphertext ct from public key pk, message vector $\boldsymbol{m}$, and encryption randomness s. The message length $|\boldsymbol{m}|$ is fixed.
- $b \leftarrow \mathsf{Vfy}_{\mathsf{ac}}(\mathsf{ipk}, \mathsf{cred}, \boldsymbol{m})$ denotes an algorithm for verifying a credential, where ipk is a credential issuer public key, cred is a credential, and $\boldsymbol{m}$ is a message vector to be certified. The message size $|\boldsymbol{m}|$ is fixed.
- $b \leftarrow \mathsf{Balance}(\mathsf{at}^*, \mathsf{at}, \mathsf{fee}, (v_i^{\mathsf{in}})_{i\in[n]}, (v_i^{\mathsf{out}})_{i\in[m]})$ denotes a balance preserving algorithm that checks the equations $v_1^{\mathsf{in}} = v_1^{\mathsf{out}} + \mathsf{fee}$ and
$$\sum_{i=2}^{n} v_i^{\mathsf{in}} = \sum_{i=2}^{m} v_i^{\mathsf{out}}$$
when $\mathsf{at}^* \neq \mathsf{at}$, and
$$\sum_{i=1}^{n} v_i^{\mathsf{in}} = \sum_{i=1}^{m} v_i^{\mathsf{out}} + \mathsf{fee}$$
when $\mathsf{at}^* = \mathsf{at}$.

Next we focus on describing the SNARK statements for $\mathsf{tx}_{\mathsf{mint}}$, $\mathsf{tx}_{\mathsf{axfr}}$ and $\mathsf{tx}_{\mathsf{frz}}$ transactions. By composing the instantiations of the sub-circuits, we obtain the circuits for transactions.

3.2.2 Asset Issuance After defining an asset with asset type at and asset policy policy, an asset issuer can issue/mint asset records for other users. Given an amount v and a recipient public address upk, the asset issuer creates an asset-record commitment $\mathsf{arc} := \mathsf{Com}(v, \mathsf{at}, \mathsf{upk}, \mathsf{policy}, b_{\mathsf{frz}} = 0; \gamma)$ and proves the following:

- The asset issuer knows the secret seed of the asset type. This prevents others from forging assets on behalf of the asset issuer.
- To counteract denial-of-service attacks and replay attacks, the asset issuer is also required to burn some amounts of native tokens as transaction fees (by proving knowledge of the spending keys and nullifying the tokens).
- The attached tracer memo encrypts the correct opening of the asset-record commitment. The tracer memo is used later when a freezer attempts to freeze the issued asset record.

To facilitate global policies over asset issuers (e.g., asset cap limit), the amount, asset type and policy are published as well. Looking ahead, we stress that the validator should check that the issued asset policy is valid and well-formed. In particular, the public keys in the policy should be on the curve and in a large subgroup so that users will not misuse public keys and leak private information. Fortunately, this check can be performed outside the circuit as the policy is public.

The SNARK statement to be proved is the following:

- **Public instance:** The public inputs are:
  - rt: Merkle accumulator digest.
  - at^*: native asset type.
  - $(v, \mathsf{at}, \mathsf{policy})$: minted amount and issued asset type and policy.
  - arc: issued asset record commitment.
  - $\mathsf{arc}_{\mathsf{chg}}$: fee changes commitment.
  - fee: transaction fee amount (in native assets).
  - $\mathsf{nl}^{\mathsf{in}}$: input nullifier.
  - $\mathsf{memo}_{\mathsf{trc}}$: a tracer memo ciphertext.
  - $\widehat{vk}$: a fresh random public key for verifying owner memos (see Remark 1)
- **Secret witness:** The secret witnesses are:
  - $\mathsf{oar} := (v, \mathsf{at}, \mathsf{upk}, \mathsf{policy}, b_{\mathsf{frz}}, \gamma)$: the opening of the issued asset record commitment arc. Here upk is the recipient's address, b_{frz} is a boolean indicating whether the ARC is frozen, $\mathsf{policy} := (\mathsf{threshold}, \mathsf{tpk}, \mathsf{ipk}, \mathsf{fpk}, \mathsf{reveal})$ is the policy of the issued asset.
  - $(\mathsf{usk}^{\mathsf{in}}, \mathsf{upk}^{\mathsf{in}})$: asset issuer's user key pair.
  - $\mathsf{uid}_{\mathsf{in}}$ is the unique identifier of the ARC $\mathsf{arc}^{\mathsf{in}}$ to be consumed and use to pay fee ($(\mathsf{uid}_{\mathsf{in}}, \mathsf{arc}^{\mathsf{in}})$ is the accumulated element).
  - $\mathsf{oar}_{\mathsf{in}} := (v^{\mathsf{in}}, \mathsf{at}^*, \mathsf{upk}^{\mathsf{in}}, \bot, b^{\mathsf{in}}_{\mathsf{frz}}, \gamma^{\mathsf{in}})$: the opening of input asset record commitment $\mathsf{arc}^{\mathsf{in}}$. We note that the native asset policy is dummy.
  - $\mathsf{oar}_{\mathsf{chg}} := (\mathsf{chg}, \mathsf{at}^*, \mathsf{upk}^{\mathsf{in}}, \bot, b^{\mathsf{chg}}_{\mathsf{frz}}, \gamma_{\mathsf{chg}})$: the opening of change commitment $\mathsf{arc}_{\mathsf{chg}}$. We note that the native asset policy is dummy.
  - $(s_{\mathsf{at}}, \mathsf{aux})$: the issued asset type at's secret seed and auxiliary info.
  - π_{in}: the membership proof for $\mathsf{elem}_{\mathsf{in}}$.
  - γ_e: encryption randomness.
- **Statement:** Denote by $\ell_{\max}$ a predefined integer bound (e.g., $\ell_{\max} = 64$). Given a public instance x and a secret witness w, the relation $\mathcal{R}_{\mathsf{mint}}(x, w)$ holds if and only if the following conditions all hold:
  - $\mathsf{arc} = \mathsf{Com}(\mathsf{oar})$.
  - $\mathsf{arc}_{\mathsf{chg}} = \mathsf{Com}(\mathsf{oar}_{\mathsf{chg}})$.
  - $\mathsf{at} = \mathsf{derive}_{\mathsf{at}}(s_{\mathsf{at}}, \mathsf{aux})$. Proof-of-Knowledge of asset type secret seed.
  - $v^{\mathsf{in}} = \mathsf{fee} + \mathsf{chg}$. Fee plus the change equals the input amount.
  - $\mathsf{PoB}(\mathsf{rt}, \mathsf{nl}^{\mathsf{in}}, \pi_{\mathsf{in}}, \mathsf{uid}_{\mathsf{in}}, \mathsf{oar}_{\mathsf{in}}, \mathsf{usk}^{\mathsf{in}}, \mathrm{USER}) = 1$ where $\mathrm{USER} = 1$ is a constant. Note that the freezer public key is dummy as the native asset policy is dummy and does not have a freezer public key.
  - $0 \le v < 2^{\ell_{\max}}$ and $0 \le \mathsf{chg} < 2^{\ell_{\max}}$.
  - $b_{\mathsf{frz}} = b^{\mathsf{in}}_{\mathsf{frz}} = b^{\mathsf{chg}}_{\mathsf{frz}} = 0$. All ARCs are not frozen.
  - $\mathsf{tpk} = \bot$, or $\mathsf{memo}_{\mathsf{trc}} = \mathsf{Enc}_{\mathsf{tpk}}(\mathsf{oar}; \gamma_e)$.

  Recall that Com, $\mathsf{derive}_{\mathsf{at}}$, PoB, Enc and range-checks are the building block algorithms described in Section 3.2.1.

Remark 2 (Optimizing Tracer Memo). In practice, the tracer memo only needs to encrypt user public key upk and opening randomness γ, as other fields of oar are public.

Remark 3. We can easily extend $\mathsf{tx}_{\mathsf{mint}}$ transactions to support issuing multiple ARCs in a single transaction. The only change is to replace arc with a vector of ARCs and prove their correctness in the circuit.

Remark 4. It is without loss of generality to support only a single nullifier for transaction fees. If an asset issuer wants to burn multiple ARCs for fee, she can first merge the ARCs into a single ARC via a $\mathsf{tx}_{\mathsf{axfr}}$ transaction.

3.2.3 Anonymous Transfer An anonymous transfer transaction works by nullifying non-dummy input asset records (of senders) and creating some output asset records (for receivers). The transaction further proves:

- the sender knows the spending keys of input asset records;
- the asset type and the policy do not change in the output asset records;
- the amount balance is preserved;
- all of the asset records are spendable and not frozen;
- the sender possesses a valid identity credential if a credential public key is presented in the policy;
- the tracer memo is honestly computed if a tracer public key is presented in the policy;
- the sender has burnt some amounts of native tokens as transaction fees.

We allow *dummy* input asset record with predefined asset code ($\mathsf{at}_{\mathsf{dummy}}$) and public key ($\mathsf{upk}_{\mathsf{dummy}}$), zero amount, an empty policy ($\perp$), and a random blinding factor. Accumulator membership check in Proof of burn is not required for dummy records. Finally, the tracer memo data associated with a dummy record contains default values.

Denote by $n \in \mathbb{N}$ the number of inputs and $m \in \mathbb{N}$ the number of outputs. Values $\mathsf{at}_{\mathsf{dummy}}$, $\mathsf{upk}_{\mathsf{dummy}}$ and special symbol $\perp$ are hardcoded, and hence, not part of the explicit public input. The SNARK statement for a $\mathsf{tx}_{\mathsf{axfr}}^{n,m}$ transaction is the following:

- **Public instance:** The public inputs are:
  - rt: Merkle accumulator digest.
  - T: current timestamp in the system.
  - at^*: native asset type.
  - fee: transaction fee amount (in native assets).
  - $(\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}$: a list of input nullifiers.
  - $(\mathsf{arc}_i^{\mathsf{out}})_{i\in[m]}$: a list of output asset-record commitments.
  - $\mathsf{memo}_{\mathsf{trc}}$: a tracer memo ciphertext.
  - $\widehat{vk}$: a fresh random public key for verifying owner memos (see Remark 1)
- **Secret witness:** The secret inputs are:
  - $(\mathsf{at}, \mathsf{policy})$: transfer asset type and policy. Here $\mathsf{policy} := (\mathsf{threshold}, \mathsf{tpk}, \mathsf{ipk}, \mathsf{fpk}, \mathsf{reveal})$ contains a threshold parameter, a tracer public key, a credential public key, freezer public key, and a reveal map.
  - $(b_0, b_1, b_2, \mathsf{attrmap})$: a binary vector that is supposed to be the binary representation of reveal.
  - $(\mathsf{uid}_i^{\mathsf{in}}, \pi_i^{\mathsf{in}}, \mathsf{oar}_i^{\mathsf{in}}, \mathsf{usk}_i^{\mathsf{in}})_{i\in[n]}$: the information for burning input asset record commitments. For the i-th ($1 \le i \le n$) input, $\mathsf{uid}_i^{\mathsf{in}}$ is a unique identifier of the ARC $\mathsf{arc}_i^{\mathsf{in}}$ to be consumed. π_i^{in} is a membership proof for the accumulated element $(\mathsf{uid}_i^{\mathsf{in}}, \mathsf{arc}_i^{\mathsf{in}})$. $\mathsf{oar}_i^{\mathsf{in}} := (v_i^{\mathsf{in}}, \mathsf{at}_i^{\mathsf{in}}, \mathsf{upk}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}}, b_{\mathsf{frz},i}^{\mathsf{in}}, \gamma_i^{\mathsf{in}})$ is an asset record opening. $\mathsf{usk}_i^{\mathsf{in}}$ is the user secret key with respect to $\mathsf{upk}_i^{\mathsf{in}}$.
  - $(\mathsf{cred}_i, \mathsf{attrs}_i, \mathsf{expiry}_i)_{i\in[n]}$: identity credentials for the input asset records, and the attribute lists and expiration time signed in the credentials.
  - $(\mathsf{oar}_i^{\mathsf{out}} := (v_i^{\mathsf{out}}, \mathsf{at}_i^{\mathsf{out}}, \mathsf{upk}_i^{\mathsf{out}}, \mathsf{policy}_i^{\mathsf{out}}, b_{\mathsf{frz},i}^{\mathsf{out}}, \gamma_i^{\mathsf{out}}))_{i\in[m]}$: output asset record openings.
  - γ_e: encryption randomness.
- **Statement:** Denote by $\ell_{\max}$, ℓ_{attrs}, ℓ_T predefined integer bounds (e.g., $\ell_{\max} = 64$, $\ell_{\mathsf{attrs}} = 8$, $\ell_T = 32$). Given a public instance x and a secret witness w, the relation $\mathcal{R}_{\mathsf{xfr}}(x, w)$ holds if and only if the following conditions all hold:
  - Asset type and policy consistency:[3]
    - $(\mathsf{at}_1^{\mathsf{in}}, \mathsf{policy}_1^{\mathsf{in}}) = (\mathsf{at}_1^{\mathsf{out}}, \mathsf{policy}_1^{\mathsf{out}}) = (\mathsf{at}^*, \perp)$.
    - For all $i \in [2, n]$: $\mathsf{at}_i^{\mathsf{in}} = \mathsf{at}_{\mathsf{dummy}}$ or $(\mathsf{at}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}}) = (\mathsf{at}, \mathsf{policy})$
    - For all $j \in [2, m]$: $(\mathsf{at}_j^{\mathsf{out}}, \mathsf{policy}_j^{\mathsf{out}}) = (\mathsf{at}, \mathsf{policy})$

[3] Looking ahead, the constraints can be satisfied for free by directly assigning $(\mathsf{at}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}})$ (and $(\mathsf{at}_i^{\mathsf{out}}, \mathsf{policy}_i^{\mathsf{out}})$) to $(\mathsf{at}^*, \perp)$ (or $(\mathsf{at}, \mathsf{policy})$). We add the statements only for clarity.

    - ⋆ For all $i \in [2, n]$: $\mathsf{at}_i^{\mathsf{in}} \neq \mathsf{at}_{\mathsf{dummy}}$ or $(v_i^{\mathsf{in}} = 0$ and $\mathsf{policy}_i^{\mathsf{in}} = \perp$ and $\mathsf{upk}_i^{\mathsf{in}} = \mathsf{upk}_{\mathsf{dummy}})$.
  - $b_{\mathsf{frz},i}^{\mathsf{in}} = b_{\mathsf{frz},j}^{\mathsf{out}} = 0$ for all $i \in [n]$, $j \in [m]$. No ARCs are frozen.
  - For all $i \in [n]$:
    - ⋆ $\mathsf{PoB}(\mathsf{rt}, \mathsf{nl}_i^{\mathsf{in}}, \pi_i^{\mathsf{in}}, \mathsf{uid}_i^{\mathsf{in}}, \mathsf{oar}_i^{\mathsf{in}}, \mathsf{usk}_i^{\mathsf{in}}, \mathrm{USER}) = 1$ where $\mathrm{USER} = 1$ is a constant.
    - ⋆ $T \leq \mathsf{expiry}_i < 2^{\ell_T}$ (i.e., the credential has not expired yet).
    - ⋆ $\mathsf{ipk}_i^{\mathsf{in}} = \perp$ or $\mathsf{Vfy}_{\mathsf{ac}}(\mathsf{ipk}, \mathsf{cred}_i, \boldsymbol{m}_i = (\mathsf{expiry}_i, \mathsf{upk}_i^{\mathsf{in}}, \mathsf{attrs}_i)) = 1$.
  - For all $i \in [m]$:
    - ⋆ $\mathsf{arc}_i^{\mathsf{out}} = \mathsf{Com}(\mathsf{oar}_i^{\mathsf{out}})$.
    - ⋆ $0 \leq v_i^{\mathsf{out}} < 2^{\ell_{\max}}$.
  - $\mathsf{Balance}(\mathsf{at}^*, \mathsf{at}, \mathsf{fee}, (v_i^{\mathsf{in}})_{i\in[n]}, (v_i^{\mathsf{out}})_{i\in[m]}) = 1$.
  - $(b_0, b_1, b_2, \mathsf{attrmap}) = \mathsf{range}(\mathsf{reveal}, 3 + \ell_{\mathsf{attrs}})$. That is, $0 \leq \mathsf{reveal} < 2^{3+\ell_{\mathsf{attrs}}}$ and $(b_0, b_1, b_2, \mathsf{attrmap})$ is the binary representation of reveal.
  - For all $i \in [2..n]$ $\tilde{b_{0,i}} \leftarrow b_0 \vee (\mathsf{at}_i^{\mathsf{in}} = \mathsf{at}_{\mathsf{dummy}})$.
  - Derive $\Delta_{\mathsf{threshold}} \leftarrow \mathsf{threshold} - \sum_{i\in[2..n]} v_i^{\mathsf{in}}$.
  - Derive
$$b_{\mathsf{threshold}} \leftarrow (\Delta_{\mathsf{threshold}} \geq 0) \wedge (\Delta_{\mathsf{threshold}} < 2^{\ell_{\max}}) .$$
    Note that $b_{\mathsf{threshold}} = 1$ if the transfer amount does not exceed $\mathsf{threshold}$, and $b_{\mathsf{threshold}} = 0$ otherwise.
  - $\mathsf{tpk} = \perp$ or $b_{\mathsf{threshold}} = 1$ or $\mathsf{memo}_{\mathsf{trc}} = \mathsf{Enc}_{\mathsf{tpk}}(\boldsymbol{m}; \gamma_{\mathsf{trc}})$ where the message vector $\boldsymbol{m}$ is
$$\Big(\mathsf{at}, (\tilde{b_{0,i}} \cdot \mathsf{upk}_i^{\mathsf{in}}, b_1 \cdot v_i^{\mathsf{in}}, b_2 \cdot \gamma_i^{\mathsf{in}}, \mathsf{attrmap} \odot \mathsf{attrs}_i)_{i\in\{2\ldots n\}}, (b_0 \cdot \mathsf{upk}_j^{\mathsf{out}}, b_1 \cdot v_j^{\mathsf{out}}, b_2 \cdot \gamma_j^{\mathsf{out}})_{j\in\{2\ldots m\}}\Big) .$$
    Here $\boldsymbol{a} \odot \boldsymbol{b}$ denotes component-wise multiplications between vector $\boldsymbol{a}$ and $\boldsymbol{b}$.

Recall that PoB, Com, $\mathsf{Vfy}_{\mathsf{ac}}$, $\mathsf{Balance}$, Enc and rangeproofs are the building block algorithms described in Section 3.2.1.

3.2.4 (Un)Freezing A (un)freezing transaction works by nullifying a set of asset records to be (un)frozen, and creating the corresponding output asset records with flipped frozen states. The transaction further proves:

- the freezer public keys in the input asset records' policies are non-empty;
- the freezer knows the spending/nullifying keys of the input asset records;
- the output asset records preserves the same asset record info (e.g., amount, asset type, address, policy) to that in the corresponding input asset records;
- the freezer has burnt some amounts of native tokens as transaction fees.

Denote by $n \in \mathbb{N}$ the number of inputs/outputs. The SNARK statement for a $\mathsf{tx}_{\mathsf{frz}}^n$ transaction is the following:

- **Public instance:** The public inputs are:
  - rt: Merkle accumulator digest.
  - at^*: native asset type.
  - fee: transaction fee amount (in native assets).
  - $(\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}$: input nullifiers.
  - $(\mathsf{arc}_i^{\mathsf{out}})_{i\in[n]}$: output asset-record commitments.
- **Secret witness:** The secret inputs are:
  - $(\mathsf{uid}_i^{\mathsf{in}}, \pi_i^{\mathsf{in}}, \mathsf{oar}_i^{\mathsf{in}}, \mathsf{sk}_i^{\mathsf{in}})_{i\in[n]}$: the information for burning input asset record commitments. For the i-th $(1 \leq i \leq n)$ input, $\mathsf{uid}_i^{\mathsf{in}}$ is a unique identifier of the ARC $\mathsf{arc}_i^{\mathsf{in}}$ to be consumed. π_i^{in} is a membership proof for the accumulated element $(\mathsf{uid}_i^{\mathsf{in}}, \mathsf{arc}_i^{\mathsf{in}})$. $\mathsf{oar}_i^{\mathsf{in}} := (v_i^{\mathsf{in}}, \mathsf{at}_i^{\mathsf{in}}, \mathsf{upk}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}}, b_{\mathsf{frz},i}^{\mathsf{in}}, \gamma_i^{\mathsf{in}})$ is an asset record opening. $\mathsf{sk}_i^{\mathsf{in}}$ is the secret key used to derive nullifier key.
  - $(\mathsf{oar}_i^{\mathsf{out}} := (v_i^{\mathsf{out}}, \mathsf{at}_i^{\mathsf{out}}, \mathsf{upk}_i^{\mathsf{out}}, \mathsf{policy}_i^{\mathsf{out}}, b_{\mathsf{frz},i}^{\mathsf{out}}, \gamma_i^{\mathsf{out}}))_{i\in[m]}$: output asset record openings.
- **Statement:** Given a public instance x and a secret witness w, the relation $\mathcal{R}_{\mathsf{frz}}(x, w)$ holds if and only if the following conditions all hold:
  - Native ARCs validity:
    - ⋆ $(\mathsf{at}_1^{\mathsf{in}}, \mathsf{policy}_1^{\mathsf{in}}) = (\mathsf{at}_1^{\mathsf{out}}, \mathsf{policy}_1^{\mathsf{out}}) = (\mathsf{at}^*, \perp)$.
    - ⋆ $b_{\mathsf{frz},1}^{\mathsf{in}} = b_{\mathsf{frz},1}^{\mathsf{out}} = 0$. Native ARCs are not frozen.
    - ⋆ $v_1^{\mathsf{in}} = v_1^{\mathsf{out}} + \mathsf{fee}$. Fee plus the change equals the input amount.
    - ⋆ $\mathsf{PoB}(\mathsf{rt}, \mathsf{nl}_1^{\mathsf{in}}, \pi_1^{\mathsf{in}}, \mathsf{uid}_1^{\mathsf{in}}, \mathsf{oar}_1^{\mathsf{in}}, \mathsf{sk}_1^{\mathsf{in}}, \mathrm{USER}) = 1$ where $\mathrm{USER} = 1$ is a constant. Note that the freezer public key (in the policy) is dummy as the native asset policy is dummy and does not have a freezer public key.

- For all $i \in [2, n]$:
  - $\star$ $b^{\mathsf{in}}_{\mathsf{frz},i} + b^{\mathsf{out}}_{\mathsf{frz},i} = 1$. Freezing flag flipped.
  - $\star$ $(v^{\mathsf{in}}_i, \mathsf{at}^{\mathsf{in}}_i, \mathsf{upk}^{\mathsf{in}}_i, \mathsf{policy}^{\mathsf{in}}_i) = (v^{\mathsf{out}}_i, \mathsf{at}^{\mathsf{out}}_i, \mathsf{upk}^{\mathsf{out}}_i, \mathsf{policy}^{\mathsf{out}}_i)$. Output ARCs preserves the amounts, addresses, asset types and policies of input ARCs.
  - $\star$ $(\mathsf{fpk}^{\mathsf{in}}_i \neq \perp) \vee (\mathsf{at}^{\mathsf{in}}_i = \mathsf{at}_{\mathsf{dummy}})$. That is, the freezer's public key $\mathsf{fpk}^{\mathsf{in}}_i$ can be $\perp$ only if the record is dummy.
  - $\star$ $\mathsf{PoB}(\mathsf{rt}, \mathsf{nl}^{\mathsf{in}}_i, \pi^{\mathsf{in}}_i, \mathsf{uid}^{\mathsf{in}}_i, \mathsf{oar}^{\mathsf{in}}_i, \mathsf{sk}^{\mathsf{in}}_i, \text{FREEZER}) = 1$ where FREEZER $= 0$ is a constant. The nullifier key is derived from freezer secret key $\mathsf{sk}^{\mathsf{in}}_i$ and user public key $\mathsf{upk}^{\mathsf{in}}_i$.
- $\mathsf{arc}^{\mathsf{out}}_i = \mathsf{Com}(\mathsf{oar}^{\mathsf{out}}_i)$ for all $i \in [n]$.

