Multi-Asset Shielded Pool Specification

by Heliax AG

based on the original Zcash spec by:

Daira Hopwood[†]

Sean Bowe[†] — Taylor Hornby[†] — Nathan Wilcox[†]

April 26, 2021

Abstract. Changes to the Sapling protocol to support multiple asset types. Research and experimental.

Keywords: anonymity, applications, cryptographic protocols, electronic commerce and payment, financial privacy, proof of work, zero knowledge.

The purpose of this document is to describe the changes made to the **Sapling** circuits to allow for user-defined assets. Only the circuit-level changes are specified; protocol-level or contract-level specifications must be described as well.

The following discussions, proposals, and demos provide background and context for the development of this specification:

- `https://github.com/zcash/zips/pull/269`
- `https://github.com/zcash/zcash/issues/830`
- `https://github.com/zcash/zcash/issues/2277#issuecomment-321106819`
- `https://github.com/str4d/librustzcash/tree/funweek-uda-demo`

As well as the original **Sapling** specification. Where possible, sections copied from the original **Sapling** specification have changes highlighted in purple and additional comments highlighted in blue.

0.1 Overview and Approach

The **Sapling** circuits rely on homomorphic Pedersen commitments to represent the value of a shielded Note. The homomorphic Pedersen commitment requires two generators of the same subgroup: one to serve as the value base, and another as the randomness base. For security, no discrete log relationship should be known between these two generators. In **Sapling**, both generators are carefully constructed and fixed outside of the circuits as images of a *Pseudo Random Function*.

User-defined assets may be added by varying the generator used as the value base, using a custom asset generator for each distinct asset type. However, since the value base generator is no longer a fixed constant, each asset generator must be dynamically constructed with similar security properties to the construction of the original fixed generator of **Sapling**.

This approach has several significant advantages:

[†] Electric Coin Company

- Based on the preliminary approach proposed in the ZIP
- Allows most of the extensive Sapling feature set to be extended to multiple assets
- Leverages the significant effort invested in developing and auditing the high quality Sapling specification and codebase by making as few changes as possible
- General enough that the same multi-asset circuits can be reused in different contexts

0.2 Asset Types: Notation and Nomenclature

An *asset type* is an abstract property added to a **Sapling** Note, in addition to the value. Notes only have one asset type and all transactions are balanced independently across all asset types. However, different mathematical and computational representations of an asset type will be necessary. To ensure consistency and unambiguity, we will use the following **names and nomenclature** for different representations of an *asset type*:

- The *name* of an asset is a user-defined bytestring of arbitrary length that uniquely represents a given asset type. Examples of this may include a combination of:
  - a smart contract address
  - contract-specific data or fields
  - cryptographic salt
  - random beacon
- The *identifier* of an asset is a 32-byte string derived from the asset *name* in a deterministic way. The asset *identifier* differs from the asset *name* in three respects:
  1. The asset *identifier* is a compressed representation of the asset type. The *name* may be an arbitrary length whereas the *identifier* is always 32 bytes.
  2. Only a constant fraction (approximately 45%) of 32 byte strings will be valid asset identifiers
  3. The asset *identifier* is always the Blake2s preimage of the asset *generator* (defined next)
- The asset *generator* (also known as the *value base*) is a valid *ctEdwards curve* point on the *Jubjub curve*, whose compressed bit representation is the BLAKE2s image of the asset *identifier*

The exact contents of the *asset name* may be defined outside of the circuit specifications. The asset name could include the output of a random beacon or other unpredictable randomness to prevent the possibility of precomputation attacks against a particular asset type.

In all cases, the asset *identifier* should be derived from the asset *name* in such a way that invalid identifiers are never generated and all generated identifiers are the same length. The simplest way to derive such identifiers is by rejection sampling.

The asset *generator* will be derived via a *Pseudo Random Function* from the asset *identifier*. This computation must be efficient (it is computed in the Output circuit) and also be plausibly computationally infeasible to know a discrete log relationship between the asset generators of two distinct asset types.

Asset types may also be associated with a *human-readable asset name* and/or a *asset symbol*. The human-readable asset name and asset symbol may be used for user-facing presentations of the asset type, particularly if the *asset name* is not suitable for this purpose. Assignment and use of human-readable asset names and asset symbols are outside the scope of this document.

The 32 byte size of the asset identifier is somewhat arbitrary, but strongly motivated by the following observations:

- The asset identifier length should be constant size; this simplifies the circuit design, and also the encrypted note should use a constant size field to represent the asset type.
- The asset identifier length should not be longer than 64 bytes unless explicitly necessary; otherwise the cost of hashing in the Output circuit increases significantly with every additional 64 byte input block. Additionally, if encrypted notes represent the note's asset type with an asset identifier field, longer asset identifiers increase the encrypted note size and use more storage space on the block chain.

- The benefits of asset identifier length shorter than 32 bytes are not significant. There is insignificant benefit in the cost of the Output circuit, as the in-circuit hash uses at least one 32 byte input block. If a shorter asset type representation is implementation-desired (e.g. for encrypted notes), the implementation could use a shorter asset name, truncated asset identifier, lookup table of assets, etc., as appropriate for the application.
- The asset identifier should be long enough to include added entropy, if desired. For example, adding entropy from a randomness beacon (discussed later) to add to the unpredictability of the asset generator. A 32 byte asset identifier accommodates a large amount of potential added entropy without going over the 64 byte block input size.

0.3 Derivation of Asset Generator from Asset Identifer

The asset generator associated with each asset type must be derived in such a way that plausibly no discrete log relationship is known between every two distinct asset types (or between an asset generator and the common randomness base generator).