Recall that PoB, Com are the building block algorithms described in Section 3.2.1.

3.3 Scheme Instantiation

We now proceed to describe a construction of CAP scheme that contains all algorithms required and specified in Section 2.2 and a few more auxiliary ledger-specific algorithms to complete the entire system workflow.

$\mathsf{Setup}(1^\lambda, N, \ell_{\max}, \ell_{\mathsf{attrs}}, \ell_T, \mathsf{at}^*) \to \mathsf{pp}$

- *inputs*:
  - security parameter λ
  - SNARK circuit bound N
  - max coin value $2^{\ell_{\max}}$
  - max attributes length ℓ_{attrs}
  - max expiry value 2^{ℓ_T}
  - native asset type: at^*
- *outputs*: public parameters pp

1. Set description of field upon which circuits operate: $\langle \mathbb{F}_p \rangle$
2. Generate universal SRS: $\mathsf{srs} \xleftarrow{\$} \mathcal{G}(1^\lambda, N)$
3. Pre-process circuit for $\mathsf{tx}_{\mathsf{mint}}$: $(\sigma_{\mathsf{mint}}, \tau_{\mathsf{mint}}) \leftarrow \mathcal{I}^{\mathsf{srs}}(i_{\mathsf{mint}})$
   Pre-process circuit for $\mathsf{tx}_{\mathsf{axfr}}$: $(\sigma_{\mathsf{xfr}}, \tau_{\mathsf{xfr}}) \leftarrow \mathcal{I}^{\mathsf{srs}}(i_{\mathsf{xfr}})$
   Pre-process circuit for $\mathsf{tx}_{\mathsf{frz}}$: $(\sigma_{\mathsf{frz}}, \tau_{\mathsf{frz}}) \leftarrow \mathcal{I}^{\mathsf{srs}}(i_{\mathsf{frz}})$
4. Set $\mathsf{pp} := (\lambda, \ell_{\max}, \ell_{\mathsf{attrs}}, \ell_T, \mathsf{at}^*, \langle \mathbb{F}_p \rangle, \sigma_{\mathsf{mint}}, \tau_{\mathsf{mint}}, \sigma_{\mathsf{xfr}}, \tau_{\mathsf{xfr}}, \sigma_{\mathsf{frz}}, \tau_{\mathsf{frz}})$
5. Output pp

$\mathsf{CredKeyGen}(\mathsf{pp}) \to (\mathsf{isk}, \mathsf{ipk})$

- *inputs*: public parameter pp
- *outputs*:
  - issuer secret key isk
  - issuer public key ipk

1. Generate keys for credential issuer: $(\mathsf{isk}, \mathsf{ipk}) \leftarrow \mathsf{Gen}_{\mathsf{cred\_issuer}}(1^\lambda)$
2. Output $(\mathsf{isk}, \mathsf{ipk})$

$\mathsf{UserKeyGen}(\mathsf{pp}) \to (\mathsf{usk}, \mathsf{upk})$

- *inputs*: public parameter pp
- *outputs*:
  - user secret key usk
  - user public key upk

1. Generate encryption keys: $(pk_{\mathsf{enc}}, sk_{\mathsf{enc}}) \xleftarrow{\$} \mathsf{Gen}_{\mathsf{enc}}(1^\lambda)$
2. Generate address secret seed (for identification): $\mathsf{addr}_{\mathsf{sk}} \xleftarrow{\$} \mathbb{F}_p$
   Derive public address: $\mathsf{addr}_{\mathsf{pk}} \leftarrow \mathsf{derive}_{\mathsf{pk}}(\mathsf{addr}_{\mathsf{sk}})$
3. Set $\mathsf{usk} := (sk_{\mathsf{enc}}, \mathsf{addr}_{\mathsf{sk}})$, $\mathsf{upk} := (pk_{\mathsf{enc}}, \mathsf{addr}_{\mathsf{pk}})$
4. Output $(\mathsf{usk}, \mathsf{upk})$

$\mathsf{TracerKeyGen}(\mathsf{pp}) \to (\mathsf{tsk}, \mathsf{tpk})$

- *inputs*: public parameter pp
- *outputs*:
  - tracer secret key tsk
  - tracer public key tpk

1. Generate keys for tracer: $(pk_{\mathsf{enc}}, sk_{\mathsf{enc}}) \xleftarrow{\$} \mathsf{Gen}_{\mathsf{enc}}(1^{\lambda})$
2. Set $\mathsf{tsk} := sk_{\mathsf{enc}}, \mathsf{tpk} := pk_{\mathsf{enc}}$
3. Output $(\mathsf{tsk}, \mathsf{tpk})$

$\underline{\mathsf{FreezerKeyGen}(\mathsf{pp}) \rightarrow (\mathsf{fsk}, \mathsf{fpk})}$

- *inputs*: public parameter pp
- *outputs*:
  - freezer secret key fsk
  - freezer public key fpk

1. Generate secret key: $\mathsf{fsk} \xleftarrow{\$} \mathbb{F}_p$
   Derive public address: $\mathsf{fpk} \leftarrow \mathsf{derive}_{\mathsf{fpk}}(\mathsf{fsk})$
2. Output $(\mathsf{fsk}, \mathsf{fpk})$

$\underline{\mathsf{RequestCred}(\mathsf{pp}, \mathsf{ipk}, \mathsf{usk}, \mathsf{upk}, \mathsf{msg}) \rightarrow \mathsf{request}}$

- *inputs*:
  - public parameter pp
  - credential issuer public key ipk
  - user secret key usk
  - user public key upk
  - Message to issuer msg
- *outputs*: request $\mathsf{request}$

1. Compute $\mathsf{Sign}(\mathsf{usk}, \mathsf{upk}||\mathsf{ipk}||\mathsf{msg})) \rightarrow s$
2. Output $\mathsf{request} := (s, \mathsf{msg})$

$\underline{\mathsf{VerifyCredReq}(\mathsf{pp}, \mathsf{ipk}, \mathsf{upk}, \mathsf{request}) \rightarrow b}$

- *inputs*:
  - public parameter pp
  - credential issuer public key ipk
  - user (credential owner) public key upk
  - user request $\mathsf{request}$
- *outputs*: $0/1$

1. Parse $\mathsf{request} \rightarrow (s, \mathsf{msg})$
2. return $\mathsf{Vfy}(\mathsf{upk}, s, \mathsf{upk}||\mathsf{ipk}||\mathsf{msg})$

$\underline{\mathsf{IssueCred}(\mathsf{pp}, \mathsf{isk}, \mathsf{upk}, \textbf{attrs}, \mathsf{expiry}) \rightarrow \mathsf{cred}}$

- *inputs*:
  - public parameter pp
  - credential issuer secret key isk
  - user public key upk
  - user identity attributes **attrs**
  - expiry timestamp of the credential expiry
- *outputs*: issued credential cred

1. Check $\mathsf{expiry} < 2^{\mathsf{pp}.\ell_T}$ and $\mathsf{attrs}.len + 1 = pp.\ell_{\mathsf{attrs}}$
   If failed, returns $\perp$
2. Compute $\mathsf{cred} \leftarrow \mathsf{Issue}_{\mathsf{cred}}(\mathsf{isk}, \mathsf{upk}, \mathsf{attrs}||\mathsf{expiry})$ as follows:
   - $s \leftarrow \mathsf{Sign}(\mathsf{isk}, (\mathsf{expiry}||\mathsf{upk}||\mathsf{attrs}))$
   - output $\mathsf{cred} := (s, \mathsf{attrs}, \mathsf{expiry})$
3. Output cred

$\mathsf{IssueAsset}(\mathsf{pp}, \mathbf{oar}_{\mathsf{out}}, \mathsf{aux}_{\mathsf{at}}, \mathsf{aux}_{\mathsf{fee}}, \mathsf{rt}, \mathsf{aux}_{\mathsf{memo}}) \to (\mathsf{tx}_{\mathsf{mint}}, \mathsf{memo}_{\mathsf{own}}, \sigma_{\mathsf{memo}})$

- *inputs*:
  - public parameter $\mathsf{pp} := (\mathsf{at}^*, \sigma_{\mathsf{mint}}, \ldots)$
  - opening of output ARCs: $\mathbf{oar}_{\mathsf{out}} := (v_j^{\mathsf{out}}, \mathsf{at}_j^{\mathsf{out}}, \mathsf{upk}_j^{\mathsf{out}}, \mathsf{policy}_j^{\mathsf{out}}, b_{\mathsf{frz},j}^{\mathsf{out}}; \gamma_j^{\mathsf{out}})_{j\in\{1,2\}}$ where the first is fee change, the second is the ARC to be minted
  - auxiliary opening of asset type to be minted: $\mathsf{aux}_{\mathsf{at}} := (s_{\mathsf{at}}, \mathsf{aux}')$ such that $\mathsf{at} = \mathsf{derive}_{\mathsf{at}}(s_{\mathsf{at}}, \mathsf{aux}')$ where s_{at} is the issued asst type's secret seed
  - auxiliary info about the fee/input ARC: $\mathsf{aux}_{\mathsf{fee}} := (\mathsf{usk}_{\mathsf{fee}}, \mathsf{oar}_{\mathsf{fee}}, \mathsf{uid}_{\mathsf{fee}}, \pi_{\mathsf{fee}})$ where
    $\mathsf{usk}_{\mathsf{fee}}$ is fee payer secret
    $\mathsf{oar}_{\mathsf{fee}} := (v_{\mathsf{fee}}, \mathsf{at}^*, \mathsf{upk}_{\mathsf{fee}}, \bot, b_{\mathsf{frz}}^{\mathsf{fee}}; \gamma_{\mathsf{fee}})$ is the opening of fee/input asset record
    uid is the unique identifier of the fee ARC inside the accumulator
    π is the accumulator membership proof of fee ARC
  - Merkle accumulator digest: rt
  - auxiliary info about owner memo: $\mathsf{aux}_{\mathsf{memo}} := (\widehat{vk}, \widehat{sk})$ is a pair of fresh random key pair (see Remark 1)
- *outputs*:
  - asset issuance transaction body: $\mathsf{tx}_{\mathsf{mint}}$
  - an owner memos for the issued asset record: $\mathsf{memo}_{\mathsf{own}}$
  - signature over all owner memos under $\widehat{vk}$: σ_{memo}

1. Set $\mathsf{fee} = v_{\mathsf{fee}} - v_1^{\mathsf{out}}$
2. Compute fee ARC: $\mathsf{arc}_{\mathsf{fee}} = \mathsf{Com}(\boldsymbol{m}; \gamma_{\mathsf{fee}})$ where $\boldsymbol{m} := (v_{\mathsf{fee}}, \mathsf{at}^*, \mathsf{upk}_{\mathsf{fee}}, \bot, b_{\mathsf{frz}}^{\mathsf{fee}})$
   Compute nullifier key: $\mathsf{nk} = \mathsf{derive}_{\mathsf{nk}}(\mathsf{usk}_{\mathsf{fee}}, \infty)$ where $\infty = (0, 1)$ is the neutral point
   Compute nullifier for fee ARC: $\mathsf{nl}_{\mathsf{fee}} = \mathsf{Nullify}_{\mathsf{nk}}(\mathsf{elem}_{\mathsf{fee}})$ where $\mathsf{elem}_{\mathsf{fee}} := (\mathsf{uid}_{\mathsf{fee}}, \mathsf{arc}_{\mathsf{fee}})$
3. For $j \in \{1, 2\}$:
   (a) Compute output ARCs: $\mathsf{arc}_j^{\mathsf{out}} = \mathsf{Com}(\boldsymbol{m}_j; \gamma_j^{\mathsf{out}})$ where $\boldsymbol{m}_j := (v_j^{\mathsf{out}}, \mathsf{at}_j^{\mathsf{out}}, \mathsf{upk}_j^{\mathsf{out}}, \mathsf{policy}_j^{\mathsf{out}}, b_{\mathsf{frz},j}^{\mathsf{out}})$
4. Prepare an owner memo: $\mathsf{memo}_{\mathsf{own}} = \mathsf{Enc}(\mathsf{upk}_2^{\mathsf{out}}, \boldsymbol{m})$ where $\boldsymbol{m} := (v_2^{\mathsf{out}}, \mathsf{at}_2^{\mathsf{out}}, \mathsf{policy}_2^{\mathsf{out}}, b_{\mathsf{frz},2}^{\mathsf{out}}, \gamma_2^{\mathsf{out}})$
5. If $\mathsf{tpk} \neq \bot$, prepare a tracer memo: $\mathsf{memo}_{\mathsf{trc}} = \mathsf{Enc}(\mathsf{policy.tpk}, (\boldsymbol{m}_2, \gamma_2^{\mathsf{out}}); \gamma_{\mathsf{trc}})$ where $\gamma_{\mathsf{trc}} \xleftarrow{\$} \mathbb{F}$.
6. Set public instance: $x := (\mathsf{rt}, \mathsf{at}^*, v, \mathsf{at}, \mathsf{policy}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in\{1,2\}}, \mathsf{fee}, \mathsf{nl}_{\mathsf{fee}}, \mathsf{memo}_{\mathsf{trc}}, \widehat{vk})$
   Set secret witness: $w := (\mathbf{oar}_{\mathsf{out}}, \mathsf{aux}_{\mathsf{at}}, \mathsf{aux}_{\mathsf{fee}}, \gamma_{\mathsf{trc}})$
7. Compute SNARK proof: $\pi_{\mathsf{mint}} \leftarrow \mathcal{P}(\sigma_{\mathsf{mint}}, x, w)$
8. Set $\mathsf{tx}_{\mathsf{mint}} := (\mathsf{rt}, \mathsf{nl}_{\mathsf{fee}}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in\{1,2\}}, (v, \mathsf{at}, \mathsf{policy}), \pi_{\mathsf{mint}}, \mathsf{memo}_{\mathsf{trc}}, \mathsf{fee}, \widehat{vk})$
9. Sign the owner memo: $\sigma_{\mathsf{memo}} = \mathsf{Sign}(\widehat{sk}, \mathsf{memo}_{\mathsf{own}})$
10. Output $(\mathsf{tx}_{\mathsf{mint}}, \mathsf{memo}_{\mathsf{own}}, \sigma_{\mathsf{memo}})$

$\mathsf{AnonXfr}_{n,m}(\mathsf{pp}, \mathbf{usk}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{in}}, \mathbf{aux}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{out}}, \mathsf{rt}, T, \mathsf{aux}_{\mathsf{memo}}) \to (\mathsf{tx}_{\mathsf{axfr}}, \mathbf{memo}_{\mathbf{own}}, \sigma_{\mathsf{memo}})$

- *inputs*:
  - public parameter: $\mathsf{pp} := (\mathsf{at}^*, \sigma_{\mathsf{xfr}}, \ldots)$
  - n sender secret keys: $\mathbf{usk}_{\mathsf{in}} := (\mathsf{usk}_i^{\mathsf{in}})_{i\in[n]}$
  - n openings of input ARC[4]: $\mathbf{oar}_{\mathsf{in}} := (v_i^{\mathsf{in}}, \mathsf{at}_i^{\mathsf{in}}, \mathsf{upk}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}}, b_{\mathsf{frz},i}^{\mathsf{in}}; \gamma_i^{\mathsf{in}})_{i\in[n]}$
  - n auxiliary info about input ARC: $\mathbf{aux}_{\mathsf{in}} := (\mathsf{uid}_i^{\mathsf{in}}, \pi_i^{\mathsf{in}}, \mathsf{attrs}_i, \mathsf{expiry}_i, \mathsf{cred}_i)_{i\in[n]}$ where
    $\mathsf{uid}^{\mathsf{in}}$ is the unique identifier of input ARC inside the accumulator
    π_i^{in} is an ARC membership proof
    attrs_i is the identity attributes list
    expiry_i is the expiration time signed in the credential
    cred_i is the credential attesting for the foregoing attributes and expiry
  - m openings of output ARC: $\mathbf{oar}_{\mathsf{out}} := (v_j^{\mathsf{out}}, \mathsf{at}_j^{\mathsf{out}}, \mathsf{upk}_j^{\mathsf{out}}, \mathsf{policy}_j^{\mathsf{out}}, b_{\mathsf{frz},j}^{\mathsf{out}}; \gamma_j^{\mathsf{out}})_{j\in[m]}$
  - Merkle accumulator digest: rt
  - current system timestamp (block height)[5]: T
  - auxiliary info about owner memo: $\mathsf{aux}_{\mathsf{memo}} := (\widehat{vk}, \widehat{sk})$ is a pair of fresh random key pair (see Remark 1)
- *outputs*:

[4] In practice, these openings are decrypted from `OwnerMemo` of an arc the user owns.

[5] When passing in the timestamp, try to set T slightly bigger than exact current block height to give buffer since it may take a few blocks for this transaction to be included and the proof of non-expiry on credential has to withstand a slightly larger system timestamp by then

- anonymous transfer transaction body: $\mathsf{tx}_{\mathsf{axfr}}$
- list of owner memos (one for each output record except for the 1st output as fee change): $\mathbf{memo}_{\mathbf{own}}$
- signature over all owner memos under $\widehat{vk}$: σ_{memo}

1. Set $\mathsf{fee} = v_1^{\mathsf{in}} - v_1^{\mathsf{out}}$
   Set $\mathsf{at} := \mathsf{at}_2^{\mathsf{in}}$
   Set $\mathsf{policy} := \mathsf{policy}_2^{\mathsf{in}}$ where $\mathsf{policy} := (\mathsf{threshold}, \mathsf{tpk}, \mathsf{ipk}, \mathsf{fpk}, \mathsf{reveal})$ and $\mathsf{reveal} := (b_0, b_1, b_2, \mathsf{attrmap})$[6]
2. Check $\mathsf{at}_1^{\mathsf{in}} = \mathsf{at}_1^{\mathsf{out}} = \mathsf{at}^* \wedge \mathsf{policy}_1^{\mathsf{in}} = \mathsf{policy}_1^{\mathsf{out}} = \bot$
   Check $\mathsf{at}_i^{\mathsf{in}} = \mathsf{at} \wedge \mathsf{policy}_i^{\mathsf{in}} = \mathsf{policy}$ for all $i \in \{2 \dots n\}$
   Check $\mathsf{at}_j^{\mathsf{out}} = \mathsf{at} \wedge \mathsf{policy}_j^{\mathsf{out}} = \mathsf{policy}$ for all $j \in \{2 \dots m\}$
   Check $b_{\mathsf{frz},i}^{\mathsf{in}} = b_{\mathsf{frz},j}^{\mathsf{out}} = 0$ for all $i \in [n]$, $j \in [m]$
   If check failed, abort by returning $\bot$
3. For $i \in [n]$:
   (a) Compute input ARCs: $\mathsf{arc}_i^{\mathsf{in}} = \mathsf{Com}(\boldsymbol{m}_i; \gamma_i^{\mathsf{in}})$ where $\boldsymbol{m}_i := (v_i^{\mathsf{in}}, \mathsf{at}_i^{\mathsf{in}}, \mathsf{upk}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}}, b_{\mathsf{frz},i}^{\mathsf{in}})$
   (b) Compute nullifier key:$\mathsf{nk}_i = \mathsf{derive}_{\mathsf{nk}}(\mathsf{usk}_i^{\mathsf{in}}, \mathsf{policy.fpk})$
   (c) Compute input nullifiers: $\mathsf{nl}_i^{\mathsf{in}} = \mathsf{Nullify}_{\mathsf{nk}_i}(\mathsf{elem}_i)$ where $\mathsf{elem}_i := (\mathsf{uid}_i^{\mathsf{in}}, \mathsf{arc}_i^{\mathsf{in}})$
4. For $j \in [m]$:
   (a) Compute output ARCs: $\mathsf{arc}_j^{\mathsf{out}} = \mathsf{Com}(\boldsymbol{m}_j; \gamma_j^{\mathsf{out}})$ where $\boldsymbol{m}_j := (v_j^{\mathsf{out}}, \mathsf{at}_j^{\mathsf{out}}, \mathsf{upk}_j^{\mathsf{out}}, \mathsf{policy}_i^{\mathsf{out}}, b_{\mathsf{frz},j}^{\mathsf{out}})$
   (b) If $j \neq 1$, prepare owner memos: $\mathsf{memo}_{\mathsf{own}_j} = \mathsf{Enc}(\mathsf{upk}_j^{\mathsf{out}}, \boldsymbol{m})$ where $\boldsymbol{m} := (v_j^{\mathsf{out}}, \mathsf{at}_j^{\mathsf{out}}, \mathsf{policy}_j^{\mathsf{out}}, b_{\mathsf{frz},j}^{\mathsf{out}}, \gamma_j^{\mathsf{out}})$
5. If $\mathsf{tpk} \neq \bot$ and $\mathsf{threshold} < \sum_{i \in [2..n]} v_i^{\mathsf{in}}$, prepare a tracer memo: $\mathsf{memo}_{\mathsf{trc}} = \mathsf{Enc}(\mathsf{policy.tpk}, \boldsymbol{m}; \gamma_{\mathsf{trc}})$
   where $\gamma_{\mathsf{trc}} \xleftarrow{\$} \mathbb{F}$ and
   $$\boldsymbol{m} = \left(\mathsf{at}, (b_0 \cdot \mathsf{upk}_i^{\mathsf{in}}, b_1 \cdot v_i^{\mathsf{in}}, b_2 \cdot \gamma_i^{\mathsf{in}}, \mathsf{attrmap} \odot \mathsf{attrs}_i)_{i \in \{2 \dots n\}}, (b_0 \cdot \mathsf{upk}_j^{\mathsf{out}}, b_1 \cdot v_j^{\mathsf{out}}, b_2 \cdot \gamma_j^{\mathsf{out}})_{j \in \{2 \dots m\}}\right),$$
   otherwise prepare a dummy tracer memo.
6. Set public instance: $x := (\mathsf{rt}, T, \mathsf{at}^*, \mathsf{fee}, (\mathsf{nl}_i^{\mathsf{in}})_{i \in [n]}, (\mathsf{arc}_j^{\mathsf{out}})_{j \in [m]}, \mathsf{memo}_{\mathsf{trc}}, \widehat{vk})$
   Set secret witness: $w := (\mathsf{at}, \mathsf{policy}, \mathsf{reveal}, \mathbf{usk}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{in}}, \mathbf{aux}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{out}}, \gamma_{\mathsf{trc}})$
7. Compute SNARK proof: $\pi_{\mathsf{xfr}} \leftarrow \mathcal{P}(\sigma_{\mathsf{xfr}}, x, w)$
8. Set $\mathsf{tx}_{\mathsf{axfr}} := (\mathsf{rt}, (\mathsf{nl}_i^{\mathsf{in}})_{i \in [n]}, (\mathsf{arc}_j^{\mathsf{out}})_{j \in [m]}, \pi_{\mathsf{xfr}}, \mathsf{memo}_{\mathsf{trc}}, \mathsf{fee}, \widehat{vk})$
9. Sign an aggregated list of owner memos: $\sigma_{\mathsf{memo}} = \mathsf{Sign}(\widehat{sk}, \boldsymbol{m})$ where $\boldsymbol{m} := (\mathsf{memo}_{\mathsf{own}_j})_{j \in \{2 \dots m\}}$
10. Output $(\mathsf{tx}_{\mathsf{axfr}}, (\mathsf{memo}_{\mathsf{own}_j})_{j \in \{2 \dots m\}}, \sigma_{\mathsf{memo}})$

$\underline{\mathsf{Freeze}_n(\mathsf{pp}, \mathbf{fsk}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{in}}, \mathbf{aux}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{out}}, \mathsf{rt}) \rightarrow (\mathsf{tx}_{\mathsf{frz}}, \mathbf{memo}_{\mathbf{own}}, \sigma_{\mathsf{memo}})}$

- *inputs*:
  - public parameter: $\mathsf{pp} := (\mathsf{at}^*, \sigma_{\mathsf{frz}}, \dots)$
  - n freezer secret keys: $\mathbf{fsk}_{\mathsf{in}} := (\mathsf{fsk}_i^{\mathsf{in}})_{i \in [n]}$
  - n openings of input ARC: $\mathbf{oar}_{\mathsf{in}} := (v_i^{\mathsf{in}}, \mathsf{at}_i^{\mathsf{in}}, \mathsf{upk}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}}, b_{\mathsf{frz},i}^{\mathsf{in}}; \gamma_i^{\mathsf{in}})_{i \in [n]}$
  - n auxiliary info about input ARC: $\mathbf{aux}_{\mathsf{in}} := (\mathsf{uid}_i^{\mathsf{in}}, \pi_i^{\mathsf{in}})_{i \in [n]}$ where
    $\mathsf{uid}^{\mathsf{in}}$ is the unique identifier of input ARC inside the accumulator
    π_i^{in} is an ARC membership proof
  - n openings of output ARC: $\mathbf{oar}_{\mathsf{out}} := (v_j^{\mathsf{out}}, \mathsf{at}_j^{\mathsf{out}}, \mathsf{upk}_j^{\mathsf{out}}, \mathsf{policy}_j^{\mathsf{out}}, b_{\mathsf{frz},j}^{\mathsf{out}}; \gamma_j^{\mathsf{out}})_{j \in [n]}$
  - Merkle accumulator digest: rt
- *outputs*:
  - freezing transaction body: $\mathsf{tx}_{\mathsf{frz}}$
  - list of owner memos (one for each output record except for the 1st output as fee change): $\mathbf{memo}_{\mathbf{own}}$
  - signature over all owner memos under $\widehat{vk}$: σ_{memo}

1. Set $\mathsf{fee} = v_1^{\mathsf{in}} - v_1^{\mathsf{out}}$
2. Check $\mathsf{at}_1^{\mathsf{in}} = \mathsf{at}_1^{\mathsf{out}} = \mathsf{at}^* \wedge \mathsf{policy}_1^{\mathsf{in}} = \mathsf{policy}_1^{\mathsf{out}} = \bot$
   Check $b_{\mathsf{frz},1}^{\mathsf{in}} = b_{\mathsf{frz},1}^{\mathsf{out}} = 0$
   Check $b_{\mathsf{frz},i}^{\mathsf{in}} = 0 \wedge \mathsf{policy}_i^{\mathsf{in}}.\mathsf{fpk} \neq \bot$ for all $i \in \{2 \dots n\}$
   Check $(v_i^{\mathsf{in}}, \mathsf{at}_i^{\mathsf{in}}, \mathsf{upk}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}}) = (v_i^{\mathsf{out}}, \mathsf{at}_i^{\mathsf{out}}, \mathsf{upk}_i^{\mathsf{out}}, \mathsf{policy}_i^{\mathsf{out}})$ for all $i \in \{2 \dots n\}$
   Check $b_{\mathsf{frz},j}^{\mathsf{out}} = 1$ for all $j \in \{2 \dots n\}$
   If check failed, abort by returning $\bot$

[6] Currently only support single asset transfer and "per asset type policy", thus apart from the first fixed slot for fee change, the second output should share the same asset type and asset policy (thus the same reveal map) for the rest of the output records.