In this specification, the asset generator associated with a given asset identifier is derived using a *Pseudo Random Function*; specifically, instantiating $\mathsf{PRF}^{\mathsf{vcgMASP}}()$ with BLAKE2s similar to how other *Pseudo Random Functions* are instantiated in the original **Sapling** specification. Therefore, the asset generator associated with asset identifier t should be $\mathsf{repr}_{\mathbb{J}}(\mathsf{PRF}^{\mathsf{vcgMASP}}(\mathsf{t}))$, if it exists, and this derivation is verified in at least one circuit.

The *Pseudo Random Function* $\mathsf{PRF}^{\mathsf{vcgMASP}}()$ should take as input the 32 byte asset identifier, and produce as output a (potential) 32 byte *ctEdwards compressed encoding* on the *Jubjub curve*. The output must be verified to be a valid *ctEdwards curve* point, and when instantiated with BLAKE2s, use a distinct personalization from the other *Pseudo Random Functions* used in the original Sapling specification and this specification.

One may wonder if it is necessary to verify the derivation of the asset generator from the asset identifier in circuit. The answer is "yes": if the asset generator was witnessed to the circuit's private inputs without checking its validity as an asset generator, then someone may witness the negation of an asset generator and produce notes with negative value of the actual asset (and therefore, creating notes of arbitrarily positive value that homomorphically balance with the negative value note)

One may also wonder if a Pedersen hash may be used instead (particularly as it is much more efficient to compute in the circuit than a *Pseudo Random Function*). The answer is that it may not be used: a Pedersen hash is not a *Pseudo Random Function*, and while it may offer collision resistance, it is possible to find related preimages easily. For example, because the Pedersen hash generators are publicly known, given an existing asset identifier and asset generator, someone may derive new asset identifiers and new asset generators that have some known fixed relationship to the existing asset generator. This may allow unwanted conversion between valid asset types.

0.4 Rejection Sampling of Asset Identifiers Hashing to Curve Point

The previous section noted that the asset generator associated with asset identifier t should be $\mathsf{repr}_{\mathbb{J}}(\mathsf{PRF}^{\mathsf{vcgMASP}}(\mathsf{t}))$. However, $\mathsf{repr}_{\mathbb{J}}(\mathsf{PRF}^{\mathsf{vcgMASP}}(\mathsf{t}))$ may not exist, in which case t is an *invalid* asset identifier. To avoid excessive computation in the Output circuit, such invalid t identifiers should always be rejected by the Output circuit, and all external implementations should only use valid asset identifiers exclusively.

The asset identifier should be deterministically derived from the asset name. Since there is some probability of deriving an invalid asset identifier, one potential approach is to try potential asset identifiers, rejecting invalid ones, until a valid asset identifier that properly hashes to an asset generator. We can describe such a process as *rejection sampling*.

$\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}$, which is used to find generators for Pedersen commitments and hashes in **Sapling**, may also be used to derive the asset identifier. In this case $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}$(asset name) may be the asset generator associated with some asset name; the asset identifier would be the preimage of the resulting asset generator (computed as an intermediate step of $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}$). Alternatively, the asset name may be hashed repeatedly until a valid asset identifier is found. The size of a 32 byte asset identifier is also intended to facilitate derivation of an asset identifier as the image of a hash function.

Hashing a uniformly random asset identifier bytestring to a group element, a *ctEdwards curve* point on the *Jubjub curve*, can fail in one of three ways:

1. The identifier could hash to a small order point on the curve. Since the *Jubjub curve* is the direct sum of a small order subgroup with a large prime order subgroup, the BLAKE2s image of the identifier may be the y coordinate of a small order point on the curve, and so when multiplied by the cofactor gives the identity. The small order subgroup contains very few elements, so the probability of hashing to one of these points is extremely small (exponentially small).

   Identifiers whose BLAKE2s hash is a small order point are rejected.

2. The identifier could hash to 256 bits, of which the leading 255 bits encode an integer that is at least the order of the underlying field of the *Jubjub curve*, and therefore is not a valid field element unless taken modulo the order of the field (which we cannot do, if we desire a uniformly random curve point in the random oracle model).

   The probability of this event is approximately 9.431% and so it occurs reasonably often.

   Identifiers whose BLAKE2s hash is larger than the field modulus are rejected.

3. The identifier could hash to 256 bits, of which the leading 255 bits encode a field element such that no point on the curve has that field element as y coordinate. Then it is not possible to interpret the BLAKE2s hash image as a compressed representation of a curve point/group element at all.

   The probability of this event is approximately (but not precisely) 1/2

   Identifiers whose BLAKE2s hash is not the compressed representation of some *Jubjub curve* point are rejected.

The overall probability that a uniformly random identifier hashes successfully is approximately 0.5 * 0.9057 = 0.453 and so the expected number of identifiers tried is approximately 2.2.

Some theoretical attacks against the asset identifier generation process are noted:

1. Rejection sampling is not constant time, potentially allowing side channel attacks that leak the asset type.
2. An attacker may attempt to find asset names that generate long sequences of invalid asset identifiers before finding a valid one. Extremely long sequences are likely infeasible to precompute but shorter sequences are more feasible, causing the asset identifier generation process to use more computation than average for a certain asset.

0.5 Hash-to-curve RFC

There exists a draft RFC (`https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/`) for hashing data to curves which differs substantially from the methods described to derive the asset generator. Indeed, the RFC explicitly disallows the use of rejection sampling. There are several advantages of using the RFC hash-to-curve methods:

1. The RFC describes a well documented and well analyzed approach to hash-to-curve
2. The RFC hash-to-curve is designed to be implemented in constant time, mitigating certain information leakage
3. The RFC hash-to-curve may later be used in other projects and there is potential value in standardization

However, there are also factors in favor of the rejection sampling approach:

1. Rejection sampling is conceptually somewhat simpler and easier to reason about

2. Rejection sampling is typically less expensive in total circuit operations, reducing the total cost
3. Rejection sampling primarily uses well-audited primitives and existing code/gadgets; potentially less novel code and fewer bugs than implementing new gadgets
4. Rejection sapling is already used widely in **Sapling** to derive group hashes (for Pedersen hashes and commitments. In particular, the circuits permit using exactly the existing **Sapling** GroupHash$^{\mathbb{J}^{(r)*}}$ to derive the asset generator (among other possible choices) while the RFC is a draft.

Unfortunately, supporting both RFC and rejection sampling based asset generator derivation in the same circuits appears impractical at the moment. At minimum, the added complexity of supporting both would increase the potential for implementation bugs.

Implementation of RFC compatible hash-to-curve gadgets in the bellman library would allow the use of RFC hash-to-curve inside the circuits; however such an implementation still adds significant complexity and requires significant effort to achieve the same level of analysis and scrutiny that the simpler rejection sampling method would. The MASP demo only supports rejection sampling based asset generator derivation.

Not using the RFC hash-to-curve method does introduce some concrete risks. The asset generator derivation is now likely not constant time, introducing potential for side channel attacks and information leakage. It should be noted that constant time implementation and side channel attack prevention is not a goal of the MASP demo. Side channel and information leakage should be avoided by typical isolation of private data and computation as much as possible.

Side channel information leakage and timing attacks may be slightly mitigated by observing that asset identifier derivation is a public process (in order to allow shielding/unshielding of transparent balance) and does not use private data. In this way the situation is slightly different than if private data is hashed to curve. Implementations could store derived asset identifiers and hash those in constant time to asset generators. It should be noted that since asset names are not specified in the circuits or this document that variable length asset names (or other properties of the asset identifier derivation process) may leak additional information besides timing.

0.6 Security

The homomorphic Pedersen value commitments are constructed similarly to the original Sapling circuit and should be similarly *value hiding* (infeasible to recover the value from the commitment without knowledge of the trapdoor randomness) and *non-forgeable* (infeasible to open the value commitment to another value). This requires that no discrete log relationship is known between the *value base* (in this case, the *asset generator*) and the *randomness trapdoor generator*.

When there are multiple assets, the value commitment should also be *asset hiding* and *non-exchangeable*: it should be infeasible to recover the asset type without knowledge of the trapdoor, and it should be infeasible to open the value commitment to another asset. This requires that no discrete log relationship is known between every pair of asset generators. If asset generators are derived in a uniformly random way, then deriving a discrete log relationship between asset generators should be approximately as difficult as finding a discrete log relationship between a constant value base and fixed randomness base generator.

The security of these multiple asset value commitments relies on similar assumptions underlying the security of the homomorphic Pedersen commitments and Pedersen hashes of the original **Sapling** circuits.

The security of those commitments and hashes is based on the hardness of the discrete log problem over a given elliptic curve group. For expository purposes, here is an informal argument sketch: Let $G_1, \ldots, G_k$ be k uniformly random elliptic curve points. Assume there is an algorithm that finds a discrete log relationship between a single pair G_i, G_j faster than finding a discrete log relationship between two chosen points P, Q. Then by choosing $2k$ uniformly random elements a_i, b_i of the finite field of the same order as the curve, finding a discrete log relationship among a single pair of $R_i = [a_i]P + [b_i]Q$ should reveal a discrete log relationship between P, Q. A more rigorous proof may be found in the literature.

Recall the use of a *Pseudo Random Function* to derive the asset generator from the asset identifier. Continuing the example from earlier, it should be infeasible for someone to find two asset identifiers whose images are points P

and $[-1]P$; otherwise an unlimited amount of those assets can be created in notes that balance homomorphically to zero in a single transaction. In the concrete case of BLAKE2s and the *Jubjub curve*, the points P and $[-1]P$ differ by a single bit in the compressed point representation (the sign of the x coordinate) and share the other 255 bits (the y coordinate). Therefore, the non-forgeability of assets depends on the infeasibility of finding two BLAKE2s preimages that differ only in one (positioned) bit.

The (in)ability to witness the negation of an asset generator is one specific example of the security requirements from the *Pseudo Random Function*. This example is a particularly dramatic one, as well, since witnessing a valid asset identifier and the negation of its asset generator will potentially satisfy all constraints in the Output circuit except for a *single bit* equality check (the sign bit).

It is similarly important that for every pair of asset generators, no discrete log relationship should be feasibly known between them. Otherwise, the non-exchangeability or non-forgeability of those assets become broken; either a positive amount of one asset may be converted into a positive amount of the other asset, or positive amounts of both assets may be created from an overall zero incoming note value (depending on the exact discrete log relationship known). Therefore, a critically important security property of the BLAKE2s hash is *Discrete Logarithm Independence* of its outputs interpreted as curve points. As the original **Sapling** specification describes, Discrete Logarithm Independence holds almost surely for a random oracle, and is stronger than (and implies) collision resistance.

For the purposes of analyzing the security of the circuits, the desired security property is that it is infeasible to (adversarially) find two asset identifiers with a known discrete log relationship between their corresponding asset generators. The security of the circuits will be based on that hardness assumption. The security assumption may be weakened by not allowing the asset identifier to be selected entirely freely (e.g. including a randomness beacon, as in the **Sapling** $\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}$.