3. For $i \in [n]$:
   (a) Compute input ARCs: $\mathsf{arc}_i^{\mathsf{in}} = \mathsf{Com}(\boldsymbol{m}_i; \gamma_i^{\mathsf{in}})$ where $\boldsymbol{m}_i := (v_i^{\mathsf{in}}, \mathsf{at}_i^{\mathsf{in}}, \mathsf{upk}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}}, b_{\mathsf{frz},i}^{\mathsf{in}})$
   (b) Compute nullifier key:$\mathsf{nk}_i = \mathsf{derive}_{\mathsf{nk}}(\mathsf{fsk}_i^{\mathsf{in}}, \mathsf{upk}_i^{\mathsf{in}})$
   (c) Compute input nullifiers: $\mathsf{nl}_i^{\mathsf{in}} = \mathsf{Nullify}_{\mathsf{nk}_i}(\mathsf{elem}_i)$ where $\mathsf{elem}_i := (\mathsf{uid}_i^{\mathsf{in}}, \mathsf{arc}_i^{\mathsf{in}})$
4. For $j \in [n]$:
   (a) Compute output ARCs: $\mathsf{arc}_j^{\mathsf{out}} = \mathsf{Com}(\boldsymbol{m}_j; \gamma_j^{\mathsf{out}})$ where $\boldsymbol{m}_j := (v_j^{\mathsf{out}}, \mathsf{at}_j^{\mathsf{out}}, \mathsf{upk}_j^{\mathsf{out}}, \mathsf{policy}_i^{\mathsf{out}}, b_{\mathsf{frz},j}^{\mathsf{out}})$
   (b) If $j \neq 1$, prepare owner memos: $\mathsf{memo}_{\mathsf{own}_j} = \mathsf{Enc}(\mathsf{upk}_j^{\mathsf{out}}, \boldsymbol{m})$ where $\boldsymbol{m} := (v_j^{\mathsf{out}}, \mathsf{at}_j^{\mathsf{out}}, \mathsf{policy}_j^{\mathsf{out}}, b_{\mathsf{frz},j}^{\mathsf{out}}, \gamma_j^{\mathsf{out}})$
5. Set public instance: $x := (\mathsf{rt}, \mathsf{at}^*, \mathsf{fee}, (\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in[n]})$
   Set secret witness: $w := (\mathbf{fsk}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{in}}, \mathbf{aux}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{out}})$
6. Compute SNARK proof: $\pi_{\mathsf{frz}} \leftarrow \mathcal{P}(\sigma_{\mathsf{frz}}, x, w)$
7. Sign an aggregated list of owner memos: $\sigma_{\mathsf{memo}} = \mathsf{Sign}(\widehat{sk}, \boldsymbol{m})$ where $\boldsymbol{m} := (\mathsf{memo}_{\mathsf{own}_j})_{j\in\{2\ldots n\}}$
8. Output $(\mathsf{tx}_{\mathsf{frz}} := (\mathsf{rt}, (\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in[n]}, \pi_{\mathsf{frz}}, \mathsf{fee}), \mathbf{memo}_{\mathbf{own}}, \sigma_{\mathsf{memo}})$

$\underline{\mathsf{Unfreeze}_n(\mathsf{pp}, \mathbf{fsk}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{in}}, \mathbf{aux}_{\mathsf{in}}, \mathbf{oar}_{\mathsf{out}}, \mathsf{rt}) \to (\mathsf{tx}_{\mathsf{frz}}, \mathbf{memo}_{\mathbf{own}}, \sigma_{\mathsf{memo}})}$

- *inputs*: identical with Freeze above
- *outputs*:
  - Unfreezing transaction body: $\mathsf{tx}_{\mathsf{frz}}$
  - list of owner memos (one for each output record except for the 1st output as fee change): $\mathbf{memo}_{\mathbf{own}}$
  - signature over all owner memos under $\widehat{vk}$: σ_{memo}

1. Set $\mathsf{fee} = v_1^{\mathsf{in}} - v_1^{\mathsf{out}}$
2. Check $\mathsf{at}_1^{\mathsf{in}} = \mathsf{at}_1^{\mathsf{out}} = \mathsf{at}^* \wedge \mathsf{policy}_1^{\mathsf{in}} = \mathsf{policy}_1^{\mathsf{out}} = \bot$
   Check $b_{\mathsf{frz},1}^{\mathsf{in}} = b_{\mathsf{frz},1}^{\mathsf{out}} = 0$
   Check $b_{\mathsf{frz},i}^{\mathsf{in}} = 1 \wedge \mathsf{policy}_i^{\mathsf{in}}.\mathsf{fpk} \neq \bot$ for all $i \in \{2 \ldots n\}$
   Check $(v_i^{\mathsf{in}}, \mathsf{at}_i^{\mathsf{in}}, \mathsf{upk}_i^{\mathsf{in}}, \mathsf{policy}_i^{\mathsf{in}}) = (v_i^{\mathsf{out}}, \mathsf{at}_i^{\mathsf{out}}, \mathsf{upk}_i^{\mathsf{out}}, \mathsf{policy}_i^{\mathsf{out}})$ for all $i \in \{2 \ldots n\}$
   Check $b_{\mathsf{frz},j}^{\mathsf{out}} = 0$ for all $j \in \{2 \ldots n\}$
   If check failed, abort by returning $\bot$
3. Step 3∼7 are exactly the same as that of Freeze

$\underline{\mathsf{VfyTx}(\mathsf{pp}, \mathsf{tx}, \mathsf{info}_L) \to b}$

- *inputs*:
  - public parameter: $\mathsf{pp} := (\mathsf{at}^*, \tau_{\mathsf{mint}}, \tau_{\mathsf{xfr}}, \tau_{\mathsf{frz}}, \ldots)$
  - a transaction received: tx
  - current ledger state: $\mathsf{info}_L := (\mathsf{rt}, T)$
- *outputs*: accepting bit $b \in \{0, 1\}$

1. If tx is a $\mathsf{tx}_{\mathsf{mint}} := (\mathsf{rt}, \mathsf{nl}_{\mathsf{fee}}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in\{1,2\}}, (v, \mathsf{at}, \mathsf{policy}), \pi_{\mathsf{mint}}, \mathsf{memo}_{\mathsf{trc}}, \mathsf{aux} = (\mathsf{fee}, pk_{\mathsf{sig}}))$:
   (a) If policy is invalid (i.e., the public keys are not well-formed), output 0 and halt.
   (b) Prepare public instance: $x := (\mathsf{rt}, \mathsf{at}^*, \mathsf{policy}^*, v, \mathsf{at}, \mathsf{policy}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in\{1,2\}}, \mathsf{fee}, \mathsf{nl}_{\mathsf{fee}}, \mathsf{memo}_{\mathsf{trc}})$
   (c) Set $\pi = \pi_{\mathsf{mint}}, \tau = \tau_{\mathsf{mint}}$
2. If tx is a $\mathsf{tx}_{\mathsf{axfr}} := (\mathsf{rt}, (\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in[m]}, \pi_{\mathsf{xfr}}, \mathsf{memo}_{\mathsf{trc}}, \mathsf{aux} = (\mathsf{fee}, pk_{\mathsf{sig}}))$:
   (a) Prepare public instance: $x := (\mathsf{rt}, T, \mathsf{at}^*, \mathsf{policy}^*, \mathsf{fee}, (\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in[m]}, \mathsf{memo}_{\mathsf{trc}})$
   (b) Set $\pi = \pi_{\mathsf{xfr}}, \tau = \tau_{\mathsf{xfr}}$
3. If tx is a $\mathsf{tx}_{\mathsf{frz}} := (\mathsf{rt}, (\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in[n]}, \pi_{\mathsf{frz}}, \mathsf{aux} = (\mathsf{fee}, pk_{\mathsf{sig}}))$:
   (a) Prepare public instance: $x := (\mathsf{rt}, \mathsf{at}^*, \mathsf{fee}, (\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in[n]})$
   (b) Set $\pi = \pi_{\mathsf{frz}}, \tau = \tau_{\mathsf{frz}}$
4. Output $b = \mathcal{V}(\tau, x, \pi)$

Remark 5 (Policy Checks). For an asset issuance transaction, the validator should check that the issued asset policy is valid and well-formed. More precisely, each public key pk in the policy should either be $\bot$ (i.e. $\mathsf{pk} = \infty$) or a non-neutral point in a large subgroup. These checks are necessary to prevent users from misusing public keys and leaking private information. Since we are working in the Jubjub curve which only has two subgroups (i.e., the cofactor-order subgroup and the Jubjub subgroup), it is sufficient to check that the public key pk is on the curve (i.e. satisfies curve equation) but not in the order-8 cofactor subgroup (i.e. $8 \cdot \mathsf{pk} \neq \infty$ when $\mathsf{pk} \neq \infty$).

$\underline{\mathsf{Receive}(\mathsf{pp}, \mathsf{upk}, \mathsf{usk}, L) \to (\mathsf{arc}_i, \mathsf{oar}_i)_{i\in\ell}}$

- *inputs*:
  - public parameter: $\mathsf{pp} := (\ldots)$
  - recipient key pair: $\mathsf{upk}, \mathsf{usk}$
  - ledger state: $L := (\mathsf{Acc}_{\mathsf{arc}}, \{\mathsf{nl}\}, \{\mathsf{tx}\}, \ldots)$
- *outputs*: list of unspent ARCs received by upk: $(\mathsf{arc}_i, \mathsf{oar}_i)_{i\in\ell}$

1. Initialize an empty set: $S = \emptyset$
2. For every $\mathsf{tx}_i \in \{\mathsf{tx}\}$:
   (a) Extract $(\mathsf{memo}_{\mathsf{own}_j})_{j\in[m]}$ that corresponds to tx_i via the public bulletin board or a private channel.
   (b) For every $\mathsf{memo}_j \in \left\{\mathsf{memo}_{\mathsf{own}_j}\right\}_{j\in[m]}$:
      i. Compute $m \leftarrow \mathsf{Dec}(\mathsf{usk}, \mathsf{memo}_j)$
      if $m \neq \perp$, then parse $m := (v, \mathsf{at}, \mathsf{policy}, b_{\mathsf{frz}}, \gamma)$; else continues to next iteration
      ii. Prepare $\mathsf{oar} = (v, \mathsf{at}, \mathsf{upk}, \mathsf{policy}, b_{\mathsf{frz}}, \gamma)$
      Compute $\mathsf{arc} = \mathsf{Com}(\boldsymbol{m}; \gamma)$ where $\boldsymbol{m} := (v, \mathsf{at}, \mathsf{upk}, \mathsf{policy}, b_{\mathsf{frz}})$
      iii. Retrieve from accumulator $\mathsf{elem} \leftarrow \mathsf{Acc}_{\mathsf{arc}}(\mathsf{arc})$
      If $\mathsf{policy.fpk} = \perp$, Compute $\mathsf{nk} = \mathsf{derive}_{\mathsf{nk}}(\mathsf{usk}, \infty)$ where $\infty = (0, 1)$ is the neutral point
      If $\mathsf{policy.fpk} \neq \perp$, Compute $\mathsf{nk} = \mathsf{derive}_{\mathsf{nk}}(\mathsf{usk}, \mathsf{policy.fpk})$
      Compute $\mathsf{nl} = \mathsf{Nullify}_{\mathsf{nk}}(\mathsf{elem})$
      Check $\mathsf{nl} \notin \{\mathsf{nl}\}$, continues to next iteration if failed
      iv. Append $(\mathsf{arc}, \mathsf{oar})$ to S
3. Output $S := (\mathsf{arc}_i, \mathsf{oar}_i)_{i\in[\ell]}$

$\underline{\mathsf{Trace}(\mathsf{pp}, \mathsf{tpk}, \mathsf{tsk}, \mathsf{tx}) \rightarrow \mathsf{list}_{\mathsf{trc}}}$

- *inputs*:
  - public parameter: $\mathsf{pp} := (\ldots)$
  - tracer key pair: $\mathsf{tpk}, \mathsf{tsk}$
  - a transaction: tx
- *outputs*: list of traced info in tx: $\mathsf{list}_{\mathsf{trc}}$

1. If tx is a $\mathsf{tx}_{\mathsf{mint}} := (\mathsf{rt}, \mathsf{nl}_{\mathsf{fee}}, (\mathsf{arc}_j^{\mathsf{out}})_{j\in\{1,2\}}, (v, \mathsf{at}), \pi_{\mathsf{mint}}, \mathsf{aux} = (\mathsf{fee}, pk_{\mathsf{sig}}))$:
   (a) Set $\mathsf{list}_{\mathsf{trc}} = (\texttt{"mint"}, v, \mathsf{at})$
2. If tx is a $\mathsf{tx}_{\mathsf{axfr}} := (\mathsf{rt}, (\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}, (\mathsf{arc}_j^{\mathsf{out}},)_{j\in[m]}, \pi_{\mathsf{xfr}}, \mathsf{memo}_{\mathsf{trc}}, \mathsf{aux} = (\mathsf{fee}, pk_{\mathsf{sig}}))$:
   (a) Compute $m \leftarrow \mathsf{Dec}(\mathsf{tsk}, \mathsf{memo}_{\mathsf{trc}})$
   (b) Parse $\mathsf{list}_{\mathsf{trc}} = \left\{\mathsf{at}, (v_i, \mathsf{upk}_i, \mathsf{policy}_i, \mathsf{attrs}_i)_{i\in\{2\ldots n\}}, (v_j, \mathsf{upk}_j)_{j\in\ldots\{2\ldots m\}}\right\} \leftarrow m$
3. Output $\mathsf{list}_{\mathsf{trc}}$

3.4 Fee collection

CollectFee workflow is outside the CAP scheme, but part of the system API – specifically, it describe how a validator collects its fees when processing a block of transactions. The validator who builds the next block containing all pending, verified transactions is allowed to create a new asset record of the native asset type and of value equal to the sum of all fee specified in each transactions within that block. Since the entire opening of the ARC will be published, the integrity of the fee collection AR can/will be checked publicly before being accepted into the ARC accumulator with consensus. Note that anyone (including other validators) can derive the commitment (ARC) of the fee collection AR as even the blind factor is revealed in the open, but only the validator with the secret key can spend it in the future; furthermore, the spending of such ARC is no different than any other ARC as they all are anonymous and untraceable.

$\underline{\mathsf{CollectFee}(\mathsf{pp}, \mathbf{tx}, \mathsf{upk}, L) \leftarrow (L', \mathsf{arc}, \mathsf{oar})}$

- *inputs*:
  - public parameter: $\mathsf{pp} := (\mathsf{at}^*, \tau_{\mathsf{mint}}, \tau_{\mathsf{xfr}}, \tau_{\mathsf{frz}}, \ldots)$
  - a list of transactions validated: $\mathbf{tx}$
  - validator public key: upk
  - ledger state: $L := (\mathsf{Acc}_{\mathsf{arc}}, \{\mathsf{nl}\}, \{\mathsf{tx}\}, \ldots)$
- *outputs*:
  - updated ledger state: L'
  - an ARC credited to validator: arc
  - openings of arc: $\mathsf{oar} := (v, \mathsf{at}^*, \mathsf{upk}, b_{\mathsf{frz}}, \mathsf{policy}^*, \gamma)$

1. Set $v = 0, L' = L$
2. For all $\mathsf{tx}_i \in \mathbf{tx}$:
   - Verify transaction: $b = \mathsf{VfyTx}(\mathsf{pp}, \mathsf{tx}, L)$
   - If $b = 1$
     (a) Update nullifier set: $L'.\{\mathsf{nl}\} = L'.\{\mathsf{nl}\} \cup \mathsf{tx}.(\mathsf{nl}_i^{\mathsf{in}})_{i\in[n]}$
     (b) Update ARC accumulators: for $j \in [m]$, $L'.\mathsf{Acc.add}(L'.\mathsf{rt}, \mathsf{elem}_j)$ where elem_j contains the output ARC $\mathsf{tx}.\mathsf{arc}_j^{\mathsf{out}}$
     (c) Collect fee: $v = v + \mathsf{tx}_i.\mathsf{fee}$
   - If $b = 0$, abort by returning $\perp$
3. Sample random blind factor: $\gamma \xleftarrow{\$} \mathbb{F}_l$
   Set $\mathsf{oar} := (v, \mathsf{at}^*, \mathsf{upk}, b_{\mathsf{frz}} = 0, \mathsf{policy}^*, \gamma)$
4. Create an ARC for all fees collected: $\mathsf{arc} := \mathsf{Com}(\mathsf{oar}, \gamma)$
5. Add the fee collection ARC into accumulator: $L'.\mathsf{Acc.add}(L'.\mathsf{rt}, \mathsf{elem})$ where $\mathsf{elem} := (\mathsf{uid}_{\mathsf{arc}}, \mathsf{arc})$ and $\mathsf{uid}_{\mathsf{arc}} = L'.\mathsf{Acc.size}()$
6. Output $(L', \mathsf{arc}, \mathsf{oar})$

4 Instantiation of CAP

4.1 Cryptographic Primitives

4.1.1 Elliptic Curve

In the following we will use this notation for the parameters of a curve.

- q: the size of the field.
- a, b, d: coefficients involved in the equation of the elliptic curve.
- $B = (x, y)$: base element.
- n: order of the generator.
- h: cofactor.

BLS12-381 Our pairing based schemes use the BLS12-381 curve[Bow17]. The pairing is defined over three groups $G_1 \times G_2 \to G_T$. G_T is the field $F_{q^{12}}$. In this section i denotes the complex number such that $i^2 = -1$. The details of the curve parameters can be found in [Gri18].

Equation for G_1	$E(F_q) = y^2 \equiv x^3 + ax + b$ (Weierstrass)
q	0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaaab
a	0
b	4
B	x=0x17f1d3a73197d7942695638c4fa9ac0fc3688c4f9774b905a14e3a3f171bac586c55e83ff97a1aeffb3af00adb22c6bb y=0x08b3f481e3aaa0f1a09e30ed741d8ae4fcf5e095d5d00af600db18cb2c04b3edd03cc744a2888ae40caa232946c5e7e1
n	0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff0000001
h	0x396c8c005555e1568c00aaab0000aaab

Equation for G_2	$E(F_{q^2}) = y^2 \equiv x^3 + ax + b$ (Weierstrass)
q	0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaaab
a	0
b	$4i + 1$
B	x=0x24aa2b2f08f0a91260805272dc51051c6e47ad4fa403b02b4510b647ae3d1770bac0326a805bbefd48056c8c121bdb8+i·0x13e02b6052719f607dacd3a088274f65596bd0d09920b61ab5da61bbdc7f5049334cf11213945d57e5ac7d055d042b7e y=0xce5d527727d6e118cc9cdc6da2e351aadfd9baa8cbdd3a76d429a695160d12c923ac9cc3baca289e193548608b82801+i·0x606c4a02ea734cc32acd2b02bc28b99cb3e287e85a763af267492ab572e99ab3f370d275cec1da1aaa9075ff05f79be
n	0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff0000001
h	0x5d543a95414e7f1091d50792876a202cd91de4547085abaa68a205b2e5a7ddfa628f1cb4d9e82ef21537e293a6691ae1616ec6e786f0c70cf1c38e31c7238e5

Jubjub Jubjub[ZCa17] is an elliptic curve built on top of the scalar field of the BLS12-381 curve 4.1.1. Its security has been analyzed by Daira Hopwood [Hop17].

Equation	$ax^2 + y^2 \equiv 1 + dx^2y^2$ (Twisted Edwards)
q	0x73eda753299d7d483339d80809a1d80553bda402fffe5bfefffffffff00000001
a	0x73eda753299d7d483339d80809a1d80553bda402fffe5bfefffffffff00000000
d	0x2a9318e74bfa2b48f5fd9207e6bd7fd4292d7f6d37579d2601065fd6d6343eb1
B	x=0x11dafe5d23e1218086a365b99fbf3d3be72f6afd7d1f72623e6b071492d1122b y=0x1d523cf1ddab1a1793132e78c866c0c33e26ba5cc220fed7cc3f870e59d292aa
n	0xe7db4ea6533afa906673b0101343b00a6682093ccc81082d0970e5ed6f72cb7
h	0x08

4.1.2 Digital Signatures

DSA signature over jubjub. We propose a DSA based signature over the Jubjub group using Rescue as the underlying hash function. Our implementation variant (see pseudo-code in Algorithm 1) has the following characteristics:

- The group G is the Jubjub group (see Section 4.1.1). Recall that the order of this group is a $8p$ for a prime p.
- The hash function H used is Rescue Sponge with padding. The reason for using this group and hash function is that the the SNARG circuit (see Section 4.2.5) checks the validity of signatures, specifically for user's credential validation.
- The scheme is deterministic, that is the scalar r is computed as $r := H(\texttt{alg\_desc}, m, \texttt{sk})$ where m is the message and sk is the private key. This is to avoid attacks based on bad implementation of pseudo-random generators. Note that the algorithm description alg_desc is passed to H. This is to prevent attacks where the same private key is used with different algorithms (e.g. ECDSA and Schnorr).

Based on [CGN20], our DSA-over-jubjub signature is a Schnorr scheme in which the verification function checks that the public key is not a low order point.