0.7 Multiple Asset Heterogenous Transactions

As in the single asset **Sapling** model, a transaction may consist of some number of incoming notes and some number of outgoing notes (typically at least two of each) such that the sum of values of outgoing (created) notes minus the sum of values of incoming (spent) notes is equal to the change in the total transparent value of the pool. In the case of multiple assets, this sum should be balanced independently across all possible asset types. While every note has only one asset type, it is possible that transactions may contain notes of different asset types (*heterogenous transactions*). The use of homomorphic Pedersen commitments allows the sum to be balanced verifiably outside of the circuits even when the asset types of the notes are unknown.

Since every asset generator is prime order, the theoretical possibility exists of overflow of the value field when notes are balanced in a transaction. It is not possible to externally observe overflow, as it may occur even if the value commitments balance. For example, in a transaction, the sum of incoming note values could equal 0, and the sum of outgoing note values could be some integer multiple of the prime order. Opening this transparent value change commitment to 0 modulo the prime order would violating the desired non-forgeable property. This attack is likely impractical due to the limitation of each note value to an unsigned 64-bit integer; however there is no restriction inside the circuit to prevent overflow. The number of notes used in a transaction should be limited to a safe value to further mitigate this issue. The original Zcash **Sapling** protocol addresses this issue by limiting the maximum transaction size.

Additionally, it should be noted that the binding signature in the original **Sapling** protocol only supports a single asset generator, and so only a single asset type can be shielded or unshielded with a nonzero transparent balance change in a given transaction. While this behavior is logical in the single asset shielded pool, in the multiple asset shielded pool this is an unnecessary restriction. Binding signatures for nonzero transparent balance change for multiple assets in a single transaction is a potential feature that can be implemented entirely outside of the circuits.

0.8 Random beacon

Derivation of an asset identifier from a name may include the input of a random beacon, to lower the probability that some party did precomputation on the resulting asset generator prior to the asset name becoming public (or some other point in time). Various preexisting random beacons can be used, or new randomness beacons can be used for this purpose, or even dynamically used every time a new asset type is created.

Since there is no provision inside of the circuits for an entropy source, all entropy that will be included in the final asset generator must be included in the asset identifier. A randomness beacon is not used to derive the asset generator from the asset identifier inside the circuit because:

- Adding another 32 byte input block to the BLAKE2s hash inside the circuit would significantly increase the size of the Output circuit with a corresponding performance penalty.
- A 32 byte asset identifier can contain a reasonable amount of entropy from a randomness beacon used to derive the asset identifier; therefore, the entropy from a beacon can be inserted in the asset identifier derivation, outside of the circuit, while still influencing the final asset generator.
- The randomness beacon used to (ultimately) derive a particular asset generator does not need to be known at the time of circuit creation. New asset types could use a fresh randomness beacon, or randomness generated in some other way, depending on the specific case.

The randomness beacons used in the original system are based on hashes of specified bitcoin blocks; subsequently the entropy available from this source has slightly decreased and other randomness beacon sources (e.g. `https://blog.cloudflare.com/league-of-entropy/`) have become available.

0.9 Personalizations

The original **Sapling** circuits and accompanying out-of-circuit implementations use unique personalizations for each instantiation of a pseudorandom function or collision resistant hash function. Each personalization is an 8 byte string prefixed by the 5 bytes "Zcash". The personalizations in the MASP beta demo version of librustzcash are modified to be the exact same strings prefixed by the 5 bytes "MASP_" to add domain separation from the Zcash protocol, with the exceptions of the value base (now "MASP__v_"), and of the randomness base (now "MASP__r_")

0.10 Risks

The following (non-exhaustive) risks are noted and should be considered in all uses of the MASP demo:

- The use of non-constant time operations such as rejection sampling could allow side channel attacks and information leakage
- The Discrete Log Independence assumption necessary for non-forgeability and non-exchangeability properties of a given asset may not hold, either because of implementation errors or because the assumptions are false for the *Pseudo Random Function* used.
- Other newly introduced implementation errors from the incorrect use of existing circuit constructs or gadgets
- Unintentional and unexpected behavior of the **Sapling** protocol when used in a multiple asset context
- Unintentional and unexpected behavior of the **Sapling** protocol when used with assets with unlimited token issuance
- Potential unknown design or implementation flaws preexisting in the **Sapling** protocol
- Failure of the trusted setup process if Groth16 is used as the proving scheme for the Spend and Output circuits
- High levels of effort required to patch even minor bugs in the Spend and Output circuits, because of the repeated trusted setup required
- Catastrophic failures of even minor bugs, since bugs may be actively exploited for an indefinite period of time without publicly discovery, because of the zero knowledge properties of the proving system.
- Bugs or vulnerabilities may not be publicly discovered until the entire shielded pool has been drained of assets

- Privacy may be compromised by side channel attacks, information leakage, metadata and traffic analysis of transactions, payment of transparent fees to use the shielded pool, and other potential sources of information

0.11 Notes

A *note* (denoted $\mathbf{n}$) can be a **Sprout** *note* or a **Sapling** *note*. In either case it represents that a value v is spendable by the recipient who holds the *spending key* corresponding to a given *shielded payment address*.

Let MAX_MONEY, $\ell_{\mathsf{PRFSprout}}$, $\ell_{\mathsf{PRFnfSapling}}$, and ℓ_{d} be as defined in the original **Sapling** specification.

Let $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as defined in the original **Sapling** specification.

Let $\mathsf{KA}^{\mathsf{Sapling}}$ be as defined in the original **Sapling** specification.

Let $\ell_{\mathsf{t}} = 32$ bytes be the length of the asset identifier.

A **Sapling** *note* is a tuple $(\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \mathsf{rcm}, \mathsf{t})$, where:

- $\mathsf{d} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathsf{d}}]}$ is the *diversifier* of the recipient's *shielded payment address*;
- $\mathsf{pk_d} \mathrel{⦂} \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder}$ is the *diversified transmission key* of the recipient's *shielded payment address*;
- $\mathsf{v} \mathrel{⦂} \{0 \,..\, \mathsf{MAX\_MONEY}\}$ is an integer representing the value of the *note* in *zatoshi*;
- $\mathsf{rcm} \mathrel{⦂} \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor}$ is a random *commitment trapdoor* as defined in the original **Sapling** specification.
- $\mathsf{t} \mathrel{⦂} \mathbb{B}^{[\ell_{\mathsf{t}}]}$ is a bytestring representing the asset identifier of the note

Let $\mathsf{Note}^{\mathsf{Sapling}}$ be the type of a **Sapling** *note*, i.e.

$$\mathsf{Note}^{\mathsf{Sapling}} := \mathbb{B}^{[\ell_{\mathsf{d}}]} \times \mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder} \times \{0 \,..\, \mathsf{MAX\_MONEY}\} \times \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{Trapdoor} \times \mathbb{B}^{[\ell_{\mathsf{t}}]}.$$

Creation of new *notes* is as described in the original **Sapling** specification. When *notes* are sent, only a commitment to the above values is disclosed publically, and added to a data structure called the *note commitment tree*. This allows the value and recipient to be kept private, while the commitment is used by the *zero-knowledge proof* when the *note* is spent, to check that it exists on the *block chain*.

Let DiversifyHash be as defined in the original **Sapling** specification.

A **Sapling** *note commitment* on a *note*$\mathbf{n} = (\mathsf{d}, \mathsf{pk_d}, \mathsf{v}, \mathsf{rcm}, \mathsf{t})$ is computed as

$$\mathsf{g_d} := \mathsf{DiversifyHash}(\mathsf{d})$$

$$\mathsf{NoteCommitment}^{\mathsf{Sapling}}(\mathbf{n}) := \begin{cases} \bot, & \text{if } \mathsf{g_d} = \bot \\ \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v}, \mathsf{repr}_{\mathbb{J}}(\mathsf{PRF}^{\mathsf{vcgMASP}}(\mathsf{t}))), & \text{otherwise.} \end{cases}$$

where $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ is instantiated as in the original **Sapling** specification.

Notice that the above definition of a **Sapling** *note* does not have a ρ field. There is in fact a ρ value associated with each **Sapling** *note*, but this can only be computed once its position in the *note commitment tree* is known. We refer to the combination of a *note* and its *note position* pos, as a *positioned note*.

For a *positioned note*, we can compute the value ρ as described in the original **Sapling** specification.

A *nullifier* (denoted nf) is derived from the ρ value of a *note* and the recipient's *spending key* $\mathsf{a_{sk}}$ or *nullifier deriving key* nk. This computation uses a *Pseudo Random Function*, as described in the original **Sapling** specification.

A *note* is spent by proving knowledge of $(\rho, \mathsf{a_{sk}})$ or $(\rho, \mathsf{ak}, \mathsf{nsk})$ in zero knowledge while publically disclosing its *nullifier* nf, allowing nf to be used to prevent double-spending. In the case of **Sapling**, a *spend authorization signature* is also required, in order to demonstrate knowledge of ask.

0.11.1 Sending Notes (Sapling)

This section describes potential outside of circuit implementation details.

In order to send **Sapling** *shielded* value, the sender constructs a *transaction* containing one or more *Output descriptions*.

Let $\mathsf{ValueCommit}$, $\mathsf{NoteCommit}^{\mathsf{Sapling}}$, $\mathsf{KA}^{\mathsf{Sapling}}$, $\mathsf{DiversifyHash}$, $\mathsf{repr}_{\mathbb{J}}$, $r_{\mathbb{J}}$, and $h_{\mathbb{J}}$ be as defined in the original **Sapling** specification.

Let ovk be an *outgoing viewing key* that is intended to be able to decrypt this payment. This may be one of:

- the *outgoing viewing key* for the address (or one of the addresses) from which the payment was sent;
- the *outgoing viewing key* for all payments associated with an *"account"*, to be defined in [**ZIP-32**];
- $\bot$, if the sender should not be able to decrypt the payment once it has deleted its own copy.

Note: Choosing $\mathsf{ovk} = \bot$ is useful if the sender prefers to obtain forward secrecy of the payment information with respect to compromise of its own secrets.

For each *Output description*, the sender selects a value $\mathsf{v}^{\mathsf{new}} ⦂ \{0\,..\,\mathsf{MAX\_MONEY}\}$ and a destination **Sapling** *shielded payment address* $(\mathsf{d}, \mathsf{pk_d})$, and then performs the following steps:

- Check that $\mathsf{pk_d}$ is of type $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder}$, i.e. it is a valid *ctEdwards curve* point on the *Jubjub curve* (as defined in the original **Sapling** specification) not equal to $\mathcal{O}_{\mathbb{J}}$, and $[r_{\mathbb{J}}]\, \mathsf{pk_d} = \mathcal{O}_{\mathbb{J}}$.
- Calculate $\mathsf{g_d} = \mathsf{DiversifyHash}(\mathsf{d})$ and check that $\mathsf{g_d} \neq \bot$.
- Choose independent uniformly random commitment trapdoors:

$$\mathsf{rcv}^{\mathsf{new}} \overset{\mathrm{R}}{\leftarrow} \mathsf{ValueCommit.GenTrapdoor}()$$
$$\mathsf{rcm}^{\mathsf{new}} \overset{\mathrm{R}}{\leftarrow} \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{GenTrapdoor}()$$