Algorithm 1 DSA over Jubjub Signature

$\texttt{KeyGen}(1^\lambda)$	$\texttt{Sign}(m, x)$	$\texttt{Verify}(m, X, sig)$
$x \leftarrow_\$ \mathbb{Z}_q$	$r \leftarrow H(\texttt{alg\_desc}, m, x)$	Fail if $8X = 0$
$X \leftarrow xB$	$R \leftarrow rB$	Parse sig as (R, s)
return (x, X)	$c \leftarrow H(X, R, m)$	$c \leftarrow H(X, R, m)$
	$s \leftarrow r + cx$	Succeed if $R + cX = sB$
	return (R, s)	

Encodings. The verification function above assumes that the public key and signature are already deserialized into Jubjub objects. It is important that this deserialization is *cannonical*. That is, no two different byte arrays can be deserialized into one single jubjub scalar (or group element). Scalars elements are serialized as integers in little-endian byte array representation. At deserialization, we reject integers larger or equal than the jubjub scalar field. For group elements curve points (x, y), x is serialized as an integer in little-endian byte representation together with the sign of y. At deserialization, we reject if x is an integer larger that the jubjub *base* field or if y cannot be derived (there is no point having that x coordinate).

4.1.3 Pseudorandom Permutation (PRP)

Rescue We use Rescue[AABS+19] PRP to implement sponge based PRF (see Section A) and collision-resistant hash function (see Section 4.1.4).

Our Rescue instance works over the BLS12_381 scalar field, with keys and inputs of size 4 field elements. The parameters of the Rescue permutation were obtained using the script from https://github.com/KULeuven-COSIC/Marvellous (revision: 1bad94a239dd52f7206cd7f2d3a7f023e006f533) as shown in listing 1.1:

Listing 1.1. Generating Rescue parameters

```
> sage
> sage: load("instance_generator.sage")
> sage: q=0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001
> sage: instance = Rescue(128, q, 4, 5)
```

The Rescue pseudorandom permutation[AABS+19] (see pseudo-code in Algorithm 2) is defined by a square matrix **MDS** of size $\mathtt{w} \times \mathtt{w}$ (in our instantiation $\mathtt{w} = 4$), an initial constants vector **IC**, and a key-scheduling constant vector **C** and a key-scheduling matrix **K**. We set the number of rounds $\mathtt{n_r} = 12$ and $\alpha = 5$. Note that during key scheduling, the key injection vectors can be preprocessed yielding a much faster generation of round keys.

Algorithm 2 Rescue Pseudorandom permutation.

`LinearOp`$(\boldsymbol{M}, \boldsymbol{v}, \boldsymbol{c})$

/* $\boldsymbol{M}$ is a $\mathtt{w} \times \mathtt{w}$ matrix */
/* $\boldsymbol{v}$ and $\boldsymbol{c}$ are vector of length $\mathtt{w}$ */
return $\boldsymbol{M} \cdot \boldsymbol{v} + \boldsymbol{c}$

`KeyInjection`()

$\boldsymbol{prev} \leftarrow \mathbf{IC}$
$\boldsymbol{key\_inj} \leftarrow [\boldsymbol{prev}]$
for $i \in [0..2 * \mathtt{n_r}]$ **do**
  $\boldsymbol{prev} \leftarrow \mathtt{LinearOp}(\mathbf{K}, \boldsymbol{key\_inj}, \mathbf{C})$
  $\boldsymbol{key\_inj}.push(\boldsymbol{prev})$
endfor
return $\boldsymbol{key\_inj}$

`KeyScheduling`$(\boldsymbol{k})$

/* $\boldsymbol{k}$ is a vector of length $\mathtt{w}$ */
/* Returns round keys matrix of size $(2\mathtt{n_r} + 1) \times \mathtt{w}$ */
$\boldsymbol{key\_inj} \leftarrow \mathtt{KeyInjection}(\mathtt{n_r})$
$\boldsymbol{prev\_key} \leftarrow \boldsymbol{k} + \boldsymbol{key\_inj}[0]$
$\boldsymbol{keys} \leftarrow [\boldsymbol{prev\_key}]$
for $i \in [0..2 * \mathtt{n_r}[$ **do**
  if $i\%2 == 0$
    $\boldsymbol{prev\_key} \leftarrow \boldsymbol{prev\_key}^{1/\alpha}$
  else
    $\boldsymbol{prev\_key} \leftarrow \boldsymbol{prev\_key}^{\alpha}$
  endif
  $\boldsymbol{prev\_key} \leftarrow \mathtt{LinearOp}(\mathbf{MDS}, \boldsymbol{prev\_key}, \boldsymbol{key\_inj}[i+1])$
  $\boldsymbol{keys}.push(\boldsymbol{prev\_key})$
endfor
return $\boldsymbol{keys}$

`RescueWithRoundKeys`$(\boldsymbol{m}, \boldsymbol{k})$

/* Preconditions */
$\mathtt{assert}(len(\boldsymbol{k}), 2\mathtt{n_r} + 1)$
$\mathtt{assert}(len(\mathbf{MDS}), \mathtt{w})$
$\mathtt{assert}(len(\boldsymbol{m}), \mathtt{w})$
$\boldsymbol{S} \leftarrow \boldsymbol{m} + \boldsymbol{k}[0]$
for $i \in [1..len(\boldsymbol{k})[$ **do**
  if $(i-1)\%2 == 0$
    $\boldsymbol{S} = \boldsymbol{S}^{1/\alpha}$
  else
    $\boldsymbol{S} = \boldsymbol{S}^{\alpha}$
  endif
  $\boldsymbol{S} \leftarrow \mathtt{LinearOp}(\mathbf{MDS}, \boldsymbol{S}, \boldsymbol{k}[i])$
endfor
return $\boldsymbol{S}$

`RescuePseudoRandomPermutation`$(\boldsymbol{m}, \boldsymbol{k})$

$\boldsymbol{k'} \leftarrow \mathtt{KeyScheduling}(\boldsymbol{k})$
return $\mathtt{RescueWithRoundKeys}(\boldsymbol{m}, \boldsymbol{k'})$

A fixed-key Permutation Our construction of hash functions and pseudo-random functions use a fixed-key permutation rather than the Rescue PRP. We build it by setting the key to the 0 vector as in Algorithm 3.

Algorithm 3 Permutation based on Rescue PRP

`RescuePermutation`$(\boldsymbol{m})$

$\boldsymbol{key} \leftarrow [0; \mathtt{w}]$
return $\mathtt{RescuePseudoRandomPermutation}(\boldsymbol{m}, \boldsymbol{key})$

We stress that the round keys can be preprocessed by calling key scheduling on the zero-vector key, hence we only need to call `RescueWithRoundKeys` online, leading to a faster algorithm for computing the permutation.

4.1.4 Collision-resistant Hash Function

Sha(512). We use the Sha512 hash function [NIS15]. Test vectors can be found at https://www.cosic.esat.kuleuven.be/nessie/testvectors/hash/sha/Sha-2-512.unverified.test-vectors.

Rescue Sponge-based CRHF. Given the Rescue permutation described in Section 4.1.3, we can build a collision-resistant hash function using the Sponge construction [BDPVA07]. In our instantiation of Rescue, the permutation state is of width 4: 3 slots for the *rate* and 1 for the *capacity* of the sponge construction.

We provide two instantiations of the Sponge based CRHF. The first one assumes the input length is multiple of the *rate*. The second one applies the following simple padding before calling the Sponge CRHF: append the field element 1 to the input, then append zeroes as necessary until length is multiple of *rate*. Algorithm is presented in Figure 4.

Algorithm 4 Sponge CRHF over Rescue permutation

`sponge_no_padding`($\boldsymbol{m}, rate, capacity,$ `num_output`)

```
/* m: input to be hashed.The size of this input must be a multiple of the rate. */
/* rate: rate of the Rescue permutation. */
/* capacity: capacity of the Rescue permutation. */
/* returns: hash of input of length num_output */
l ← len(input)
if l % rate != 0
  abort
endif
if rate + capacity! = w
  abort
endif
state ← [0; w]
/* Absorb phase: */
for i ∈ [0..l/rate]
  block ← m[rate * i..rate * (i + 1) − 1]
  padded_block ← PadBlock(block)
  state = state + padded_block
  state ← RescuePermutation(state)
endfor
/* Squeeze phase: */
m ← ceil(num_output/rate)
output ← ∅
for i ∈ [0..m − 1]
  output.append(state[0..rate])
  num_output = num_output − rate
  state ← RescuePermutation(state)
endfor
output.append(state[0..num_output]
return output
```

`PadBlock`($block$)

```
while len(block) < w
  block.append(0)
endwhile
return block
```

`sponge`($\boldsymbol{m}, rate, capacity,$ `num_output`)

```
/* rate: rate of the Rescue permutation. */
/* capacity: capacity of the Rescue permutation. */
/* returns: hash of input of length num_output */
m.append(1)
while len(m)%rate! = 0
  m.append(0)
endwhile
output ← sponge_no_padding(m, rate, capacity, num_output)
return output
```

4.1.5 Pseudorandom Function (PRF) We implement a sponge-based PRF from the Rescue fixed permutation. The construction follows the Full-State Keyed Sponge (FKS) paradigm (see Algorithm 1 in [MRV15]) but here is simplified to output a single field element.

The PRF takes a secret key k of one field element, a message $\boldsymbol{m}$ of fixed, yet arbitrary length.

The Full-State Keyed Sponge construction works as follows: it set the initial state with zeroes and the key in the last slot. Then it divides the input in chunks of Rescue's state size, and absorb them sequentially by 1) adding the chunk to the state, and 2) calling the Rescue permutation to produce a new state. After the input has been absorbed, it outputs the first element of the state[7].

The pseudo-code of the single output PRF can be found in Algorithm 5.

[7] For arbitrary length output, the squeeze phase proceeds as in a sponge construction: the rate part of the state is outputed, then the permutation is applied to the state to produce more output chunks until desired output length is achieved

Algorithm 5 PRF based on Full-State-Keyed-Sponge over Rescue

$\texttt{PRF}_\texttt{n}(k, \boldsymbol{input})$

$assert\_equal(len(\boldsymbol{input}), n)$
/* Pad input with zeroes */
while $len(\boldsymbol{input})\%\texttt{w} \neq 0$
  $\boldsymbol{input}.push(0)$
endwhile
$\boldsymbol{state} \leftarrow [0, \ldots, 0, k]$
for $i \in [0..len(\boldsymbol{input})/\texttt{w})[$
  $\boldsymbol{state} \leftarrow \boldsymbol{state} + m[i\texttt{w} \,..\, (i+1)\texttt{w}]$
  $\boldsymbol{state} \leftarrow \texttt{RescuePermutation}(\boldsymbol{state})$
endfor
return $\boldsymbol{state}[0]$

Algorithm 6 Counter mode rescue

$\texttt{ApplyKeyStream}(\boldsymbol{k}, \boldsymbol{data}, is\_add)$

/* $\boldsymbol{data}$: input to be processed */
/* $\boldsymbol{k}$: keys */
/* is_add: boolean. True for *encryption*, False for *decryption* */
/* encryption or decryption of $\boldsymbol{data}$ */
$StateSize \leftarrow 4$
$nonce \leftarrow 0$
$\texttt{n}_\texttt{r} \leftarrow 12$
$\boldsymbol{roundKeys} \leftarrow \texttt{KeyScheduling}(\boldsymbol{k}, \texttt{n}_\texttt{r})$
/* Process the data in blocks of StateSize */
for $i \in [0..len(\boldsymbol{data})/StateSize - 1]$
  $\boldsymbol{block} \leftarrow data[StateSize * i..StateSize * (i+1) - 1]$
  $\boldsymbol{input} \leftarrow [nonce]$
  for $i \in [1..StateSize - 1]$
    $\boldsymbol{input}[i] \leftarrow 0$
  endfor
  $\boldsymbol{keyStream} \leftarrow \texttt{RescueWithRoundKeys}(\boldsymbol{input}, \boldsymbol{roundKeys})$
  $l \leftarrow len(block)$
  if is_add
    $\boldsymbol{block} \leftarrow \boldsymbol{block} + \boldsymbol{keyStream}[0..l-1]$
  else
    $\boldsymbol{block} \leftarrow \boldsymbol{block} - \boldsymbol{keyStream}[0..l-1]$
  endif
  $nonce \leftarrow nonce + 1$
endfor return $\boldsymbol{data}$

$\mathsf{rescue}^{\texttt{gen}}_{\texttt{ctr}}(1^\lambda)$

$\boldsymbol{k} \leftarrow_\$ \mathbb{F}_p^4$
return $\boldsymbol{k}$

$\mathsf{rescue}^{\texttt{enc}}_{\texttt{ctr}}(\boldsymbol{k}, \boldsymbol{m})$

$\boldsymbol{c} \leftarrow \texttt{ApplyKeyStream}(\boldsymbol{k}, \boldsymbol{m}, true)$
return $\boldsymbol{c}$

$\mathsf{rescue}^{\texttt{dec}}_{\texttt{ctr}}(\boldsymbol{k}, \boldsymbol{c})$

$\boldsymbol{m} \leftarrow \texttt{ApplyKeyStream}(\boldsymbol{k}, \boldsymbol{c}, false)$
return $\boldsymbol{m}$

4.1.6 Stream cipher

Algorithm 7 El-Gamal Hybrid Public Key Encryption Scheme

$\texttt{KeyGen}(1^\lambda)$

$\mathsf{sk} \leftarrow_\$ \mathbb{F}_p$
$\mathsf{PK} \leftarrow \mathsf{sk}B$
return $(\mathsf{sk}, \mathsf{PK})$

$\texttt{Encrypt}(\mathsf{PK}, \boldsymbol{m})$

$r \leftarrow_\$ \mathbb{F}_p$
$R \leftarrow rB$
$U = (x_U, y_U) \leftarrow r\mathsf{PK}$
$\boldsymbol{k} \leftarrow \mathsf{rescue}_p(x_U, y_U, 0, 0)$
$\boldsymbol{c} \leftarrow \mathsf{rescue}^{\texttt{enc}}_{\texttt{ctr}}(\boldsymbol{k}, \boldsymbol{m})$
return $(R, \boldsymbol{c})$

$\texttt{Decrypt}(\mathsf{sk}, (R, \boldsymbol{c}))$

$U = \mathsf{sk}R$
$\boldsymbol{k} \leftarrow \mathsf{rescue}_p(x_U, y_U, 0, 0)$
$\boldsymbol{m} \leftarrow \mathsf{rescue}^{\texttt{dec}}_{\texttt{ctr}}(\boldsymbol{k}, \boldsymbol{c})$
return $\boldsymbol{m}$

4.1.7 Public-Key Encryption

Let $B \in \mathbb{F}_p^2$ be a generator of the Jubjub's large prime order subgroup.

We use a hybrid encryption scheme based on El-Gamal. Encrypting a message $\boldsymbol{m}$ is done by generating a random point $R = rB$, computing $U = (x_U, y_U) = r\mathsf{PK}$ where PK is the El-Gamal public key, and then interpreting $U \in F_p^2$ as a secret key for the rescue counter mode cipher described in Section 4.1.6. For the decryption, the owner of PK will use the corresponding private key sk to derive the symmetric cipher secret key $U = R^{\mathsf{sk}}$. With the symmetric key and the rescue cipher text, the plaintext can be recovered. Indeed, we can see U as the result of a key exchange between the owner of PK and the participant who generates $R = rB$. That is $U = \mathsf{sk}R = r\mathsf{PK} = r\mathsf{sk}B$.

4.1.8 Merkle tree

Our Merkle tree data structure produces a short representation of a set of elements of the form (uid, arc) where uid is a counter and arc is a commitment value and allows to prove efficiently that such elements have been inserted. The hash function used in our Merkle tree is the one defined in Section 4.1.4. We will denote it as $\mathsf{H} : \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{F}_p \to \mathbb{F}_p$ where $\mathbb{F}_p$ is the BLS12-381 scalar field. The Merkle tree is of fixed height, initialized during the setup phase. This data structure is also incremental, which means it is possible to dynamically insert new leaves (placed at the first available slot on the left), and update the root value in time $O(\log M)$ where M is the maximum number of leaves. A root of an empty subtree inside the Merkle tree is called an *empty node*.

In order to prevent attacks where the adversary leverages the flexible structure of the tree [8], we implement domain separation[9] for the following types of nodes: internal node, leaf and *empty node*. In practice this means the value of these nodes cannot be confused under some reasonable assumption about the hash function H. We define the values of such nodes as follows:

- *Empty node* value: 0
- Leaf value: $\mathsf{H}(0, a, b)$
- Internal node: $\mathsf{H}(a, b, c)$ where $a \neq 0$

Intuitively the reason why each of these kind of values are "separated" is because either the verification algorithm will reject a proof (e.g. because it contains an internal node of value $v = \mathsf{H}(a, b, c)$ with $a = 0$) or the adversary will have to find a collision (e.g. find an internal node value $v = \mathsf{H}(a, b, c)$ where $a \neq 0$ equal to a leaf value $v = \mathsf{H}(0, x, y)$). We analyze the security of the data structure in appendix C.

The Merkle tree is defined by four algorithms (see Algorithms 8, 9,10 for the pseudo-code):

- MTGen allows to initialize some empty Merkle tree.
- MTInsert allows to insert a new leaf, using the first empty slot on the left.
- MTProve allows to compute a merkle path corresponding to a specific leaf position.
- MTVerify is used to check that a leaf belongs to a tree represented by its root value, using a Merkle path computed by MTProve.

For those algorithms, h is the height of the tree, nodes are represented by the expression $Node(v, A, B, C)$ where v is some value and A, B, C are some pointers to other nodes. $EmptyNode()$ represents the node $Node(0, Nil, Nil, Nil)$ where Nil is the null pointer for a node. The global state of the Merkle tree st is a tuple (N, h, n) where N is a node, h is the height of the tree and n is the number of inserted elements. An element elem is a data structure with two fields uid and arc, both being values in $\mathbb{F}_p$.

[8] E.g. grafting attacks where the tree is extended with other nodes, or "internal-node-as-leaf" attacks; see https://blog.enuma.io/update/2019/06/10/merkle-trees-not-that-simple.html.

[9] See https://github.com/zcash/zcash/issues/501 for a discussion regarding implementing domain separation for the ZCash Merkle tree.

Algorithm 8 Merkle tree 1/3

decompBase3($index, h$)

```
/* Helper function to compute the decomposition of some integer in base 3. */
/* index: correspond to the position of a leaf (from left to right). */
/* h: height of the tree. */
/* returns: 3-ary decomposition of index of length h in "big-endian". */

current ← index
q ← current
components ← []
while current ≥ 3
   q, r ← current.divmod(3)
   components.append(r)
   current ← q
endwhile
/* Pad with 0s to the right if needed */
while len(components) < h
   components.append(0)
endwhile
/* Reverse the order of the list */
return components.reverse()
```

`MTGen`(h)

```
/* Generates an empty tree */
/* h: height of the tree */
/* returns: state for an empty tree */

rootNode ← Node(0, EmptyNode(), EmptyNode(), EmptyNode())
st ← (rootNode, h, 0)
return (st)
```

Algorithm 9 Merkle tree 2/3

MTInsert(st, elem)

```
/* Insert a new element into the tree */
/* st: state of the Merkle tree */
/* elem: element to be inserted */
/* returns: new state of the tree after insertion */

pos ← st.numElems
/* Insert new leaf at position st */
ThreeAryPos ← decompBase3(pos, st.h))
previousNode ← st.root
currentNode ← st.root
/* Go as deep in the tree as possible until finding an empty node */
i ← 0
branchNodes ← []
while not(currentNode.isEmpty())
  localPos ← ThreeAryPos[i]
  branchNodes.append(currentNode)
  previousNode ← currentNode
  currentNode ← match(localPos){
    0 => currentNode.left
    1 => currentNode.middle
    2 => currentNode.right
  }
  i ← i + 1
endwhile
/* Create the new nodes (always extend to the left) */
for j ∈ [i..st.h − 1] do
  newNode ← Node(0, EmptyNode(), EmptyNode(), EmptyNode())
  branchNodes.append(newNode)
  localPos ← ThreeAryPos[j − 1]
  match(localPos){
    0 => previousNode.left ← newNode,
    1 => previousNode.middle ← newNode,
    2 => previousNode.right ← newNode,
  }
  previousNode ← newNode
endfor
/* Add the leaf */
localPos ← ThreeAryPos[st.h − 1]
leafValue = H(0, elem.uid, elem.arc)
leafNode = Node(leafValue, Nil, Nil, Nil)
match(localPos){
  0 => previousNode.left ← leafNode,
  1 => previousNode.middle ← leafNode
  2 => previousNode.right ← leafNode
}
/* Refresh the values of the nodes on the path */
for i ∈ [0..h − 1] do
  level ← h − 1 − i
  node = branchNodes[level]
  node.value ← H(node.left.value, node.middle.value, node.right.value)
endfor
st.numElems ← st.numElems + 1
return (st)
```

Algorithm 10 Merkle tree 3/3

`MTProve`(st, pos)

```
/* Compute a membership proof for some element */
/* st: state of the tree */
/* pos: position of the element from left to right */
/* returns: proof π for the element in position pos */

/* Collect the siblings from the root to the leaf */
ThreeAryPos ← decompBase3(pos, st.h)
currentNode ← st.root
π ← []
for i ∈ [0..st.h − 1] do
  localPos ← ThreeAryPos[i]
  proofElem ← match(localPos){
      0 => [0, currentNode.middle, currentNode.right],
      1 => [1, currentNode.left, currentNode.right],
      2 => [2, currentNode.left, currentNode.middle],
    }
  π.append(proofElem)
  currentNode ← match(localPos){
      0 => currentNode.left
      1 => currentNode.middle
      2 => currentNode.right
    }
endfor
return π
```

`MTVerify`(rt, elem, π)

```
/* Recompute the root from the leaf and the path */
/* rt: root value of the Merkle tree */
/* elem: element to be tested for membership */
/* π: proof of membership */
/* returns: True if elem has been inserted, False otherwise. */

currentVal ← H(0, elem.uid, elem.arc)
l ← π.len()
for i ∈ [0..l − 1] do
  level ← l − 1 − i
  proofElem ← π[level]
  pos ← proofElem[0]
  sibling1 ← proofElem[1]
  sibling2 ← proofElem[2]
  /* Discard invalid internal node values */
  if (pos = 1||pos = 2)&&(sibling1.value = 0)
    return False
  endif
  if (pos = 0)&&(currentVal = 0)
    return False
  endif
  currentVal ← match(pos){
    0 => H(currentVal, sibling1.value, sibling2.value),
    1 => H(sibling1.value, currentVal, sibling2.value),
    2 => H(sibling1.value, sibling2.value, currentVal),
  }
endfor
/* Compare with the original root */
res ← (rt = currentVal)
return res
```

4.1.9 Polynomial commitments In this Section we consider a bilinear map $e : G_1 \times G_2 \to G_t$, where G_1, G_2, G_t are cyclic groups of prime order q. In practice, the curve used is BLS12-381 (see section 4.1.1). Let g_1 be a generator of G_1 and g_2 be a generator of G_2. We will use the additive notation for group operations and the following writing convention: Let $s \in \mathbb{Z}_q$ then the group element $s \cdot g_1$ will be written as $[s]_1$. More generally for $0 \leq i \leq n$ $[s^i]_1 := s^i \cdot g_1$ and $[s^i]_2 := [s^i] \cdot g_2$. Finally in all this section the algorithms `Commit`, `ProveEval` and `VerifyEval` use the public parameters generated by the algorithm `Setup` implicitly.

Algorithm 11 KZG scheme

`Setup`$(1^\lambda, n)$

$s \xleftarrow{\$} \mathbb{Z}_q$

$\mathtt{pp}_{g_1} \leftarrow ([1]_1, [s]_1, \cdots, [s^n]_1)$

$\mathtt{pp}_{g_2} \leftarrow ([1]_2, [s]_2)$

return $(\mathtt{pp}_{g_1}, \mathtt{pp}_{g_2})$

`Commit`(P)

Parse P as $P(X) = a_0 + a_1 X + a_2 X^2 + ... + a_n X^n$

$C \leftarrow P(s) \cdot g_1 = \sum_{i=0}^{n} a_i \cdot [s^i]_1$

return C

`ProveEval`(P, x)

$y \leftarrow P(x)$

$Q(X) \leftarrow (P(X) - P(x))/(X - x)$

return $Q(s) \cdot [1]_1$

`VerifyEval`(C, x, y, π)

return $e(C - y \cdot [1]_1, [1]_2) \stackrel{?}{=} e(\pi, [s]_2 - x \cdot [1]_2)$

[**KZG10**] The idea of the scheme is to represent a polynomial $P(X) = a_0 + a_1 X + \cdot + a_n X^n$ by the value $g_1^{P(s)}$ where s is a secret scalar. Proving that a value y is such that $y = P(x)$ for some x, consists of showing that the polynomial $X - x$ divides $P(X) - y$, i.e. $P(X) - y = Q(X)(X - x)$ for some polynomial $Q(X)$. While the value $Q(s)$ cannot be computed directly as s is secret, it is possible to compute the coefficients of the polynomial $Q(X)$ by performing an euclidean division and then using the public parameter $\mathtt{pp}_{g_1} := (g_1, g_1^s, g_1^{s^2}, ..., g_1^{s^n})$, we can obtain $g_1^{Q(s)}$. Finally the relation $P(X) - y = Q(X)(X - x)$ can be checked in the exponent using the bilinear operator.

The scheme is described in Algorithm 11.

Batched polynomial evaluation [**BDFG20a**] When several polynomials need to be evaluated at once on possibly different points, it is convenient to use some batching techniques in order to reduce the size of the evaluation proof. We use this approach in our Plonk implementation following the polynomial scheme variant described in Section 4.1 of [BDFG20a]. In this scheme the proof consists of two group elements of G_1 and the verifier needs to perform $t + 3$ group operations in G_1 and two pairing operations, where t is the number of polynomials and also the number of evaluation points.

Note that this batching technique can be generalized to any homomorphic polynomial commitment scheme. The batched version of the KZG scheme is described in Algorithm 12. The `Setup` and `Commit` operations are the same as in Algorithm 11. For this scheme we define the following polynomials:

- $Z(X) := \prod_{j=1}^{t}(X - x_j)$ where $\{x_j\}_{j=1}^{t}$ are the evaluation points.
- $\bar{Z}_i(X) = \prod_{j=1, j\neq i}^{t}(X - x_j)$.
- $R_i(X)$: polynomial such that $R_i(x_i) = y_i$ for $1 \leq i \leq t$, where y_i are the expected values of P_i on x_i.

Moreover the Prover and Verifier have access to an object `Transcript` which is derived from applying the Fiat-Shamir transform to the interactive variant of the protocol. `Transcript` has three methods :

- `Transcript`.*init*$(\cdot)$: initializes the state of the transcript with the parameters of the scheme.
- `Transcript`.*append*$(\cdot)$: allows to add information to the transcript.
- `Transcript`.*challenge*$(\cdot)$: returns a challenge value in $\mathbb{Z}_q$ based on the current state of the object.

Algorithm 12 Batch evaluation of different polynomials on different points

$\texttt{ProveEval}([P_1, P_2, \cdots, P_t], [x_1, x_2, \cdots, x_t])$

$\texttt{Transcript}.init(n, t, \texttt{pp}_{g_1}, \texttt{pp}_{g_2})$
$\alpha \leftarrow \texttt{Transcript}.challenge(\text{“}alpha\text{”})$
$H(X) \leftarrow \sum_{i=1}^{t} \alpha^{i-1} \bar{Z}_i(X)[P_i(X) - R_i(X)]$
$Q(X) \leftarrow H(X)/Z(X)$
$C_Q \leftarrow \texttt{KZG.Commit}(Q(X))$
$\texttt{Transcript}.append(C)$
$\rho \leftarrow \texttt{Transcript}.challenge(\text{“}rho\text{”})$
$G(X) \leftarrow \sum_{i=1}^{t} \alpha^{i-1} \bar{Z}_i(\rho)[P_i(X) - R_i(\rho)] - Q(X)Z(\rho)$
$\Delta \leftarrow \texttt{KZG.ProveEval}(G(X), \rho)$
return $\pi = (C_Q, \Delta)$

$\texttt{VerifyEval}([C_1, C_2, \cdots, C_t], [x_1, x_2, \cdots, x_t], y_1, y_2, \cdots, y_t], \pi)$

$\texttt{Transcript}.init(n, t, \texttt{pp}_{g_1}, \texttt{pp}_{g_2})$
$\alpha \leftarrow \texttt{Transcript}.challenge(\text{“}alpha\text{”})$
$(C_Q, \Delta) \leftarrow \pi$
$\texttt{Transcript}.append(C_q)$
$\rho \leftarrow \texttt{Transcript}.challenge(\text{“}rho\text{”})$
$Z(\rho) \leftarrow \prod_{i=1}^{t} (\rho - x_i)$
$G = C_i^{\sum_{i=1}^{t} \alpha^{i-1} \bar{Z}_i(\rho)} - [\sum_{i=1}^{t} \alpha^{i-1} \bar{Z}_i(\rho) R_i(\rho)]_1 - C_Q^{Z(\rho)}$
$res \leftarrow \texttt{KZG.VerifyEval}(G, \rho, 0, \Delta)$
return res

4.2 SNARKs Circuits

In the following sections, we specify the zk-SNARKs circuits for the transactions in our scheme. We start by introducing a variant of the PLONK constraint system [GWC19]. Next we describe in Section 4.2.2 and Section 4.2.3 the arithmetic and elliptic curve gates. We specify the crypto-primitive gadgets related to Rescue permutations in Section 4.2.4. Finally, we specify in Section 4.2.5 the circuits for the list of building block algorithms whose syntax was specified in Section 3.2.1. Some components we described may make use of others; the order of our presentation is “bottom-up”.

4.2.1 Constraint System The Configurable Asset Policy scheme uses PLONK [GWC19] as the zk-SNARKs proof system. Denote by $\boldsymbol{w} := (w_1, \ldots, w_m)$[10] a witness input where m is the number of witness elements. A “circuit” in the PLONK constraint system is a list of n constraints where the i-th $(1 \leq i \leq n)$ constraint is with the form

$$\boldsymbol{q}_L^{(i)} w_{\boldsymbol{a}_i} + \boldsymbol{q}_R^{(i)} w_{\boldsymbol{b}_i} + \boldsymbol{q}_M^{(i)} w_{\boldsymbol{a}_i} w_{\boldsymbol{b}_i} + \boldsymbol{q}_C^{(i)} = \boldsymbol{q}_O^{(i)} w_{\boldsymbol{c}_i}$$

Here $\boldsymbol{q}_L^{(i)}$, $\boldsymbol{q}_R^{(i)}$, $\boldsymbol{q}_O^{(i)}$, $\boldsymbol{q}_M^{(i)}$, $\boldsymbol{q}_C^{(i)}$ are some field elements called *selector coefficients* – that are used to configure a constraint/gate. For example, an addition gate $x + y = z$ can be captured by configuring $\boldsymbol{q}_L^{(i)} = \boldsymbol{q}_R^{(i)} = \boldsymbol{q}_O^{(i)} = 1$ (with the other selectors being zero); a multiplication gate $x \cdot y = z$ can be captured by configuring $\boldsymbol{q}_M^{(i)} = \boldsymbol{q}_O^{(i)} = 1$ (with the other selectors being zero). The indices $\boldsymbol{a}_i$, $\boldsymbol{b}_i$, $\boldsymbol{c}_i$ are called *indexers* – that point to some elements in the witness inputs $\boldsymbol{w} := (w_1, \ldots, w_m)$. We can think of an indexer (e.g., $\boldsymbol{a}_i$) as some wire that connects a gate input/output to a variable (e.g., $w_{\boldsymbol{a}_i}$). The circuit is satisfied by a witness $\boldsymbol{w}$ if all of the constraints are satisfied given the values of $\boldsymbol{w}$.

[10] WLOG the public inputs is also a part of the witness.

The original PLONK constraint system above is nice and simple, but may lead to large number of constraints for certain operations. For example, a single curve addition requires ≈ 7 constraints to be captured. Our scheme generalizes the PLONK constraint system and introduces more indexers and selector coefficients, thus making a constraint more expressive. For example, a curve addition now only requires 2 constraints to be captured; the Rescue hash gate (used in our system) requires only ≈ 144 constraints to be captured, while it requires ≈ 576 constraints using the original constraint system.

Specifically, our constraint system has a list of indexers $\mathcal{I} := (\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}) \in ([m]^n)^5$ and a list of selectors

$$\mathcal{Q} := (\boldsymbol{q}_1, \boldsymbol{q}_2, \boldsymbol{q}_3, \boldsymbol{q}_4, \boldsymbol{q}_{M_{1,2}}, \boldsymbol{q}_{M_{3,4}}, \boldsymbol{q}_O, \boldsymbol{q}_C, \boldsymbol{q}_{H_1}, \boldsymbol{q}_{H_2}, \boldsymbol{q}_{H_3}, \boldsymbol{q}_{H_4}, \boldsymbol{q}_{\mathsf{ecc}}) \in (\mathbb{F}_p^n)^{13} .$$

Informally, $\boldsymbol{q}_C$ and $\boldsymbol{q}_O$ are used to configure constants and output wires respectively; $(\boldsymbol{q}_1, \ldots, \boldsymbol{q}_4, \boldsymbol{q}_{M_{1,2}}, \boldsymbol{q}_{M_{3,4}})$ are mainly used for configuring arithmetic operations; $(\boldsymbol{q}_{H_1}, \ldots, \boldsymbol{q}_{H_4})$ are used for configuring Rescue hash operations; $\boldsymbol{q}_{\mathsf{ecc}}$ is used to configure elliptic curve operations.

The i-th $(1 \le i \le n)$ constraint is satisfied if and only if

$$\begin{aligned} &\boldsymbol{q}_1^{(i)} w_{\boldsymbol{a}_i} + \boldsymbol{q}_2^{(i)} w_{\boldsymbol{b}_i} + \boldsymbol{q}_3^{(i)} w_{\boldsymbol{c}_i} + \boldsymbol{q}_4^{(i)} w_{\boldsymbol{d}_i} + \boldsymbol{q}_{M_{1,2}}^{(i)} w_{\boldsymbol{a}_i} w_{\boldsymbol{b}_i} + \boldsymbol{q}_{M_{3,4}}^{(i)} w_{\boldsymbol{c}_i} w_{\boldsymbol{d}_i} + \boldsymbol{q}_{C,i} \\ &\qquad + \boldsymbol{q}_{H_1}^{(i)} w_{\boldsymbol{a}_i}^5 + \boldsymbol{q}_{H_2}^{(i)} w_{\boldsymbol{b}_i}^5 + \boldsymbol{q}_{H_3}^{(i)} w_{\boldsymbol{c}_i}^5 + \boldsymbol{q}_{H_4}^{(i)} w_{\boldsymbol{d}_i}^5 + \boldsymbol{q}_{\mathsf{ecc}}^{(i)} w_{\boldsymbol{a}_i} w_{\boldsymbol{b}_i} w_{\boldsymbol{c}_i} w_{\boldsymbol{d}_i} w_{\boldsymbol{e}_i} = \boldsymbol{q}_O^{(i)} w_{\boldsymbol{e}_i} . \end{aligned} \quad (1)$$

We note that, however, as a single constraint becomes more complex in the generalized constraint system, the prover's cost per constraint also increases. Informally, this is because when computing a proof, (i) the prover has to perform more operations per constraint; (ii) the prover needs to interpolate and commit more polynomials for selectors and indexers; and (iii) the *ratio* between the quotient polynomial's degree and the total number of constraints increases when the *constraint arity* (i.e., the number of indexers) and the *constraint degree*[11] increases. We refer to [GWC19] for more technical details.

Therefore, given the arguments above, care must be taken to achieve the optimal balance between constraint expressiveness and the prover's cost per constraint.

Notation. In the following sections, when defining constraints for circuits, we only specify the *selector coefficients* and the *variables* (e.g., the witness $w_{\boldsymbol{a}_i}$ above) of the constraints, as the indexers are implicit given the variables. Moreover, when unnecessary, we omit the superscript i in the selectors notation, and set a selector coefficient to 0 by default if not explicitly specified in a constraint. Without loss of generality, we also setup two default variables 0_{var} and 1_{var}, which point to values 0 and 1 respectively.

4.2.2 Arithmetic Gates

In this section, we specify the basic arithmetic gates used in our scheme.

Addition. An addition constraint $x + y = z$ can be implemented as

$$\boldsymbol{q}_1 \cdot x + \boldsymbol{q}_2 \cdot y = \boldsymbol{q}_O \cdot z$$

where $\boldsymbol{q}_1 = \boldsymbol{q}_2 = \boldsymbol{q}_O = 1$.

Constant Addition. A constant addition constraint $x + c = y$ (where c is a public constant) can be implemented as

$$\boldsymbol{q}_1 \cdot x + \boldsymbol{q}_C = \boldsymbol{q}_O \cdot y$$

where $\boldsymbol{q}_1 = \boldsymbol{q}_O = 1$ and $\boldsymbol{q}_C = c$.

Subtraction. A subtraction constraint $x - y = z$ can be implemented as

$$\boldsymbol{q}_1 \cdot x + \boldsymbol{q}_2 \cdot y = \boldsymbol{q}_O \cdot z$$

where $\boldsymbol{q}_1 = \boldsymbol{q}_O = 1$ and $\boldsymbol{q}_2 = -1$.

Linear Combination. A linear combination constraint $c_1 x_1 + c_2 x_2 + c_3 x_3 + c_4 x_4 = y$ (where $x_1, \ldots, x_4, y$ are variables and $c_1, \ldots, c_4 \in \mathbb{F}_p$ are coefficients) can be implemented as

$$\boldsymbol{q}_1 \cdot x_1 + \boldsymbol{q}_2 \cdot x_2 + \boldsymbol{q}_3 \cdot x_3 + \boldsymbol{q}_4 \cdot x_4 = \boldsymbol{q}_O \cdot y$$

where $\boldsymbol{q}_i = c_i$ $(1 \le i \le 4)$ and $\boldsymbol{q}_O = 1$.

[11] Constraint degree is the maximal degree of a monomial in the constraint formula. E.g., in our constraint system, the constraint degree of Equation 1 is 6.

Sum Gates. A sum gate $\sum_{i=0}^{n} x_i = y$ can be implemented as follows: WLOG we assume n is a multiple of 3 and denote by $n' := n/3$. We add the following *linear combination gates*:

- $x_0 + x_1 + x_2 + x_3 = z_1$ where z_1 is an intermediate variable.
- For i from 2 to $n' - 1$:
$$z_{i-1} + x_{3i-2} + x_{3i-1} + x_{3i} = z_i$$
where z_i is an intermediate variable.
- $z_{n'-1} + x_{n-2} + x_{n-1} + x_n = y$.

Multiplication. A multiplication constraint $x \cdot y = z$ can be implemented as

$$\boldsymbol{q}_{M_{1,2}} \cdot x \cdot y = \boldsymbol{q}_O \cdot z$$

where $\boldsymbol{q}_{M_{1,2}} = \boldsymbol{q}_O = 1$.

Multiply-then-add gates. A mul-add constraint $c_1 x_1 x_2 + c_2 x_3 x_4 = y$ (where $x_1, \ldots, x_4, y$ are variables and $c_1, c_2 \in \mathbb{F}_p$ are coefficients) can be implemented as

$$\boldsymbol{q}_{M_{1,2}} \cdot x_1 \cdot x_2 + \boldsymbol{q}_{M_{3,4}} \cdot x_3 \cdot x_4 = \boldsymbol{q}_O \cdot y$$

where $\boldsymbol{q}_{M_{1,2}} = c_1$, $\boldsymbol{q}_{M_{3,4}} = c_2$, and $\boldsymbol{q}_O = 1$.

Boolean Gates. A boolean constraint $x \in \{0, 1\}$ can be implemented by a multiplication constraint $x \cdot x = x$.

Constant Gates. A constant constraint $x = c$ (where c is a public constant) can be implemented as

$$\boldsymbol{q}_C = \boldsymbol{q}_O \cdot x$$

where $\boldsymbol{q}_C = c$ and $\boldsymbol{q}_O = 1$.

Equality. An equality constraint $x = y$ can be implemented by a subtraction constraint $x - y = 0_{\mathsf{var}}$.

Conditional Selection. A conditional selection constraint $(b \ ? \ x_0 : x_1) = y$ (where b is a boolean variable and $y = x_b$) can be implemented as

$$\boldsymbol{q}_2 \cdot x_0 + \boldsymbol{q}_{M_{1,2}} \cdot b \cdot x_0 + \boldsymbol{q}_{M_{3,4}} \cdot b \cdot x_1 = \boldsymbol{q}_O \cdot y$$

where $\boldsymbol{q}_2 = \boldsymbol{q}_{M_{3,4}} = \boldsymbol{q}_O = 1$ and $\boldsymbol{q}_{M_{1,2}} = -1$. The constraint is correct because

$$x_b = (1 - b) \cdot x_0 + b \cdot x_1 = x_0 - b \cdot x_0 + b \cdot x_1 \,,$$

and

$$\boldsymbol{q}_O \cdot y = \boldsymbol{q}_2 \cdot x_0 + \boldsymbol{q}_{M_{1,2}} \cdot b \cdot x_0 + \boldsymbol{q}_{M_{3,4}} \cdot b \cdot x_1 \iff y = x_0 - b \cdot x_0 + b \cdot x_1 \,.$$

IsZero Gates. Given variable x, we want to obtain a bool variable y which equals 1 if x has value zero, or $y := 0$ otherwise. Denote by x^{-1} an intermediate variable. The gate can be implemented by adding following constraints:

1. A mul-add constraint $x \cdot x^{-1} + 1_{\mathsf{var}} \cdot y = 1_{\mathsf{var}}$. Equivalently, it is constraining that $1 - x \cdot x^{-1} = y$.
2. A multiplication constraint $y \cdot x = 0_{\mathsf{var}}$.

To see why the gate is correct we argue that

- when $x = 0$, the gate can be satisfied only if $y = 1$;
- when $x \neq 0$, the gate can be satisfied only if $y = 0$.

Specifically, if $x = 0$ and the constraints are satisfied, then $y = 1 - x \cdot x^{-1} = 1 - 0 \cdot x^{-1} = 1$. If $x \neq 0$ and the constraints are satisfied, then $y = 0$ because of the constraint $y \cdot x = 0_{\mathsf{var}}$.

We also note that the constraints can always be satisfied regardless of the value of x:

- If $x = 0$, the constraints are satisfied by setting $x^{-1} := 0$ and $y := 1$.
- If $x \neq 0$, the constraints are satisfied by setting $y := 0$ and x^{-1} the inverse of x.

Match Gates. Given variables x_1, x_2, y, our goal is to assign $y := 1$ when $x_1 = x_2$, and $y := 0$ otherwise. Denote by Δ an intermediate variable. The match gate can be implemented by adding the following gates:

1. A subtraction constraint $x_1 - x_2 = \Delta$.
2. An is-zero gate $\mathsf{is\_zero}(\Delta) = y$.

Range-checks. A range-check gate $x \in [0, 2^\ell)$ (where ℓ is an integer parameter) can be implemented as follows: For ease of explanation we assume $\ell - 1$ is a multiple of 3 and define $\ell' := (\ell - 1)/3$. Denote by $(b_0, \ldots, b_{\ell-1})$ a list of boolean variables and $(\mathsf{acc}_0, \ldots, \mathsf{acc}_{\ell'-1})$ a list of accumulation variables (where $\mathsf{acc}_0 = b_{\ell-1}$). We add the following constraints to enforce $x = \sum_{i=0}^{\ell-1} b_i \cdot 2^i$.

- For $i \in [\ell' - 1]$, add a linear combination constraint

$$8 \cdot \mathsf{acc}_{i-1} + 4 \cdot b_{\ell-1-3i+2} + 2 \cdot b_{\ell-1-3i+1} + b_{\ell-1-3i} = \mathsf{acc}_i \,.$$

- Add a linear combination constraint

$$8 \cdot \mathsf{acc}_{\ell'-1} + 4 \cdot b_2 + 2 \cdot b_1 + b_0 = x \,.$$

The gate above makes use of ℓ boolean constraints (to constrain b_i) and $\ell/3$ linear combination constraints, which is $\ell + \ell/3$ constraints in total.

4.2.3 Elliptic Curve Gates

In this section, we describe the elliptic curve gates used in our scheme. The gates make use of the arithmetic gates specified in Section 4.2.2.

Notation. The curve we use is called Jubjub [ZCa17], which is built on top of the BLS12-381 scalar field. In the following, denote by $\mathbb{F}_p$ the BLS12-381 scalar field, and $\mathbb{F}_r$ the Jujub scalar field. We express a curve point $V = (x, y)$ in its affine form where $x, y \in \mathbb{F}_p$ are the x-coordinate and y-coordinate. Thus we can define a *point variable* as a pair $(x, y) \in \mathbb{F}_p \times \mathbb{F}_p$. For points U, V and a scalar $x \in \mathbb{F}_r$, we use $U + V$ to denote point addition and $x \cdot U$ to denote scalar multiplication.

Neutral Point Check. Given a point variable $V = (x, y) \in \mathbb{F}_p^2$, a neutral point checking gate $\mathsf{neutral}(V) = b$ constrains b to be a boolean variable indicating whether $V = \infty = (0, 1)$. It consists of the following constraints:

- A match gate (c.f. Section 4.2.2) $b_x = \mathsf{match}(x, 0_{\mathsf{var}})$.
- A match gate $b_y = \mathsf{match}(y, 1_{\mathsf{var}})$.
- A multiplication gate $b = b_x \cdot b_y$.

Curve Equation Check. A curve equation gate $\mathsf{oncurve}(V)$ constrains $V = (x, y) \in \mathbb{F}_p^2$ to be a point on the Jubjub curve. It can be implemented via the constraint

$$\boldsymbol{q}_{M_{1,2}} \cdot x \cdot x + \boldsymbol{q}_{M_{3,4}} \cdot y \cdot y + \boldsymbol{q}_C + \boldsymbol{q}_{\mathsf{ecc}} \cdot x \cdot x \cdot y \cdot y \cdot 1_{\mathsf{var}} = \boldsymbol{q}_O \cdot 1_{\mathsf{var}}$$

where $(x, x, y, y, 1_{\mathsf{var}})$ are the wires of the constraint, and $\boldsymbol{q}_{M_{1,2}} = \boldsymbol{q}_C = 1$, $\boldsymbol{q}_{M_{3,4}} = -1$, $\boldsymbol{q}_{\mathsf{ecc}} = d = -(10240/10241)$, $\boldsymbol{q}_O = 0$. The constraint is equivalent to the Jubjub curve equation

$$-x^2 + y^2 = 1 + dx^2y^2 \,.$$

Point Addition. Given point variables $V_1 = (x_1, y_1)$, $V_2 = (x_2, y_2)$ and $V_3 = (x_3, y_3)$, a point addition gate $V_1 + V_2 = V_3$ can be implemented as follows: Recall the point addition formula for Jubjub that

$$(x_3, y_3) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1y_1x_2y_2}, \frac{x_1x_2 + y_1y_2}{1 - dx_1y_1x_2y_2} \right) ,$$

where $d = -(10240/10241) \in \mathbb{F}_p$ is a constant. By lifting the denominator to the nominator side and rearranging terms, we can rewrite the above formula as

$$x_3 = x_1y_2 + x_2y_1 - dx_1y_1x_2y_2x_3 \,, \qquad y_3 = x_1x_2 + y_1y_2 + dx_1y_1x_2y_2y_3 \,.$$

Hence we can implement the point addition gate with 2 constraints, one for the x-coordinate and one for the y-coordinate. More specifically, the first constraint is

$$\boldsymbol{q}_{M_{1,2}} \cdot x_1 y_2 + \boldsymbol{q}_{M_{3,4}} \cdot x_2 y_1 + \boldsymbol{q}_{\mathsf{ecc}} \cdot x_1 y_1 x_2 y_2 x_3 = \boldsymbol{q}_O \cdot x_3$$

where $\boldsymbol{q}_{M_{1,2}} = \boldsymbol{q}_{M_{3,4}} = \boldsymbol{q}_O = 1$ and $\boldsymbol{q}_{\mathsf{ecc}} = -d$. The second constraint is

$$\boldsymbol{q}_{M_{1,2}} \cdot x_1 x_2 + \boldsymbol{q}_{M_{3,4}} \cdot y_1 y_2 + \boldsymbol{q}_{\mathsf{ecc}} \cdot x_1 y_1 x_2 y_2 y_3 = \boldsymbol{q}_O \cdot y_3$$

where $\boldsymbol{q}_{M_{1,2}} = \boldsymbol{q}_{M_{3,4}} = \boldsymbol{q}_O = 1$ and $\boldsymbol{q}_{\mathsf{ecc}} = d$.