- Check that $[h_{\mathbb{J}}]\, \mathsf{repr}_{\mathbb{J}}(\mathsf{PRF}^{\mathsf{vcgMASP}}(\mathsf{t}))$ is of type $\mathsf{KA}^{\mathsf{Sapling}}.\mathsf{PublicPrimeOrder}$, i.e. it is a valid *ctEdwards curve* point on the *Jubjub curve* (as defined in the original **Sapling** specification) not equal to $\mathcal{O}_{\mathbb{J}}$. If it is equal to $\mathcal{O}_{\mathbb{J}}$, t is an invalid *asset identifier*.

$$\mathsf{vb} := \mathsf{repr}_{\mathbb{J}}(\mathsf{PRF}^{\mathsf{vcgMASP}}(\mathsf{t}))$$

- Calculate

$$\mathsf{cv}^{\mathsf{new}} := [\mathsf{v}^{\mathsf{new}} h_{\mathbb{J}}]\, \mathsf{vb} + [\mathsf{rcv}^{\mathsf{new}}]\, \mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}(\texttt{"MASP\_\_r\_"}, \texttt{"r"})$$
$$\mathsf{cm}^{\mathsf{new}} := \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}^{\mathsf{new}}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v}^{\mathsf{new}}, \mathsf{vb})$$

- Let $\mathbf{np} = (\mathsf{d}, \mathsf{v}^{\mathsf{new}}, \underline{\mathsf{rcm}}, \mathsf{memo}, \mathsf{t})$, where $\underline{\mathsf{rcm}} = \mathsf{LEBS2OSP}_{256}(\mathsf{I2LEBSP}_{256}(\mathsf{rcm}^{\mathsf{new}}))$.
- Encrypt $\mathbf{np}$ to the recipient *diversified transmission key* $\mathsf{pk_d}$ with *diversified transmission base* $\mathsf{g_d}$, and to the *outgoing viewing key* ovk, giving the *transmitted note ciphertext* $(\mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}})$ as described in the original **Sapling** specification. This procedure also uses $\mathsf{cv}^{\mathsf{new}}$ and $\mathsf{cm}^{\mathsf{new}}$ to derive the *outgoing cipher key*.
- Generate a proof π_{ZKOutput} for the *Output statement* in § 0.12.3 *'Output Statement (**Sapling**)'* on p. 12.
- Return $(\mathsf{cv}^{\mathsf{new}}, \mathsf{cm}^{\mathsf{new}}, \mathsf{epk}, \mathsf{C}^{\mathsf{enc}}, \mathsf{C}^{\mathsf{out}}, \pi_{\mathsf{ZKOutput}})$.

In order to minimize information leakage, the sender **SHOULD** randomize the order of *Output descriptions* in a *transaction*. Other considerations relating to information leakage from the structure of *transactions* are beyond the scope of this specification. The encoded *transaction* is submitted to the network.

0.12 Dummy Notes

0.12.1 Dummy Notes (Sapling)

In **Sapling** there is no need to use *dummy notes* simply in order to fill otherwise unused inputs as in the case of a *JoinSplit description*; nevertheless it may be useful for privacy to obscure the number of real *shielded inputs* from **Sapling** *notes*.

Let ℓ_{sk}, $r_{\mathbb{J}}$, $\mathsf{repr}_{\mathbb{J}}$, $\mathcal{H}$, $\mathsf{PRF}^{\mathsf{nfSapling}}$, $\mathsf{NoteCommit}^{\mathsf{Sapling}}$ be as defined in the original **Sapling** specification.

A *dummy* **Sapling** input *note* is constructed as follows:

- Choose uniformly random $\mathsf{sk} \xleftarrow{R} \mathbb{B}^{[\ell_{\mathsf{sk}}]}$.
- Generate a new *diversified payment address* $(\mathsf{d}, \mathsf{pk_d})$ for sk as described in the original **Sapling** specification.
- Set $\mathsf{v}^{\mathsf{old}} = 0$, and set $\mathsf{pos} = 0$.
- Choose uniformly random $\mathsf{rcm} \xleftarrow{R} \mathsf{NoteCommit}^{\mathsf{Sapling}}.\mathsf{GenTrapdoor}()$. and $\mathsf{nsk} \xleftarrow{R} \mathbb{F}_{r_{\mathbb{J}}}$.
- Compute $\mathsf{nk} = [\mathsf{nsk}]\,\mathcal{H}$ and $\mathsf{nk}\star = \mathsf{repr}_{\mathbb{J}}(\mathsf{nk})$.
- Compute $\rho = \mathsf{cm}^{\mathsf{old}} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v}^{\mathsf{old}}, \mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}(\text{“}\mathtt{MASP\_\_r\_}\text{”}, \text{“}\mathtt{r}\text{”}))$.
- Compute $\mathsf{nf}^{\mathsf{old}} = \mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}\star}(\mathsf{repr}_{\mathbb{J}}(\rho))$.
- Construct a *dummy Merkle path* path for use in the *auxiliary input* to the *Spend statement* (this will not be checked, because $\mathsf{v}^{\mathsf{old}} = 0$).

As in **Sprout**, a *dummy* **Sapling** output *note* is constructed as normal but with zero value, and sent to a random *shielded payment address*.

0.12.2 Spend Statement (Sapling)

The new Spend circuit has 100637 constraints. The original Sapling Output circuit has 98777 constraints.
Let $\ell_{\mathsf{MerkleSapling}}$, $\ell_{\mathsf{PRFnfSapling}}$, ℓ_{scalar}, $\mathsf{ValueCommit}$, $\mathsf{NoteCommit}^{\mathsf{Sapling}}$, $\mathsf{SpendAuthSig}$, $\mathbb{J}$, $\mathbb{J}^{(r)}$, $\mathsf{repr}_{\mathbb{J}}$, $q_{\mathbb{J}}$, $r_{\mathbb{J}}$, $h_{\mathbb{J}}$, $\mathsf{Extract}_{\mathbb{J}^{(r)}} : \mathbb{J}^{(r)} \to \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]}$, $\mathcal{H}$ be as defined in the original **Sapling** specification.