Fixed-base Scalar Multiplication. We first specify a building block gate called *point selection gate.*

Point Selection Gates. Denote by $U_0 = (0, 1)$ the neutral point of the Jubjub curve, and U_1, U_2, U_3 three *public fixed* points. Given two boolean variables b_0, b_1 and a point variable V, a point selection gate $\mathsf{sel}_{\mathsf{ecc}}^{U_1,U_2,U_3}(b_0, b_1) = V$ (where $V = U_{b_0+2b_1}$) can be implemented as follows. Denote by $U_i = (x_i, y_i)$ $(1 \leq i \leq 3)$ and $V = (x^*, y^*)$. it is not hard to see that x^* is supposed to be

$$\begin{aligned} x^* &= b_0 \cdot (1 - b_1) \cdot x_1 + (1 - b_0) \cdot b_1 x_2 + b_0 b_1 x_3 \\ &= x_1 b_0 + x_2 b_1 + (x_3 - x_2 - x_1) \cdot b_0 b_1 \end{aligned}$$

and y^* is supposed to be

$$\begin{aligned} y^* &= (1 - b_0) \cdot (1 - b_1) + b_0 \cdot (1 - b_1) \cdot y_1 + (1 - b_0) \cdot b_1 \cdot y_2 + b_0 \cdot b_1 \cdot y_3 \\ &= (y_1 - 1) \cdot b_0 + (y_2 - 1) \cdot b_1 + (y_3 - y_2 - y_1 + 1) \cdot b_0 b_1 + 1\,. \end{aligned}$$

We stress that here b_0 and b_1 are variables, while x_i and y_i $(1 \leq i \leq 3)$ are public constants. Hence we can express the gate by adding the following 2 constraints:

$$\boldsymbol{q}_1 \cdot b_0 + \boldsymbol{q}_2 \cdot b_1 + \boldsymbol{q}_{M_{1,2}} \cdot b_0 b_1 = \boldsymbol{q}_O \cdot x^*$$

where $\boldsymbol{q}_1 = x_1$, $\boldsymbol{q}_2 = x_2$, $\boldsymbol{q}_{M_{1,2}} = x_3 - x_2 - x_1$, and $\boldsymbol{q}_O = 1$, and

$$\boldsymbol{q}_1 \cdot b_0 + \boldsymbol{q}_2 \cdot b_1 + \boldsymbol{q}_{M_{1,2}} \cdot b_0 b_1 + \boldsymbol{q}_C = \boldsymbol{q}_O \cdot y^*$$

where $\boldsymbol{q}_1 = y_1 - 1$, $\boldsymbol{q}_2 = y_2 - 1$, $\boldsymbol{q}_{M_{1,2}} = y_3 - y_2 - y_1 + 1$, and $\boldsymbol{q}_O = \boldsymbol{q}_C = 1$.

Scalar Multiplication. A fixed-base scalar multiplication gate $s \cdot B = V$ (where B is a *fixed* base point, $s \in \mathbb{F}_p$ is a variable encoding a Jubjub scalar[12], and V is a point variable) can be implemented as follows: Let ℓ be the maximal bit-length of a Jubjub scalar. WLOG we assume ℓ is even and denote by $\ell' := \ell/2$. Define a list of base points

- $\boldsymbol{B_1} = (B, 4 \cdot B, \ldots, 4^{\ell'-1} \cdot B)$,
- $\boldsymbol{B_2} = (2 \cdot B, 2 \cdot 4 \cdot B, \ldots, 2 \cdot 4^{\ell'-1} \cdot B)$,
- $\boldsymbol{B_3} = (3 \cdot B, 3 \cdot 4 \cdot B, \ldots, 3 \cdot 4^{\ell'-1} \cdot B)$.

Denote by $(b_0, \ldots, b_{\ell-1})$ a list of boolean variables and $(\mathsf{acc}_0, \ldots, \mathsf{acc}_{\ell'-1})$ a list of accumulation *point variables.* We add the following gates/constraints:

1. Add a rangecheck (unpacking) gate $\mathsf{range}(s, \ell) = (b_0, \ldots, b_{\ell-1})$.
2. Add a point selection gate
$$\mathsf{sel}_{\mathsf{ecc}}^{\boldsymbol{B_1}[0],\boldsymbol{B_2}[0],\boldsymbol{B_3}[0]}(b_0, b_1) = \mathsf{acc}_0$$
where $\boldsymbol{B_i}[0]$ is the first element of $\boldsymbol{B_i}$.
3. For each $i \in [\ell' - 2]$, add a point selection gate
$$\mathsf{sel}_{\mathsf{ecc}}^{\boldsymbol{B_1}[i],\boldsymbol{B_2}[i],\boldsymbol{B_3}[i]}(b_{2i}, b_{2i+1}) = Z_i$$
where Z_i is an intermediate point variable, and add a point addition gate
$$\mathsf{acc}_{i-1} + Z_i = \mathsf{acc}_i\,.$$

[12] WLOG we also assume $s \in \mathbb{F}_r$.

4. Add a point selection gate

$$\mathsf{sel}_{\mathsf{ecc}}^{\boldsymbol{B_1}[\ell'-1],\boldsymbol{B_2}[\ell'-1],\boldsymbol{B_3}[\ell'-1]}(b_{2\ell'-2}, b_{2\ell'-1}) = Z_{\ell'-1}$$

where $Z_{\ell'-1}$ is an intermediate point variable, and add a point addition gate

$$\mathsf{acc}_{\ell'-2} + Z_{\ell'-1} = V\,.$$

Since $s = \sum_{i=0}^{\ell-1} b_i \cdot 2^i$, it holds that $V = s \cdot B$. The above gate makes use of 1 rangecheck gate, ℓ' point selection gates and $\ell' - 1$ point addition gate. Since both point selection and point addition cost 2 constraints per gate and a rangecheck gate needs $\approx 1.3\ell$ constraints, in total, it requires approximately 3.3 constraints per scalar bit. Sometimes it is unnecessary to add the rangecheck gate when the boolean variables $(b_0, \ldots, b_{\ell-1})$ are already set, and hence it requires ≈ 2 constraints per scalar bit.

Variable-base Scalar Multiplication. We first specify a gate called the *point variable selection gate*. Given 2 *point variables* $U_0 = (x_0, y_0)$, $U_1 = (x_1, y_1)$, a boolean variables b, and a point variable $V = (x^*, y^*)$, the goal is to constrain V to be U_b. A point variable selection gate $\mathsf{sel}'_{\mathsf{ecc}}(U_0, U_1, b) = V$ can be implemented by adding two conditional selection gates (c.f. Section 4.2.2):

$$\mathsf{sel}(x_0, x_1, b) = x^*\,, \qquad \mathsf{sel}(y_0, y_1, b) = y^*\,.$$

A variable-base scalar multiplication gate $s \cdot U = V$ (where U,V are point variables and $s \in \mathbb{F}_p$ is a variable encoding a Jubjub scalar) can be implemented as follows: Let ℓ be the maximal bit-length of a Jubjub scalar. Denote by $I = (0_{\mathsf{var}}, 1_{\mathsf{var}})$ a (default) *point variable* that represents the neutral point of the Jubjub curve,

$(b_0, \ldots, b_{\ell-1})$ a list of boolean variables, and $(\mathsf{acc}_1, \ldots, \mathsf{acc}_\ell)$ a list of accumulation *point variables* where $\mathsf{acc}_\ell = I$. We add the following gates/constraints:

1. Add a rangecheck (unpacking) gate $\mathsf{range}(s, \ell) = (b_0, \ldots, b_{\ell-1})$.
2. For i from $\ell - 1$ downto 1, add a point variable selection gate

   $$\mathsf{sel}'_{\mathsf{ecc}}(I, U, b_i) = Z_i$$

   where Z_i is an intermediate point variable, and add two point addition gates

   $$\mathsf{acc}_{i+1} + \mathsf{acc}_{i+1} = \mathsf{acc}_i^*\,, \qquad \mathsf{acc}_i^* + Z_i = \mathsf{acc}_i\,,$$

   where acc_i^* is an intermediate point variable.
3. Add a point variable selection gate

   $$\mathsf{sel}'_{\mathsf{ecc}}(I, U, b_0) = Z_0$$

   where Z_0 is an intermediate point variable, and add two point addition gates

   $$\mathsf{acc}_1 + \mathsf{acc}_1 = \mathsf{acc}_0^*\,, \qquad \mathsf{acc}_0^* + Z_0 = V\,,$$

   where acc_0^* is an intermediate point variable.

Since $s = \sum_{i=0}^{\ell-1} b_i \cdot 2^i$, it holds that $V = s \cdot U$. Excluding the rangecheck gate, the above gate makes use of 2ℓ point addition gates and ℓ point variable selection gate. Since both point variable selection and point addition cost 2 constraints per gate, in total, there are 6ℓ constraints (i.e., 6 constraints per scalar bit).

4.2.4 Rescue-based Gadgets

In this section, we specify the circuits for the crypto-primitives built on top of Rescue permutations.

Rescue Permutations. Recall the Rescue permutation algorithm in Section 4.1.3. Each Rescue state consists of $w = 4$ elements in $\mathbb{F}_p$, hence we can refer to $\mathsf{st} = (x_1, \ldots, x_w)$ as a *state variable* where $(x_1, \ldots, x_w)$ are w variables. We first specify the following building block gates.

Constant State Addition. A constant state addition gate $\mathsf{add}_c(\mathsf{st}_0, \boldsymbol{c}) = \mathsf{st}_1$ (where st_0, st_1 are state variables and $\boldsymbol{c}$ is a constant Rescue state) is implemented as follows: For $i \in [w]$, add a *constant addition gate* (c.f. Section 4.2.2) $x_i + c_i = y_i$ where x_i, c_i, y_i is the i-th element of st_0, $\boldsymbol{c}_0$, and st_1 respectively.

State Inversion. Given state variables $\mathsf{st}_0 = (x_1, \ldots, x_w)$ and $\mathsf{st}_1 = (y_1, \ldots, y_w)$, a state inversion gate $\mathsf{inv}_\alpha(\mathsf{st}_0) = \mathsf{st}_1$ constrains that $y_i = x_i^{1/\alpha}$ for all $i \in [w]$. In our scheme, the parameter α is 5, and the gate can be implemented as follows: For each $i \in [w]$, add a constraint

$$\boldsymbol{q}_{H_1} \cdot y_i^5 = \boldsymbol{q}_O \cdot x_i$$

where $\boldsymbol{q}_O = \boldsymbol{q}_{H_1} = 1$.

Affine Transformation. Given state variables $\mathsf{st}_0 = (x_1, \ldots, x_w)$ and $\mathsf{st}_1 = (y_1, \ldots, y_w)$, a public constant matrix $M \in \mathbb{F}_p^{w \times w}$, and a public constant state $\boldsymbol{c} = (c_1, \ldots, c_w)$, an *affine transformation gate* $\mathsf{affine}(\mathsf{st}_0; M, \boldsymbol{c}) = \mathsf{st}_1$ constrains that for every $i \in [w]$, it holds that

$$y_i = \sum_{j=1}^{w} M_{i,j} \cdot x_j + c_i \,.$$

The gate can be implemented as follows: For each $i \in [w]$, add a constraint

$$\boldsymbol{q}_1 \cdot x_1 + \boldsymbol{q}_2 \cdot x_2 + \boldsymbol{q}_3 \cdot x_3 + \boldsymbol{q}_4 \cdot x_4 + \boldsymbol{q}_C = \boldsymbol{q}_O \cdot y_i$$

where $\boldsymbol{q}_j = M_{i,j}$ $(1 \le j \le 4)$, $\boldsymbol{q}_C = c_i$, and $\boldsymbol{q}_O = 1$.

Non-linear Transformation. Given state variables $\mathsf{st}_0 = (x_1, \ldots, x_w)$ and $\mathsf{st}_1 = (y_1, \ldots, y_w)$, a public constant matrix $M \in \mathbb{F}_p^{w \times w}$, and a public constant state $\boldsymbol{c} = (c_1, \ldots, c_w)$, a *non-linear transformation gate* $\mathsf{nonlin}_\alpha(\mathsf{st}_0; M, \boldsymbol{c}) = \mathsf{st}_1$ constrains that for every $i \in [w]$, it holds that

$$y_i = \sum_{j=1}^{w} M_{i,j} \cdot x_j^5 + c_i \,.$$

The gate can be implemented as follows: For each $i \in [w]$, add a constraint

$$\boldsymbol{q}_{H_1} \cdot x_1^5 + \boldsymbol{q}_{H_2} \cdot x_2^5 + \boldsymbol{q}_{H_3} \cdot x_3^5 + \boldsymbol{q}_{H_4} \cdot x_4^5 + \boldsymbol{q}_C = \boldsymbol{q}_O \cdot y_i$$

where $\boldsymbol{q}_{H_j} = M_{i,j}$ $(1 \le j \le 4)$, $\boldsymbol{q}_C = c_i$, and $\boldsymbol{q}_O = 1$.

The circuit $\mathsf{rescue}_p(x_1, \ldots, x_4) = (y_1, \ldots, y_4)$ for Rescue permutations is straightforward by instantiating the building block operations in Algorithm 4 with the subcircuits specified above. We stress that the input key $\boldsymbol{k}$ in Algorithm 4 is a zero constant vector and thus the keystream is a vector of public constants, so we do not need to implement the circuit for the key scheduling algorithm. In total, the number of constraints needed is 148 (with number of rounds $\mathtt{n_r} = 12$ and $\alpha = 5$).

Rescue Ciphers. The Rescue block cipher circuit is almost identical to that of a Rescue permutation except that the input key becomes a secret state variable instead of a zero constant vector. Hence we need to further implement the *key scheduling circuit*. Moreover, the keystreams used in the round functions become variables instead of constants, so we need to implement the following *state addition gate*:

- $\mathsf{add}(\mathsf{st}_0, \mathsf{st}_1) = \mathsf{st}_2$: Given state variables st_0 and st_1, constrains a state variable st_2 to be $\mathsf{st}_0 + \mathsf{st}_1$ where $+$ indicates element-wise addition. The gate can be implemented by adding w addition gates, one per element of the state.

The Rescue cipher circuit

$$\mathsf{rescue}_c(\boldsymbol{k}; (m_1, \ldots, m_4)) = (c_1, \ldots, c_4)$$

(where $\boldsymbol{k} \in \mathbb{F}_p^4$ is a symmetric cipher key) is straightforward by instantiating the building block operations in Algorithm 4 with the sub-circuits specified above. In total, the number of constraints needed for Rescue block cipher is 344 (148 from key scheduling and 196 from rescue cipher).

Rescue Counter Modes. We can instantiate counter mode encryptions from Rescue block ciphers. We will denote $\mathsf{rescue}_{\mathsf{ctr}}^{\mathsf{enc}}$ as $\mathsf{rescue}_{\mathsf{ctr}}$ (see Section 4.1.6). We define the following variables:

- $\boldsymbol{m} = (m_1, \ldots, m_\ell) \in \mathbb{F}_p^\ell$ are ℓ variables representing the plaintexts. WLOG we assume that ℓ is a multiple of 4 (otherwise we can do padding) and denote by $\ell' := \ell/4$.

- $\boldsymbol{k} \in \mathbb{F}_p^w$ is a Rescue state variable denoting the symmetric encryption key.
- $(\mathsf{iv}_1, \ldots, \mathsf{iv}_{\ell'}) \in \mathbb{F}_p^{\ell'}$ are the counter variables.
- $\boldsymbol{c} = (c_1, \ldots, c_\ell) \in \mathbb{F}_p^\ell$ are ℓ variables representating the ciphertexts.

The Rescue counter mode circuit $\mathsf{rescue}_{\mathsf{ctr}}(\boldsymbol{k}; \mathsf{iv}_1, \boldsymbol{m}) = \boldsymbol{c}$ is implemented as follows: Denote by 0_{var} and 1_{var} the default variables for value zero and one. For $i \in [\ell']$, add the following gates:

- A Rescue cipher gate
$$\mathsf{rescue}_c(\boldsymbol{k}; (\mathsf{iv}_i, 0_{\mathsf{var}}, 0_{\mathsf{var}}, 0_{\mathsf{var}})) = \mathsf{st}_i$$
where $\mathsf{st}_i \in \mathbb{F}_p^4$ is an intermediate state variable.
- A state addition gate (that masks the plaintexts)
$$\mathsf{add}(\mathsf{st}_i, (m_{4i-3}, m_{4i-2}, m_{4i-1}, m_{4i})) = (c_{4i-3}, c_{4i-2}, c_{4i-1}, c_{4i}) .$$
- If $i < \ell'$, add an addition gate $\mathsf{iv}_i + 1_{\mathsf{var}} = \mathsf{iv}_{i+1}$.

The above circuit makes use of ℓ' Rescue cipher gates, ℓ' state addition gates, and $\ell' - 1$ addition gates.

PRFs. As described in Section 4.1.5, we can instantiate PRFs on top of Rescue permutations. A PRF circuit $\mathsf{PRF}_s(m_0, m_1, m_2, m_3) = y$ (where $s \in \mathbb{F}_p$ is a seed variable, $(m_0, m_1, m_2, m_3) \in \mathbb{F}_p^4$ are input variables and $y \in \mathbb{F}_p$ is output variable) can be implemented as follows:

1. Add an addition gate (c.f. Section 4.2.2) $m_3 + s = z$ where z is an intermediate variable.
2. Add a Rescue permutation gate
$$\mathsf{rescue}_p(m_0, m_1, m_2, z) = (y, z_1, z_2, z_3)$$
where z_1, z_2, z_3 are intermediate variables.

Sponge-based Hashes. As described in Section 4.1.4, we can instantiate sponge-based hashes on top of Rescue permutations. Recall that in our instantiation, the rate of the Rescue permutations is $r := 3$ and the capacity is $c := 1$. Given input variables $\boldsymbol{m} := (m_0, \ldots, m_{3\ell-1}) \in \mathbb{F}_p^{3\ell}$ and an output variable $y \in \mathbb{F}_p$, a sponge-based hash circuit $\mathsf{sponge}_\ell(\boldsymbol{m}) = y$ is implemented as follows: Denote by $(\mathsf{st}_0, \ldots, \mathsf{st}_\ell)$ a list of state variables where $\mathsf{st}_0 = (0_{\mathsf{var}}, 0_{\mathsf{var}}, 0_{\mathsf{var}}, 0_{\mathsf{var}})$. We add the following gates/constraints:

- For $i \in [\ell]$, add a state addition gate (c.f. Section 4.2.4) that absorbs message blocks
$$\mathsf{add}(\mathsf{st}_{i-1}, (m_{3i-3}, \ldots, m_{3i-1}, 0_{\mathsf{var}})) = Z_i$$
where Z_i is an intermediate state variable, and add a Rescue permutation gate
$$\mathsf{rescue}_p(Z_i) = \mathsf{st}_i .$$
- Add an equality gate (c.f. Section 4.2.2) that $\mathsf{st}_\ell[0] = y$.

4.2.5 Building Block Circuits for Transactions

In this section, we instantiate the building block algorithms used in transaction proofs and specify the corresponding circuits. The syntax of the building block algorithms is specified in Section 3.2.1. In the following context, we denote as $\mathbb{F}_p$ the BLS-12-381 scalar field, $E(\mathbb{F}_p)$ the Jubjub subgroup, and $\mathbb{F}_r$ the scalar field of the Jubjub subgroup.

Range Checks. The range check gate $\mathsf{range}(v, \ell) = (b_0, \ldots, b_\ell)$ has been specified in Section 4.2.2.

Asset Type Derivation. In our instantiation, we define an asset type $\mathsf{at} \in \mathbb{F}_p$ to be
$$\mathsf{derive}_{\mathsf{at}}(s, \mathsf{aux}) := \mathsf{PRF}_s(\mathsf{aux}, 0, 0, 0)$$
where $s \in \mathbb{F}_p$ is a secret seed and $\mathsf{aux} \in \mathbb{F}_p$ is a digest of asset type descriptions.

Denote by $s, \mathsf{aux}, \mathsf{at} \in \mathbb{F}_p$ three variables and 0_{var} a default zero variable, the circuit for the asset type derivation algorithm is basically a PRF gadget $\mathsf{PRF}_s(\mathsf{aux}, 0_{\mathsf{var}}, 0_{\mathsf{var}}, 0_{\mathsf{var}}) = \mathsf{at}$ specified in Section 4.2.4.

User/Freezer Public Key Derivation. Let $B \in E(\mathbb{F}_p)$ be a base point of the Jubjub subgroup, we derive a user/freezer public key $\mathsf{pk} \in E(\mathbb{F}_p)$ from a user/freezer secret key $\mathsf{sk} \in \mathbb{F}_r$ as

$$\mathsf{derive}_{\mathsf{pk}}(\mathsf{sk}) := \mathsf{sk} \cdot B.$$

Note that in the circuit, sk is encoded as a scalar in $\mathbb{F}_p$.

Nullifier key Derivation. Given a secret key $\mathsf{sk}_1 \in \mathbb{F}_r$ and a public key $\mathsf{pk}_2 \in E(\mathbb{F}_p)$, we define the nullifier key $\mathsf{nk} := \mathsf{sk}_1$ if $\mathsf{pk}_2 = \infty = (0, 1)$; otherwise, we define it to be

$$\mathsf{derive}_{\mathsf{nk}}(\mathsf{sk}_1, \mathsf{pk}_2) := \mathsf{sponge}_1(\mathsf{affine}(\mathsf{sk}_1 \cdot \mathsf{pk}_2), 0)\,,$$

where $\mathsf{affine}(U) \in \mathbb{F}_p^2$ is the affine form of a Jubjub point $U \in E(\mathbb{F}_p)$.

To illustrate the intuition of the scheme, suppose sk_1 is a user secret key and pk_2 is a freezer public key. If pk_2 is a valid public key in the Jubjub subgroup, the user can use the Diffie-Hellman key exchange scheme and derive the secret nullifier key $\mathsf{nk} := \mathsf{sponge}_1(\mathsf{affine}(\mathsf{sk}_1 \cdot \mathsf{pk}_2), 0)$; the same holds true when sk_1 is a freezer secret key and pk_2 is a user public key. For the special case where $\mathsf{pk}_2 = \perp = \infty$ (e.g., the freezer public key is empty), the DH shared key becomes a fixed point and thus nk is no longer secret, that's why we need to redefine $\mathsf{nk} := \mathsf{sk}_1$ when pk_2 is a null value (i.e. $\mathsf{pk}_2 = \perp = \infty$).

In the circuit, we encode sk_1 as a scalar in $\mathbb{F}_p$, and represent Jubjub points in their affine form (i.e., $(x, y) \in \mathbb{F}_p^2$). Denote by sk_1, pk_2, nk the variables, the constraint system works as follows:

- Constrain $\mathsf{pk}^* = \mathsf{sk}_1 \cdot \mathsf{pk}_2$, where $\mathsf{pk}^* \in \mathbb{F}_p^2$ is the affine form of the Jubjub point $\mathsf{sk}_1 \cdot \mathsf{pk}_2$.
- Constrain $\mathsf{hk} = \mathsf{sponge}_1(\mathsf{pk}^*, 0_{\mathsf{var}})$, where $\mathsf{hk} \in \mathbb{F}_p$ is an intermediate variable.
- Constrain $b = \mathsf{neutral}(\mathsf{pk}_2)$, where the intermediate variable $b \in \{0, 1\}$ is used to indicate whether pk_2 is a neutral point.
- Constrain $(b\ ?\ \mathsf{hk} : \mathsf{sk}_1) = \mathsf{nk}$ where $(b\ ?\ x_0 : x_1) = x_b$ is a conditional selection gate (c.f. Section 4.2.2).