A valid instance of π_{ZKSpend} assures that given a *primary input*:

$$\begin{aligned}
&\big(\mathsf{rt} ⦂ \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]},\\
&\ \mathsf{cv}^{\mathsf{old}} ⦂ \mathsf{ValueCommit.Output},\\
&\ \mathsf{nf}^{\mathsf{old}} ⦂ \mathbb{B}^{[\ell_{\mathsf{PRFnfSapling}}]},\\
&\ \mathsf{rk} ⦂ \mathsf{SpendAuthSig.Public}\big),
\end{aligned}$$

the prover knows an *auxiliary input*:

$$\begin{aligned}
&\big(\mathsf{path} ⦂ \mathbb{B}^{[\ell_{\mathsf{Merkle}}][\mathsf{MerkleDepth}^{\mathsf{Sapling}}]},\\
&\ \mathsf{pos} ⦂ \{0\,..\,2^{\mathsf{MerkleDepth}^{\mathsf{Sapling}}}-1\},\\
&\ \mathsf{g_d} ⦂ \mathbb{J},\\
&\ \mathsf{pk_d} ⦂ \mathbb{J},\\
&\ \mathsf{v}^{\mathsf{old}} ⦂ \{0\,..\,2^{\ell_{\mathsf{value}}}-1\},\\
&\ \mathsf{rcv}^{\mathsf{old}} ⦂ \{0\,..\,2^{\ell_{\mathsf{scalar}}}-1\},\\
&\ \mathsf{cm}^{\mathsf{old}} ⦂ \mathbb{J},\\
&\ \mathsf{rcm}^{\mathsf{old}} ⦂ \{0\,..\,2^{\ell_{\mathsf{scalar}}}-1\},\\
&\ \alpha ⦂ \{0\,..\,2^{\ell_{\mathsf{scalar}}}-1\},\\
&\ \mathsf{ak} ⦂ \mathsf{SpendAuthSig.Public},\\
&\ \mathsf{nsk} ⦂ \{0\,..\,2^{\ell_{\mathsf{scalar}}}-1\},\\
&\ \mathsf{vb} ⦂ \mathbb{J}\big)
\end{aligned}$$

such that the following conditions hold:

Note commitment integrity $\mathsf{cm}^{\mathsf{old}} = \mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}^{\mathsf{old}}}(\mathsf{repr}_{\mathbb{J}}(\mathsf{g_d}), \mathsf{repr}_{\mathbb{J}}(\mathsf{pk_d}), \mathsf{v}^{\mathsf{old}}, \mathsf{vb})$.

Merkle path validity Either $\mathsf{v}^{\mathsf{old}} = 0$; or $(\mathsf{path}, \mathsf{pos})$ is a valid *Merkle path* of depth $\mathsf{MerkleDepth}^{\mathsf{Sapling}}$, as defined in the original **Sapling** specification, from $\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}(\mathsf{cm}^{\mathsf{old}})$ to the *anchor* rt.

Value commitment integrity $\mathsf{cv}^{\mathsf{old}} = [\mathsf{v}^{\mathsf{new}} h_{\mathbb{J}}]\,\mathsf{vb} + [\mathsf{rcv}^{\mathsf{new}}]\,\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}(\text{“}\mathtt{MASP\_\_r\_}\text{”}, \text{“}\mathtt{r}\text{”})$

Small order checks $\mathsf{g_d}$ and ak and vb are not of small order, i.e. $[h_{\mathbb{J}}]\,\mathsf{g_d} \neq \mathcal{O}_{\mathbb{J}}$ and $[h_{\mathbb{J}}]\,\mathsf{ak} \neq \mathcal{O}_{\mathbb{J}}$ and $[h_{\mathbb{J}}]\,\mathsf{vb} \neq \mathcal{O}_{\mathbb{J}}$.

Nullifier integrity $\mathsf{nf}^{\mathsf{old}} = \mathsf{PRF}^{\mathsf{nfSapling}}_{\mathsf{nk}\star}(\rho\star)$ where

$$\begin{aligned}
\mathsf{nk}\star &= \mathsf{repr}_{\mathbb{J}}([\mathsf{nsk}]\,\mathcal{H})\\
\rho\star &= \mathsf{repr}_{\mathbb{J}}\big(\mathsf{MixingPedersenHash}(\mathsf{cm}^{\mathsf{old}}, \mathsf{pos})\big).
\end{aligned}$$

Spend authority $\mathsf{rk} = \mathsf{SpendAuthSig.RandomizePublic}(\alpha, \mathsf{ak})$.

Diversified address integrity $\mathsf{pk_d} = [\mathsf{ivk}]\,\mathsf{g_d}$ where

$$\begin{aligned}
\mathsf{ivk} &= \mathsf{CRH}^{\mathsf{ivk}}(\mathsf{ak}\star, \mathsf{nk}\star)\\
\mathsf{ak}\star &= \mathsf{repr}_{\mathbb{J}}(\mathsf{ak}).
\end{aligned}$$

The form and encoding of *Spend statement* proofs may be Groth16 as in the original **Sapling** specification.

Notes:

- Public and *auxiliary inputs* **MUST** be constrained to have the types specified. In particular, see the original **Sapling** specifcation, for required validity checks on compressed representations of *Jubjub curve* points.