Asset Record Commitments. In our instantiation, the message vector committed in an ARC is the concatenation of an amount $v \in \mathbb{F}_p$, an asset type $\mathsf{at} \in \mathbb{F}_p$, a user address $\mathsf{upk} \in \mathbb{F}_p^2$, a freezing flag $b_{\mathsf{frz}} \in \{0, 1\}$, and a policy $\mathsf{policy} \in \mathbb{F}_p^7$ that consists of a tracer public key $\mathsf{tpk} \in \mathbb{F}_p^2$, an identity authority public key $\mathsf{ipk} \in \mathbb{F}_p^2$, a freezer public key $\mathsf{fpk} \in \mathbb{F}_p^2$, and a compressed reveal map $\mathsf{reveal} \in \mathbb{F}_p$. More precisely, the least-significant-form binary representation $(b_0, b_1, b_2, \mathsf{attrmap}) \in \{0, 1\}^{3+\ell_{\mathsf{attrs}}}$ of reveal is used to indicate whether to reveal upk, v, γ and each identity attribute respectively. To optimize the number of Rescue hashes invoked by the commitment, we combine reveal and b_{frz} into a single scalar $\mathsf{reveal}^* := 2 \cdot \mathsf{reveal} + b_{\mathsf{frz}} \in \mathbb{F}_p$. That is, the binary representation of reveal^* becomes $(b_{\mathsf{frz}}, b_0, b_1, b_2, \mathsf{attrmap})$.

Let $\gamma \in \mathbb{F}_p$ be a blinding factor and define the message vector

$$\boldsymbol{m} := (v, \mathsf{at}, \mathsf{upk}, \mathsf{tpk}, \mathsf{ipk}, \mathsf{fpk}, \mathsf{reveal}^*) \in \mathbb{F}_p^{11}\,,$$

the asset record commitment $\mathsf{arc} \in \mathbb{F}_p$ is defined as

$$\mathsf{Com}(\boldsymbol{m}; \gamma) := \mathsf{sponge}_4(\gamma, \boldsymbol{m})$$

where sponge_4 is a sponge-based hash (specified in Section 4.1.4) with input length $4 \cdot 3 = 12$.

Therefore, denote by $\mathsf{arc} \in \mathbb{F}_p$, $\gamma \in \mathbb{F}_p$ and

$$\boldsymbol{m} := (v, \mathsf{at}, \mathsf{upk}, \mathsf{tpk}, \mathsf{ipk}, \mathsf{fpk}, \mathsf{reveal}^*) \in \mathbb{F}_p^{11}$$

a list of variables, the circuit for the asset record commitment is basically a sponge-based hash gadget (c.f. Section 4.2.4)

$$\mathsf{sponge}_4(\gamma, \boldsymbol{m}) = \mathsf{arc}\,.$$

Nullifiers. In our instantiation, we define a nullifier $\mathsf{nl} \in \mathbb{F}_p$ to be

$$\mathsf{Nullify}_{\mathsf{nk}}(\mathsf{elem}) := \mathsf{PRF}_{\mathsf{usk}}(\mathsf{elem}, 0, 0)$$

where $\mathsf{nk} \in \mathbb{F}_p$ is a nullifying secret key, $\mathsf{elem} := (\mathsf{uid}, \mathsf{arc}) \in \mathbb{F}_p^2$ is an accumulated element where $\mathsf{uid} \in \mathbb{F}_p$ is an unique identifier and $\mathsf{arc} \in \mathbb{F}_p$ is an asset record commitment.

Denote by $\mathsf{nl}, \mathsf{nk}, \mathsf{uid}, \mathsf{arc} \in \mathbb{F}_p$ some variables and 0_{var} a default zero variable, the circuit for the nullifier computation is basically a PRF gadget

$$\mathsf{PRF}_{\mathsf{nk}}(\mathsf{uid}, \mathsf{arc}, 0_{\mathsf{var}}, 0_{\mathsf{var}}) = \mathsf{nl}$$

specified in Section 4.2.4.

Merkle Tree Membership Verification. Recall that in an anonymous transaction, the sender needs to prove that the input commitments to be spent are accumulated. More precisely, given the current Merkle tree root value $\mathsf{rt} \in \mathbb{F}_p$, one needs to prove knowledge of a unique identifier $\mathsf{uid} \in \mathbb{F}_p$, a commitment $\mathsf{arc} \in \mathbb{F}_p$, and a Merkle membership proof π, such that $\mathsf{Acc.vfy}(\mathsf{rt}, (\mathsf{uid}, \mathsf{arc}), \pi) = 1$, where $\mathsf{Acc.vfy}$ is the Merkle tree membership verification algorithm. In this section, we show how to build the verification circuit.

Before describing the circuit, we explain the format of the membership proof π as well as the building block circuits being used:

- Denote by d the depth of the 3-ary Merkle tree, the proof π consists of d tuples (where the first tuple represents the leaf), and the i-th ($1 \le i \le d$) tuple is with the form $(b_0, b_1, \mathsf{sib}_0, \mathsf{sib}_1) \in \{0,1\}^2 \times \mathbb{F}_p^2$, representing that the i-th node u is the $(1 + b_0 + 2b_1)$-th child of its parent (we require $b_0 + b_1 \le 1$), and sib_0, sib_1 are the labels of u's first and second sibling, respectively.
- The verification circuit makes use of a hash gate $\mathsf{rescue}_h : \mathbb{F}_p^3 \to \mathbb{F}_p$, which can be instantiated with a sponge-based hash gate sponge_1 described before.
- Besides, the verification circuit makes use of a sub-circuit called $\mathsf{sort} : \{0,1\}^2 \times \mathbb{F}_p^3 \to \mathbb{F}_p^3$. Given 3 variables (x_1, x_2, x_3) and two boolean variables (b_0, b_1), the circuit permutes (x_1, x_2, x_3) to a vector (y_1, y_2, y_3) so that x_1 is inserted to the position $1 + b_0 + 2b_1$. For example, if $1 + b_0 + 2b_1 = 2$ (i.e., $b_0 = 1$ and $b_1 = 0$), then $(y_1, y_2, y_3) = (x_2, x_1, x_3)$.
- Finally, we makes use of a non-zero gate that constrain a variable $x \in \mathbb{F}_p$ to be a non-zero value.

Next we specify the membership verification circuit and the instantiations of the sort gate and the non-zero gate. Given a public input variable rt, and a list of secret input variables

$$\left[\mathsf{uid}, \mathsf{arc}, \pi = \left(\left(b_0^{(1)}, b_1^{(1)}, \mathsf{sib}_0^{(1)}, \mathsf{sib}_1^{(1)}\right), \ldots, \left(b_0^{(d)}, b_1^{(d)}, \mathsf{sib}_0^{(d)}, \mathsf{sib}_1^{(d)}\right)\right)\right]$$

(where for all $i \in [d]$, it holds that $b_0^{(i)}, b_1^{(i)}$ are boolean and $b_0^{(i)} + b_1^{(i)} \le 1$), the verification circuit works as follows:

1. Add a Rescue hash gate
$$\mathsf{rescue}_h(0_{\mathsf{var}}, \mathsf{uid}, \mathsf{arc}) = v_1 \,,$$
where $v_1 \in \mathbb{F}_p$ is the leaf label of the Merkle path.
2. For i from 1 to d, add the following constraints:
   - Add a sort gate
   $$\mathsf{sort}\left(b_0^{(i)}, b_1^{(i)}, v_i, \mathsf{sib}_0^{(i)}, \mathsf{sib}_1^{(i)}\right) = \left(y_1^{(i)}, y_2^{(i)}, y_3^{(i)}\right) \,,$$
   where $(y_1^{(i)}, y_2^{(i)}, y_3^{(i)})$ are intermediate variables.
   - Constrain that the left child variable $y_1^{(i)}$ is non-zero.
   - Add a Rescue hash gate
   $$\mathsf{rescue}_h\left(y_1^{(i)}, y_2^{(i)}, y_3^{(i)}\right) = v_{i+1}$$
   where v_{i+1} is an intermediate variable.
3. Add an equality gate $v_{d+1} = \mathsf{rt}$.

The sort *circuit.* Given 3 variables (x_1, x_2, x_3) and two boolean variables (b_0, b_1) (such that $b_0 + b_1 \le 1$), the circuit constrains a vector of variables (y_1, y_2, y_3) to be a permutation of (x_1, x_2, x_3), such that x_1 equals $y_{1+b_0+2b_1}$, and the relative order of x_2 and x_3 does not change. Denote by $(b \ ? \ x_0 : x_1) = y$ a conditional selection gate (c.f. Section 4.2.2) where y is constrained to be x_b. The sort circuit works as follows:

1. Add a conditional selection gate $(b_0 \ ? \ x_2 : x_1) = y_2$.
2. Add a conditional selection gate $(b_1 \ ? \ x_3 : x_1) = y_3$.
3. Add an addition gate $y_2 + y_3 = y_{2,3}$ where $y_{2,3}$ is an intermediate variable.
4. Add a linear combination gate $x_1 + x_2 + x_3 - y_{2,3} = y_1$.

The circuit makes use of 2 selection gate, 1 addition gate, and 1 linear-combination gate. In total, the circuit requires 4 constraints.

The non-zero gate. To constrain that a variable $x \in \mathbb{F}_p$ is non-zero, we create an intermediate variable $\mathsf{inv} \in \mathbb{F}_p$ and add a multiplication gate that $x \cdot \mathsf{inv} = 1_{\mathsf{var}}$. When x is non-zero, we can set inv as the inverse of x so that the constraint is satisfied; when x is zero, the multiplication gate will never be satisfied.

Proof-of-Burn. As specified in Section 3.2.1, the proof-of-burn circuit is a combination of the circuits for public key derivation, nullifying key derivation, nullifier computation, commitment scheme, and accumulator verification, all of which have been specified above. Approximately, the circuit makes use of $d + 7$ Rescue hashes (where d is the depth of the Merkle tree), 1 fixed-base scalar multiplication, and 1 variable-base scalar multiplication.

Preserving Balance. Given a native asset type $\mathsf{at}^* \in \mathbb{F}_p$, a transfer asset type $\mathsf{at} \in \mathbb{F}_p$, transaction fee $\mathsf{fee} \in \mathbb{F}_p$, a list of input amounts $(v_i^{\mathsf{in}})_{i\in[n]} \in \mathbb{F}_p^n$, and a list of output amounts $(v_i^{\mathsf{out}})_{i\in[m]} \in \mathbb{F}_p^m$, recall that the balance preserving predicate $\mathsf{Balance}(\mathsf{at}^*, \mathsf{at}, \mathsf{fee}, (v_i^{\mathsf{in}})_{i\in[n]}, (v_i^{\mathsf{out}})_{i\in[m]})$ (c.f. Section 3.2.1) holds if *one* of the following conditions holds

- $\mathsf{at}^* \neq \mathsf{at}$ and $v_1^{\mathsf{in}} = v_1^{\mathsf{out}} + \mathsf{fee}$ and
$$\sum_{i=2}^{n} v_i^{\mathsf{in}} = \sum_{i=2}^{m} v_i^{\mathsf{out}}$$
- $\mathsf{at}^* = \mathsf{at}$ and
$$\sum_{i=1}^{n} v_i^{\mathsf{in}} = \sum_{i=1}^{m} v_i^{\mathsf{out}} + \mathsf{fee}$$

Denote by $\mathsf{at}, \mathsf{at}^*, \mathsf{fee} \in \mathbb{F}_p$, $(v_i^{\mathsf{in}})_{i\in[n]} \in \mathbb{F}_p^n$ and $(v_i^{\mathsf{out}})_{i\in[m]} \in \mathbb{F}_p^m$ a list of variables, and $\Delta_{\mathsf{at}}, \Delta_{\mathsf{at}^*} \in \mathbb{F}_p$ two intermediate variables, the constraint system for the balance preserving predicate is implemented as follows:

1. Enforce $\Delta_{\mathsf{at}} = \sum_{i=2}^{n} v_i^{\mathsf{in}} - (\sum_{i=2}^{m} v_i^{\mathsf{out}})$. This can be implemented by adding the following gates:
   - A sum gate (c.f. Section 4.2.2) $\sum_{i=2}^{n} v_i^{\mathsf{in}} = x$ where x is an intermediate variable.
   - A sum gate $\sum_{i=2}^{m} v_i^{\mathsf{out}} = y$ where y is an intermediate variable.
   - A subtraction gate $x - y = \Delta_{\mathsf{at}}$.
2. Enforce $v_1^{\mathsf{in}} - v_1^{\mathsf{out}} - \mathsf{fee} = \Delta_{\mathsf{at}^*}$ via a linear combination gate.
3. Add a match gate (c.f. Section 4.2.2)
$$\mathsf{match}(\mathsf{at}, \mathsf{at}^*) = b_{\mathsf{match}}$$
where b_{match} is enforced to be 1 when $\mathsf{at} = \mathsf{at}^*$, and 0 otherwise.
4. Add a constraint
$$\boldsymbol{q}_{M_{1,2}} \cdot b_{\mathsf{match}} \cdot \Delta_{\mathsf{at}} + \boldsymbol{q}_{M_{3,4}} \cdot b_{\mathsf{match}} \cdot \Delta_{\mathsf{at}^*} = \boldsymbol{q}_O \cdot 0_{\mathsf{var}}$$
where $\boldsymbol{q}_{M_{1,2}} = \boldsymbol{q}_{M_{3,4}} = \boldsymbol{q}_O = 1$. Note that the equation above is equivalent to
$$b_{\mathsf{match}} \cdot (\Delta_{\mathsf{at}} + \Delta_{\mathsf{at}^*}) = 0\,,$$
hence it constrains that $\Delta_{\mathsf{at}} + \Delta_{\mathsf{at}^*} = 0$ whenever $b_{\mathsf{match}} = 1$ (i.e., $\mathsf{at} = \mathsf{at}^*$).
5. Add a multiplication constraint
$$b_{\mathsf{match}} \cdot \Delta_{\mathsf{at}} = \Delta_{\mathsf{at}}\,,$$
which enforces $\Delta_{\mathsf{at}} = 0$ whenever $b_{\mathsf{match}} = 0$ (i.e., $\mathsf{at} \neq \mathsf{at}^*$).
6. Add a multiplication constraint
$$b_{\mathsf{match}} \cdot \Delta_{\mathsf{at}^*} = \Delta_{\mathsf{at}^*}\,,$$
which enforces $\Delta_{\mathsf{at}^*} = 0$ whenever $b_{\mathsf{match}} = 0$ (i.e., $\mathsf{at} \neq \mathsf{at}^*$).

El-Gamal Public Encryption. We instantiate the El-Gamal hybrid encryption circuit as follows: Let ℓ be the message length. Define the following variables:

- $r \in \mathbb{F}_p$ is a variable representing encryption randomness.
- $R \in \mathbb{F}_p^2$ is a point variable that is a part of the ciphertexts.
- $\mathsf{PK} \in \mathbb{F}_p^2$ is a point variable representing the public key.
- $\boldsymbol{k} \in \mathbb{F}_p^4$ is a Rescue state variable representing a symmetric encryption key.
- $\boldsymbol{m} \in \mathbb{F}_p^\ell$ are ℓ variables representing the plaintexts.
- $\boldsymbol{c} \in \mathbb{F}_p^\ell$ are ℓ variables representing the symmetric ciphertexts.

Denote by B a fixed base point and 0_{var} the default zero variable, the hybrid encryption circuit $\mathsf{Enc}_{\mathsf{PK}}(\boldsymbol{m}; r) = (R, \boldsymbol{c})$ is implemented as follows:

- Add a fixed-base scalar multiplication gate (c.f. Section 4.2.3) $r \cdot B = R$.
- Add a variable-base scalar multiplication gate (c.f. Section 4.2.3) $r \cdot \mathsf{PK} = U$ where $U \in \mathbb{F}_p^2$ is an intermediate point variable and we refer to $(x_U, y_U) \in \mathbb{F}_p^2$ as the affine form of U.
- Add a Rescue permutation gate (c.f. Section 4.2.4)

$$\mathsf{rescue}_p(x_U, y_U, 0_{\mathsf{var}}, 0_{\mathsf{var}}) = \boldsymbol{k}\,.$$

- Add a counter mode encryption gate

$$\mathsf{rescue}_{\mathsf{ctr}}(\boldsymbol{k}; 0_{\mathsf{var}}, \boldsymbol{m}) = \boldsymbol{c}\,.$$

Note that the public key PK corresponding to the tracer (i.e $\mathsf{PK} = \mathsf{tpk}$) is expected to be well formed (i.e. PK is a valid point on the Jubjub curve), as it is related to a policy specified and validated when issuing a new asset (see Section 3.2.2). The above circuit makes use of 1 fixed-base scalar multiplication gate, 1 variable-base scalar multiplication gate, 1 Rescue permutation gate, and 1 counter mode encryption gate with message length ℓ.

Credential Verification. We instantiate the credential scheme with Schnorr signatures (c.f. Algorithm 1) where the underlying hash function is a sponge-based hash from Rescue permutations (c.f. Section 4.2.4). We stress that the challenge in the underlying Sigma protocol is a truncated hash output so that it can be easily encoded as a Jubjub scalar.

We define the following variables. Denote as $\mathsf{ipk} \in E(\mathbb{F}_p) \subseteq \mathbb{F}_p^2$ a point variable that represents an identity authority public key and $\boldsymbol{m} \in \mathbb{F}_p^\ell$ a message that maps to ℓ variables. Let $c \in [0, 2^{\ell_c})$ be a challenge variable where ℓ_c is the bit-length of challenge (e.g., $\ell_c = 250$). Let $\mathsf{cred} := (R, s) \in E(\mathbb{F}_p) \times \mathbb{F}_r$ be a credential where R is a point variable and $s \in \mathbb{F}_r$ represents a Jubjub scalar. We note that the public key ipk is expected to be well formed (i.e. ipk is a valid point on the Jubjub curve), as it is related to a policy specified and validated when issuing a new asset (see Section 3.2.2). The circuit for credential verification $\mathsf{Vfy}_{\mathsf{ac}}(\mathsf{ipk}, \mathsf{cred}, \boldsymbol{m})$ is implemented as follows:

1. Add a sponge-based hash gate[13] (c.f. Section 4.2.4)

$$\mathsf{sponge}_{(4+\ell)/3}(\mathsf{ipk}, R, \boldsymbol{m}) = h$$

   where $h \in \mathbb{F}_p$ is an intermediate variable.
2. Add a truncating gate

$$\mathsf{truncate}_{\ell_c}(h) = c$$

   that enforces c to be $h \bmod 2^{\ell_c}$. The truncating gate can be instantiated via a range-check gate. We stress that 2^{ℓ_c} should be less than the size of the Jubjub scalar field.
3. Add a variable-base scalar multiplication gate (c.f. Section 4.2.3)

$$c \cdot \mathsf{ipk} = Z$$

   where Z is an intermediate point variable.
4. Add a point addition gate $R + Z = U_1$ where U_1 is an intermediate point variable.
5. Add a fixed-base scalar multiplication gate (c.f. Section 4.2.3)

$$s \cdot B = U_2$$

   where B is a fixed Jubjub base point and U_2 is an intermediate variable.
6. Add an equality gate $U_1 = U_2$.

The circuit size is dominated by a fixed-base scalar multiplication, a variable-base scalar multiplication, and a sponge-based hash that makes use of $(4 + \ell)/3$ Rescue permutation gates.

[13] With out loss of generality we assume that the sponge input length $4 + \ell$ is a multiple of 3.

References

AABS+19. Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. 2019. https://eprint.iacr.org/2019/426. 4.1.3, 4.1.3

BBDP01. Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 566–582, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 3.1, A, 8

BDFG20a. Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Efficient polynomial commitment schemes for multiple points and polynomials. 2020. https://eprint.iacr.org/2020/081. 4.1.9, 4.1.9

BDFG20b. Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Recursive zk-snarks from any additive polynomial commitment scheme. Cryptology ePrint Archive, Report 2020/1536, 2020. https://eprint.iacr.org/2020/1536. 2

BDPVA07. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer, 2007. 4.1.4

Bow17. Sean Bowe. Bls12-381: New zk-snark elliptic curve construction. 2017. https://electriccoin.co/blog/new-snark-curve/. 4.1.1

CGN20. Konstantinos Chalkias, Franccois Garillot, and Valeria Nikolaenko. Taming the many eddsas. Cryptology ePrint Archive, Report 2020/1244, 2020. 4.1.2

Gri18. Jack Grigg. Bls12-381 (zcash specification). 2018. https://github.com/zcash/librustzcash/blob/6e0364cd42a2b3d2b958a54771ef51a8db79dd29/pairing/src/bls12_381/README.md. 4.1.1

Gro16. Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 2.1

GWC19. Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, 2019:953, 2019. 4.2, 4.2.1, 4.2.1

Hop17. Daira Hopwood. Jubjub supporting evidence. 2017. https://github.com/daira/jubjub. 4.1.1

KZG10. Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International conference on the theory and application of cryptology and information security*, pages 177–194. Springer, 2010. 3.1, 4.1.9

MRV15. Bart Mennink, Reza Reyhanitabar, and Damian Vizár. Security of full-state keyed sponge and duplex: Applications to authenticated encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 465–489. Springer, 2015. 4.1.5

NIS15. NIST. Secure hash standard (shs). 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf. 4.1.4

SCG+14. Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014. 2, 2.1

ZCa17. ZCash. What is jubjub? 2017. https://z.cash/technology/jubjub/. 4.1.1, 4.2.3

A Cryptographic Primitives: Definitions

Definition 2. *A cryptographic hash function $\mathcal{H} = (\mathsf{Gen}_{\mathsf{hash}}, \mathsf{H})$ is **collision resistant** if for for all polynomial-time adversaries $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$, such that:*

$$\Pr\left[\mathsf{H}^{(s)}(m) = \mathsf{H}^{(s)}(m') \wedge m \neq m' \;\middle|\; \begin{array}{l} s \xleftarrow{\$} \mathsf{Gen}_{\mathsf{hash}}(1^\lambda) \\ (m, m') \leftarrow \mathcal{A}(s) \end{array}\right] \leq \mu(\lambda)$$

Definition 3. *A deterministic algorithm F is a **pseudo-random function (PRF)** defined over $(\mathcal{K}, \mathcal{X}, \mathcal{Y})$, where $\mathcal{K}$ is the key space, $\mathcal{X}$ is the input space, $\mathcal{Y}$ is the output space, if it is:*

- efficiently computable*: on key $k \in \mathcal{K}$ and input $x \in \mathcal{X}$, it outputs $y = F(k, x) \in \mathcal{Y}$ in polynomial time.*
- pseudo-random*: for all polynomial-time adversaries $\mathcal{A}$, the probability of distinguishing F from a random function drawn from $\mathrm{Funs}[\mathcal{X}, \mathcal{Y}]$, denoting the set of all functions $f : \mathcal{X} \to \mathcal{Y}$, is negligible w.r.t the security parameter λ. Equivalently,*

$$\left|\Pr\left[\mathcal{A}^{F(k,\cdot)}(1^\lambda) = 1 \;\middle|\; k \xleftarrow{\$} \mathcal{K}\right] - \Pr\left[\mathcal{A}^{f(\cdot)}(1^\lambda) = 1 \;\middle|\; f \xleftarrow{\$} \mathrm{Funs}[\mathcal{X}, \mathcal{Y}]\right]\right| \leq \mu(\lambda)$$

Now, we move on defining another primitive called *pseudo-random permutation* in a similar fashion as a pseudo-random function. Informally, it's basically a PRF whose input space and output space is the same ($\mathcal{X} = \mathcal{Y}$).