  The ValueCommit.Output and SpendAuthSig.Public types also represent points, i.e. $\mathbb{J}$.
- In the Merkle path validity check, each *layer* does ***not*** check that its input bit sequence is a canonical encoding (in $\{0\,..\,r_{\mathbb{S}}-1\}$) of the integer from the previous *layer*.

- It is ***not*** checked in the *Spend statement* that rk is not of small order. However, this ***is*** checked outside the *Spend statement*, as specified in the original **Sapling** specifcation.
- It is ***not*** checked that $\mathsf{rcv}^{\mathsf{old}} < r_{\mathbb{J}}$ or that $\mathsf{rcm}^{\mathsf{old}} < r_{\mathbb{J}}$.
- $\mathsf{SpendAuthSig.RandomizePublic}(\alpha, \mathsf{ak}) = \mathsf{ak} + [\alpha]\,\mathcal{G}$. ($\mathcal{G}$ is as defined in the original **Sapling** specifcation.)
- Note that the asset identifier is *not* witnessed in the $SpendStatement$. Since the validity of vb is witnessed in the $OutputStatement$ and included in the $Note$ commitment, the asset generator is validated when the $Note$ commitment is validated.

0.12.3 Output Statement (Sapling)

The new Output circuit has 31205 constraints. The original Sapling Output circuit has 7827 constraints. Most of the extra cost comes from computing one Blake2s hash in the circuit.

Let $\ell_{\mathsf{MerkleSapling}}$, $\ell_{\mathsf{PRFnfSapling}}$, ℓ_{scalar}, $\mathsf{ValueCommit}$, $\mathsf{NoteCommit}^{\mathsf{Sapling}}$, $\mathbb{J}$, $\mathsf{repr}_{\mathbb{J}}$, and $h_{\mathbb{J}}$ be as defined in the original **Sapling** specification.

A valid instance of π_{ZKOutput} assures that given a *primary input*:

$$\begin{aligned}
&\big(\mathsf{cv}^{\mathsf{new}} \,⦂\, \mathsf{ValueCommit.Output},\\
&\ \mathsf{cm}_u \,⦂\, \mathbb{B}^{[\ell_{\mathsf{MerkleSapling}}]},\\
&\ \mathsf{epk} \,⦂\, \mathbb{J}\big),
\end{aligned}$$

the prover knows an *auxiliary input*:

$$\begin{aligned}
&\big(\mathsf{g_d} \,⦂\, \mathbb{J},\\
&\ \mathsf{pk{\star}_d} \,⦂\, \mathbb{B}^{[\ell_{\mathbb{J}}]},\\
&\ \mathsf{v}^{\mathsf{new}} \,⦂\, \{0\,..\,2^{\ell_{\mathsf{value}}}-1\},\\
&\ \mathsf{rcv}^{\mathsf{new}} \,⦂\, \{0\,..\,2^{\ell_{\mathsf{scalar}}}-1\},\\
&\ \mathsf{rcm}^{\mathsf{new}} \,⦂\, \{0\,..\,2^{\ell_{\mathsf{scalar}}}-1\},\\
&\ \mathsf{esk} \,⦂\, \{0\,..\,2^{\ell_{\mathsf{scalar}}}-1\},\\
&\ \mathsf{vb} \,⦂\, \mathbb{J},\\
&\ \mathsf{t} \,⦂\, \mathbb{B}^{[\ell_{\mathsf{t}}]}\big)
\end{aligned}$$

such that the following conditions hold:

Note commitment integrity $\mathsf{cm}_u = \mathsf{Extract}_{\mathbb{J}^{(r)}}\big(\mathsf{NoteCommit}^{\mathsf{Sapling}}_{\mathsf{rcm}^{\mathsf{new}}}(\mathsf{g{\star}_d}, \mathsf{pk{\star}_d}, \mathsf{v}^{\mathsf{new}}, \mathsf{vb})\big)$, where $\mathsf{g{\star}_d} = \mathsf{repr}_{\mathbb{J}}(\mathsf{g_d})$.

Value commitment integrity $\mathsf{cv}^{\mathsf{new}} = [\mathsf{v}^{\mathsf{new}} h_{\mathbb{J}}]\,\mathsf{vb} + [\mathsf{rcv}^{\mathsf{new}}]\,\mathsf{GroupHash}^{\mathbb{J}^{(r)*}}_{\mathsf{URS}}(\text{“}\mathtt{MASP\_\_r\_}\text{”}, \text{“}\mathtt{r}\text{”})$

Value base integrity $\mathsf{vb} = \mathsf{repr}_{\mathbb{J}}(\mathsf{PRF}^{\mathsf{vcgMASP}}(\mathsf{t}))$

Small order check $\mathsf{g_d}$ and vb are not of small order, i.e. $[h_{\mathbb{J}}]\,\mathsf{g_d} \neq \mathcal{O}_{\mathbb{J}}$.

Ephemeral *public key* integrity $\mathsf{epk} = [\mathsf{esk}]\,\mathsf{g_d}$.

The form and encoding of *Output statement* proofs may be Groth16 as in the original **Sapling** specification.

Notes:

- Public and *auxiliary inputs* **MUST** be constrained to have the types specified. In particular, see the original **Sapling** specification, for required validity checks on compressed representations of *Jubjub curve* points.

  The $\mathsf{ValueCommit.Output}$ type also represents points, i.e. $\mathbb{J}$.
- The validity of $\mathsf{pk}\star_{\mathsf{d}}$ is ***not*** checked in this circuit.
- It is ***not*** checked that $\mathsf{rcv}^{\mathsf{old}} < r_{\mathbb{J}}$ or that $\mathsf{rcm}^{\mathsf{old}} < r_{\mathbb{J}}$.