Definition 4. *A deterministic algorithm π is a **pseudo-random permutation (PRP)** defined over $(\mathcal{K}, \mathcal{X})$, where $\mathcal{K}$ is the key space, $\mathcal{X}$ is the input space and the output space, if it is:*

- efficiently computable*: on key $k \in \mathcal{K}$, input $x \in \mathcal{X}$, it computes $y = \pi_k(x) \in \mathcal{X}$ in polynomial time.*
- efficiently invertible*: on key $k \in \mathcal{K}$, an image $y \in \mathcal{X}$, it computes the pre-image $x = \pi_k^{-1}(y)$ in polynomial time.*
- pseudo-random*: for a permutation P randomly drawn from $\mathrm{Perms}[\mathcal{X}]$, denoting the set of all permutation $f : \mathcal{X} \to \mathcal{X}$, and for all polynomial-time adversaries $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$, such that:*

$$\left|\Pr\left[\mathcal{A}^{\pi_k(\cdot)}(1^\lambda) = 1 \;\middle|\; k \xleftarrow{\$} \mathcal{K}\right] - \Pr\left[\mathcal{A}^{P(\cdot)}(1^\lambda) = 1 \;\middle|\; P \xleftarrow{\$} \mathrm{Perms}[\mathcal{X}]\right]\right| \leq \mu(\lambda)$$

Extending from semantic security of a pseudo-random permutation in Definition 4, we define a stronger property to account for security concerns when an honest parties are require to compute $\pi_k^{-1}(x)$ in addition to computing the $\pi_k(x)$ itself.

Definition 5. *An efficient, keyed permutation $\pi_k : \{0,1\}^\ell \to \{0,1\}^\ell$ is a **strong pseudo-random permutation** if for all polynomial-time adversaries $\mathcal{A}$ who are given a permutation oracle and a permutation inversion oracle, there exists a negligible function $\mu(\cdot)$, such that:*

$$\left|\Pr\left[\mathcal{A}^{\pi_k(\cdot), \pi_k^{-1}(\cdot)}(1^\lambda) = 1 \;\middle|\; k \xleftarrow{\$} \mathcal{K}\right] - \Pr\left[\mathcal{A}^{P(\cdot), P^{-1}(\cdot)}(1^\lambda) = 1 \;\middle|\; P \xleftarrow{\$} \mathrm{Perms}[\mathcal{X}]\right]\right| \leq \mu(\lambda)$$

Definition 6. *A signature scheme has **existential unforgeability against chosen message attack** (*UF-CMA *security) if for all polynomial-time adversaries $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$, such that:*

$$\Pr\left[\begin{array}{l} \mathsf{Vfy}(pk_{\mathsf{sig}}, m, \sigma) = 1 \\ \wedge\, m \notin \{m_i\}_{i \in [Q]} \end{array} \;\middle|\; \begin{array}{l} (pk_{\mathsf{sig}}, sk_{\mathsf{sig}}) \xleftarrow{\$} \mathsf{Gen}_{\mathsf{sig}}(1^\lambda) \\ (m, \sigma) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{Sign}}}(1^\lambda, pk_{\mathsf{sig}}) \end{array}\right] \leq \mu(\lambda) \quad \left|\; \begin{array}{l} \textbf{\textit{Oracle}}\ \mathcal{O}_{\mathsf{Sign}}(\{m_i\}_{i \in [Q]}) : \\ \quad \textit{for}\ i \in [Q] : \\ \quad\quad \sigma_i \leftarrow \mathsf{Sign}(sk_{\mathsf{sig}}, m_i) \\ \quad \textbf{\textit{return}}\ \{\sigma_i\}_{i \in [Q]} \end{array}\right.$$

where $\mathcal{O}_{\mathsf{Sign}}$ is a signing oracle that can be queried with at most Q messages and Q is poly-bounded w.r.t. to the security parameter.

Definition 7. *A public-encryption scheme has **ciphertext indistinguishability under chosen-plaintext attack**(*IND-CPA *security) if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$, such that:*

$$\Pr\left[b = \hat{b} \;\middle|\; \begin{array}{c} (pk_{\mathsf{enc}}, sk_{\mathsf{enc}}) \xleftarrow{\$} \mathsf{Gen}_{\mathsf{enc}}(1^\lambda), b \xleftarrow{\$} \{0,1\} \\ \hat{b} \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{Enc}}}(1^\lambda, pk_{\mathsf{enc}}) \end{array}\right] \leq \mu(\lambda) \quad \left|\; \begin{array}{l} \textbf{\textit{Oracle}}\ \mathcal{O}_{\mathsf{Enc}}(m_0, m_1) : \\ \quad \textbf{\textit{return}}\ \mathsf{Enc}(pk_{\mathsf{enc}}, m_b) \end{array}\right.$$

Beyond the security property above, we further introduces another property called IK-CCA **security** [BBDP01] that ensures the anonymity of the keys being used during encryption, thus protecting the identity of the encrypting party.

Similarly, we formalize it by first describing an attack game in which the adversary is challenged to determine which particular key from a group of known public keys is used to produces the ciphertext. The attack game also proceeds in 2 phases, and the adversary $\mathcal{A}$ is given two decryption oracles $\mathcal{O}_{\mathsf{Dec}} = (\mathcal{O}_{\mathsf{Dec0}}, \mathcal{O}_{\mathsf{Dec1}})$, concretely:

- An IK-CCA adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ is a pair of algorithms such that $\mathcal{A}_1$ can access two decryption oracles $\mathcal{O}_{\mathsf{Dec}}$ before outputting a tuple (m, s) during phase 1 ("find phase") of the attack game where s is some state information;
- An encryption query is then immediately submitted to the challenger who will output a ciphertext ct of the message m under one of two public keys pk_{enc_0} and pk_{enc_1};
- Finally, given access to the same pair of decryption oracles $\mathcal{O}_{\mathsf{Dec}}$, and on input (ct, s), the algorithm $\mathcal{A}_2$ will output a bit $\hat{b}$ during phase 2 ("guess phase") of the attack game. Noted that $\mathcal{A}_2$ has to meet the **"no-challenge-decryption" condition**, namely it can't query ct to either oracle ($\mathsf{ct} \notin S_0 \wedge \mathsf{ct} \notin S_1$).

Informally, a public-key encryption scheme that is IK-CCA secure, should limits any efficient adversaries $\mathcal{A}$ to a negligible advantage at winning the attack game. Formally:

Definition 8. *A public-encryption scheme is **key indistinguishable under chosen ciphertext attack** (IK-CCA security [BBDP01]) if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$, such that:*

$$\Pr\left[b = \hat{b} \;\middle|\; \begin{array}{r} (pk_{\mathsf{enc}_0}, sk_{\mathsf{enc}_0}) \xleftarrow{\$} \mathsf{Gen}_{\mathsf{enc}}(1^\lambda), S_0, S_1 \leftarrow \emptyset \\ (pk_{\mathsf{enc}_1}, sk_{\mathsf{enc}_1}) \xleftarrow{\$} \mathsf{Gen}_{\mathsf{enc}}(1^\lambda), b \xleftarrow{\$} \{0,1\} \\ (m, s) \xleftarrow{\$} \mathcal{A}_1^{\mathcal{O}_{\mathsf{Dec0}}, \mathcal{O}_{\mathsf{Dec1}}}(1^\lambda, pk_{\mathsf{enc}_0}, pk_{\mathsf{enc}_1}) \\ \mathsf{ct} \xleftarrow{\$} \mathsf{Enc}(pk_{\mathsf{enc}_b}, m) \\ \hat{b} \xleftarrow{\$} \mathcal{A}_2^{\mathcal{O}_{\mathsf{Dec0}}, \mathcal{O}_{\mathsf{Dec1}}}(\mathsf{ct}, s) \end{array} \right] \leq \mu(\lambda)$$

$$\left| \begin{array}{l} \textbf{\textit{Oracle}}\ \mathcal{O}_{\mathsf{Dec0}}(\mathsf{ct}): \\ \quad S_0 \leftarrow S_0 \cup \{\mathsf{ct}\} \\ \quad \textbf{\textit{return}}\ \mathsf{Dec}(sk_{\mathsf{enc}_0}, \mathsf{ct}) \\ \textbf{\textit{Oracle}}\ \mathcal{O}_{\mathsf{Dec1}}(\mathsf{ct}): \\ \quad S_1 \leftarrow S_1 \cup \{\mathsf{ct}\} \\ \quad \textbf{\textit{return}}\ \mathsf{Dec}(sk_{\mathsf{enc}_1}, \mathsf{ct}) \end{array} \right.$$

Definition 9. *A commitment scheme $\mathcal{C} = (\mathsf{Setup}_{\mathsf{com}}, \mathsf{Com}, \mathsf{Vfy}_{\mathsf{com}})$ is **computationally binding** if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$ such that:*

$$\Pr\left[\mathsf{Com}(\mathsf{pp}, x_0; r_0) = \mathsf{Com}(\mathsf{pp}, x_1; r_1) \wedge x_0 \neq x_1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^\lambda); \\ x_0, x_1, r_0, r_1 \leftarrow \mathcal{A}(\mathsf{pp}) \end{array} \right] \leq \mu(\lambda)$$

if $\mu(\lambda) = 0$, then we say the scheme is perfectly binding

Definition 10. *A commitment scheme $\mathcal{C} = (\mathsf{Setup}_{\mathsf{com}}, \mathsf{Com}, \mathsf{Vfy}_{\mathsf{com}})$ is **statistically hiding** if for all unbounded adversaries $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$ such that:*

$$\left| \Pr\left[b = \hat{b} \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^\lambda); b \xleftarrow{\$} \{0,1\}, r \xleftarrow{\$} \mathcal{R}_{\mathsf{pp}}, \\ (x_0, x_1) \in \mathcal{M}^2_{\mathsf{pp}} \leftarrow \mathcal{A}(\mathsf{pp}) \\ \mathsf{cm} \leftarrow \mathsf{Com}(\mathsf{pp}, x_b; r) \\ \hat{b} \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{cm}) \end{array} \right] - \frac{1}{2} \right| \leq \mu(\lambda)$$

if $\mu(\lambda) = 0$, then we say the scheme is perfectly hiding.

Definition 11. *A PCS is **correct** if for all $f \in \mathbb{F}^{\leq d}[X]$ and all $z \in \mathbb{F}$,*

$$\Pr\left[\begin{array}{l} \mathsf{Vfy}_{\mathsf{pc}}(\mathsf{pp}, f, o_{\mathsf{pc}}, \mathsf{cm}_{\mathsf{pc}}) = 1 \\ \wedge\ \mathsf{Eval}_{\mathsf{pc}}(\mathcal{P}(f, o_{\mathsf{pc}}), \mathcal{V}(\mathsf{pp}, \mathsf{cm}_{\mathsf{pc}}, z, y)) = 1 \end{array} \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}_{\mathsf{pc}}(1^\lambda, d) \\ (\mathsf{cm}_{\mathsf{pc}}, o_{\mathsf{pc}}) \leftarrow \mathsf{Com}_{\mathsf{pc}}(\mathsf{pp}, f) \end{array} \right] = 1$$

Firstly, we extend the *binding* property from a general commitment scheme to PCS:

Definition 12. *A PCS is* ***binding*** *if for all efficient (p.p.t.) adversaries $\mathcal{A}$, there exists a negligible function $\mu(\cdot)$ such that:*

$$\Pr\left[b = b' = 1 \wedge f' \neq f \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}_{\mathsf{pc}}(1^\lambda, d) \\ (f, f', \mathsf{cm}, o, o') \leftarrow \mathcal{A}(\mathsf{pp}) \\ b \leftarrow \mathsf{Vfy}_{\mathsf{pc}}(f, \mathsf{cm}, o) \\ b' \leftarrow \mathsf{Vfy}_{\mathsf{pc}}(f', \mathsf{cm}, o') \end{array} \right] \leq \mu(\lambda)$$

Next, we extend the *knowledge soundness* property from a *proof of knowledge* which is essentially what $\mathsf{Eval}_{\mathsf{pc}}$ protocol is to PCS:

Definition 13. *A PCS has* ***knowledge soundness*** *if for all $d \in \mathbb{N}$, the interactive public-coin protocol $\mathsf{Eval}_{\mathsf{pc}}$ is a proof of knowledge. Specifically, for all $d \in \mathbb{N}$, all $f \in \mathbb{F}^{\leq d}[X], z \in \mathbb{F}, y \in \mathcal{C}$, and all efficient cheating prover $\mathcal{A}$, there exists an efficient extractor $\mathcal{E}$ who is given oracle access to $\mathcal{A}$ and a negligible function $\mu(\cdot)$ such that:*

$$\Pr\left[b = 1 \implies \hat{f}(z) = y \wedge deg(\hat{f}) < d \;\middle|\; \begin{array}{r} \mathsf{pp} \leftarrow \mathsf{Setup}_{\mathsf{pc}}(1^\lambda, d) \\ (\mathsf{cm}_{\mathsf{pc}}, o_{\mathsf{pc}}) \leftarrow \mathsf{Com}_{\mathsf{pc}}(\mathsf{pp}, f) \\ b \leftarrow \mathsf{Eval}_{\mathsf{pc}}(\mathcal{A}(\cdot), \mathcal{V}(\mathsf{pp}, \mathsf{cm}_{\mathsf{pc}}, z, y)) \\ \hat{f} \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{pp}, \mathsf{cm}_{\mathsf{pc}}, z, y) \end{array} \right] \geq 1 - \mu(\lambda)$$

Definition 14. *A tuple of efficient algorithms $(\mathcal{G}, \mathcal{I}, \mathcal{P}, \mathcal{V})$ is a* ***pre-processing zk-SNARK with universal SRS*** *if the following properties hold:*

- ***Completeness.*** *For all $N \in \mathbb{N}$ and all efficient adversaries $\mathcal{A}$:*

$$\Pr\left[\begin{array}{r} (i, x, w) \notin \mathcal{R}_N \\ \vee\ \mathcal{V}(\tau, x, \pi) = 1 \end{array} \;\middle|\; \begin{array}{r} \mathsf{srs} \xleftarrow{\$} \mathcal{G}(1^\lambda, N) \\ (i, x, w) \leftarrow \mathcal{A}(\mathsf{srs}) \\ (\sigma, \tau) \xleftarrow{\$} \mathcal{I}^{\mathsf{srs}}(i) \\ \pi \xleftarrow{\$} \mathcal{P}(\sigma, x, w) \end{array} \right] = 1$$

- ***Adaptive Knowledge Soundness.*** *For all $N \in \mathbb{N}$, all efficient adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists a knowledge extractor $\mathcal{E}^{\mathcal{A}}$ with oracle access to $\mathcal{A}$, such that:*

$$\Pr\left[\begin{array}{r} (i, x, w) \notin \mathcal{R}_N \\ \wedge\ \mathcal{V}(\tau, x, \pi) = 1 \end{array} \;\middle|\; \begin{array}{r} \mathsf{srs} \xleftarrow{\$} \mathcal{G}(1^\lambda, N) \\ (i, x, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{srs}) \\ (\sigma, \tau) \xleftarrow{\$} \mathcal{I}^{\mathsf{srs}}(i) \\ \pi \leftarrow \mathcal{A}_2(\mathsf{st}) \\ w \leftarrow \mathcal{E}^{\mathcal{A}}(\mathsf{srs}, i, x, \sigma, \tau) \end{array} \right] \leq \mu(\lambda)$$

- ***Statistical Zero-knowledge.*** *For all $N \in \mathbb{N}$, all unbounded adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, there exists an efficient simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ and a negligible function $\mu(\cdot)$ such that:*

$$\left| \Pr\left[\begin{array}{r} (i, x, w) \in \mathcal{R}_N \\ \wedge\ \mathcal{A}_2(\mathsf{st}, \pi) = 1 \end{array} \;\middle|\; \begin{array}{r} \mathsf{srs} \xleftarrow{\$} \mathcal{G}(1^\lambda, N) \\ (i, x, w, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{srs}) \\ (\sigma, \tau) \xleftarrow{\$} \mathcal{I}^{\mathsf{srs}}(i) \\ \pi \leftarrow \mathcal{P}(\sigma, x, w) \end{array} \right] - \Pr\left[\begin{array}{r} (i, x, w) \in \mathcal{R}_N \\ \wedge\ \mathcal{A}_2(\mathsf{st}, \pi) = 1 \end{array} \;\middle|\; \begin{array}{r} (\mathsf{srs}, \mathsf{trap}) \xleftarrow{\$} \mathcal{S}_1(1^\lambda, N) \\ (i, x, w, \mathsf{st}) \leftarrow \mathcal{A}_1(\mathsf{srs}) \\ \pi \leftarrow \mathcal{S}_2(\mathsf{trap}, i, x) \end{array} \right] \right| \leq \mu(\lambda)$$

- ***Efficiency.*** *There exists a universal polynomial* poly *(independent of relation $\mathcal{R}_N$) such that:*
  - *The indexer $\mathcal{I}$ running time is bounded by $poly_\lambda(|i|)$, namely independent of the size of the SRS and only polynomial in terms of the circuit size.*
  - *The proof size $|\pi|$ is bounded by* $\mathsf{poly}(\lambda)$.
  - *The verifier $\mathcal{V}$ running time is bounded by* $\mathsf{poly}(\lambda + |x|)$.

B Configurable Asset Policy: Completeness and Security

C Merkle tree security

Definition 15. *(Syntax) Let* $\mathsf{H} : \mathbb{F}_p \times \mathbb{F}_p \times \mathbb{F}_p \to \mathbb{F}_p$ *be a hash function sampled from a hash function family* $\mathcal{H}$*. We define a Merkle scheme* MT *as a set of PPT algorithms:*

- $\mathsf{MTGen}(h) \to \mathsf{st}$*: takes the height of the tree and returns the state* st *corresponding to an empty set.*
- $\mathsf{MTInsert}(\mathsf{st}, \mathsf{elem}) \to \mathsf{st}'$*: takes as parameter a Merkle tree state* st*, an element* elem *and returns another state* st' *corresponding to the insertion of the element* elem*.*
- $\mathsf{MTProve}(\mathsf{st}, pos) \to \pi$*: takes as parameter a state* st*, a leaf position pos and returns a membership proof for the leaf* π*.*
- $\mathsf{MTVerify}(\mathsf{rt}, \mathsf{elem}, \pi) \to \{0, 1\}$ *: takes as parameter a root value* rt*, an element* elem*, a membership proof* π *and returns* 1 *if the verification is successful and* 0 *otherwise.*

Definition 16. *A Merkle tree scheme* MT *is correct if the probability to yield an invalid witness after a polynomial number of insertions is negligible. More concretely, let* $L = (\mathsf{elem}_0, \mathsf{elem}_1, \cdots, \mathsf{elem}_n)$ *a list of elements where* n *is a polynomial in the security parameter* 1^λ*, and* st *the last state obtained after insertion of the* n *elements in an initial empty tree. Then the scheme is correct if for any* $i : 0 \leq i \leq n$ *the probability that* $\mathsf{MTVerify}(\mathsf{st}.root.val, \mathsf{elem}_i, \pi_i) = 0$ *with* $\pi_i = \mathsf{MTProve}(\mathsf{st}, i)$ *is negligible in* 1^λ*.*

Lemma 1. *An honestly computed Merkle tree* $\mathcal{T}$ *that contains at least one element, is such that for all the internal nodes* N*, the left child* $N.left$ *is a leaf or an internal node.*

Proof. By recurrence. The property of the lemma is true for $n = 1$ inserted elements. Assume that the property holds for n. If we insert another element in the tree, then the new internal nodes are indeed such that their left child is either another internal node or a leaf (and the middle,right children are empty nodes). Thus the property is satisfied for $n + 1$.

Definition 17. *(Security) We define the security of a Merkle tree scheme* MT*, by the following experiment: Let* $\mathcal{A}$ *be a PPT adversary and* $\mathcal{O}$ *be an oracle for the Merkle tree. At the beginning of the experiment we start with an empty Merkle tree state. The adversary then inserts a polynomial number of elements by forwarding them to the oracle that updates the state. Let* S *be the set of inserted elements at the end of the experiment, and* st *the last version of the state computed by* $\mathcal{O}$*. The adversary* $\mathcal{A}$ *wins if he can output an element* elem^* *and a proof* π^* *such that* $\mathsf{elem}^* \notin S$ *and* $\mathsf{MTVerify}(\mathsf{st}.root.val, \mathsf{elem}^*, \pi^*) = 1$*.*

Proposition 1. *If no PPT adversary can invert* H *on* 0*, the construction is correct.*

Proof. The only reason why an honestly computed Merkle tree could be invalid (i.e. does not allow to compute a valid proof for an inserted element) is in the case an internal node happens to have a left child value equal to 0. Due to Lemma 1, an internal node is such that its left child N_L has value

- $\mathsf{H}(a, b, c)$ if N_L is an internal node.
- $\mathsf{H}(0, b, c)$ if N_L is a leaf.

In both cases, if $N_L.val = 0$ then we obtain a preimage on 0 which can only occur with negligible probability as we have just described a specific PPT adversary.

Proposition 2. *If the hash function* H *is such that* $\mathcal{H}$ *described in Definition 15 is collision-resistant and no PPT adversary can invert* H *on the output* 0*, then the Merkle tree construction described in Section 4.1.8 is secure according to Definition 17.*

Proof. Let S be the set of elements inserted into the Merkle tree at the end of the experiment. Assume an adversary manages to obtain a valid witness π^* for an element $x \notin S$. Let T^* be the list of node values from the leaf to the root reconstructed through π^* and the leaf L^* provided by the adversary to MTVerify. The root rt^* obtained must be equal to the root of the honestly computed tree rt as otherwise the verification fails. Let $\mathcal{T}$ be the tree obtained at the end of the experiment.

For a node N we define its position by its level m and its location from left to right n inside the level. For an honestly computed tree, a position (m, n) is valid if a node in such position has been created through MTInsert, otherwise the position is said invalid.

If all the positions of the nodes corresponding to the values of T^* are valid then a collision for H can be obtained by comparing the nodes of T^* and the corresponding nodes in the tree $\mathcal{T}$

until finding two nodes N and N^* in $\mathcal{T}$ and T^* respectively such that $N.val = N^*.val$ and (a) $[N.left.val, N.middle.val, N.right.val] \neq [N^*.left.val, N^*.middle.val, N^*.right.val]$, in the case of internal nodes, (b) $N.val = \mathsf{H}(0, x, y)$ and $N^*.val = \mathsf{H}(0, x^*, y^*)$ with $(x, y) \neq (x^*, y^*)$, in the case of leaf nodes. Note that this must happen as if all the values were equal along this path, then this would mean that $x \in S$.

In the other case, let us consider $\tilde{T^*}$, the longest prefix of T^* such that all the values contained in $\tilde{T^*}$ correspond to valid positions. Note that $\tilde{T^*}$ has length greater or equal to 1 as it contains at least the root node.

First, in the case the values in $\tilde{T^*}$ are not equal to the corresponding $\mathcal{T}$ node values at the same position in the tree, then as observed above, we can obtain a collision for H. So let us assume that the values in $\tilde{T^*}$ match the values in the tree $\mathcal{T}$ for each respective position.

We can now consider three subcases:

1. The last value of $\tilde{T^*}$ corresponds to an empty node in $\mathcal{T}$. By construction such a node has value 0. In order to extend the tree from this node, the adversary needs to compute a^*, b^*, c^* such that $\mathsf{H}(a^*, b^*, c^*) = 0$ breaking the assumption about the non-invertibility of H on 0.
2. The last value of $\tilde{T^*}$ corresponds to a leaf node in $\mathcal{T}$. By construction such a node has value $\mathsf{H}(0, x, y)$ for some x, y. The verification algorithm $\mathsf{MTVerify}$ does not allow internal nodes with left node value equal to 0, hence this means the adversary was able to compute a^*, b^*, c^* where $a^* \neq 0$ such that $\mathsf{H}(a^*, b^*, c^*) = \mathsf{H}(0, x, y)$ and thus a collision for H is found.
3. The last value v^* of $\tilde{T^*}$ corresponds to an internal node in $\mathcal{T}$. Observe that in this case we must have $\tilde{T^*} = T^*$ as otherwise this would contradict the fact that $\tilde{T^*}$ is the longest prefix of T^*. Hence this last value v^* must be a leaf value and thus the adversary has found x^*, y^* such that $\mathsf{H}(0, x^*, y^*) = \mathsf{H}(a, b, c)$ where $a \neq 0$ which is a collision for H.