

Cryptography

NOTE

This section only comes at the beginning because it contains foundational cryptographic definitions that other sections use. It's not recommended that you read this section first, because you'll probably give up before getting to the interesting sections.

Honk targets and win conditions

Introduction & context

Aztec's cryptography tech stack and its associated implementation is an open-ended project with potential for many enhancements, optimisations and scope-creep.

This document is designed to definitively answer the following questions:

1. What are the metrics we care about when measuring our cryptography components?
2. What are minimum satisfiable values for these metrics?
3. What are the aspirational values for these metrics?

Important Metrics

The following is a list of the relevant properties that affect the performance of the Aztec network:

- Size of a user transaction (in kb)
- Time to generate a user transaction proof
- Memory required to generate a user transaction proof
- Time to generate an Aztec Virtual Machine proof
- Memory required to generate an Aztec Virtual Machine proof
- Time to compute a 2-to-1 rollup proof
- Memory required to compute a 2-to-1 rollup proof

"MVP" = minimum standards that we can go to main-net with.

Note: gb = gigabytes (not gigabits, gigibits or gigabytes)

metric	how to measure	MVP (10tps)	ideal (100tps)
proof size	total size of a user tx incl. goblin plonk proofs	80kb	8kb
prover time	A baseline "medium complexity" transaction (in web browser). Full description further down	1 min	10 seconds
verifier time	how long does it take the verifier to check a proof (incl. grumpkin IPA MSMs)	20ms	1ms
client memory consumption	fold 2^{19} circuits into an accumulator an arbitrary number of times	4gb	1gb
size of the kernel circuit	number of gates	2^{17}	2^{15}
Aztec Virtual Machine prover time	10,000 step VM circuit	15 seconds	1.5 seconds
Aztec Virtual Machine	1 million VM step circuit	128gb	16gb

metric	how to measure	MVP (10tps)	ideal (100tps)
memory consumption			
2-to-1 rollup proving time	1 2-to-1 rollup proof	7.4 seconds	0.74 seconds
2-to-1 rollup memory consumption	1 2-to-1 rollup proof	128gb	16gb

To come up with the above estimates, we are targeting 10 transactions per second for the MVP and 100 tps for the "ideal" case. We are assuming both block producers and rollup Provers have access to 128-core machines with 128gb of RAM. Additionally, we assume that the various process required to produce a block consume the following:

process	percent of block production time allocated to process
transaction validation	10%
block building (tx simulation)	20%
public VM proof construction time	20%
rollup prover time	40%

process	percent of block production time allocated to process
UltraPlonk proof compression time	10%

These are very rough estimates that could use further evaluation and validation!

Proof size

The MVP wishes to target a tx throughput of 10 txs per second.

Each Aztec node (not sequencer/prover, just a regular node that is sending transactions) needs to download $10 * \text{proof_size}$ bytes of data to keep track of the mempool. However, this is the *best case* scenario.

More practically, the data throughput of a p2p network will be less than the bandwidth of participants due to network coordination costs. As a rough heuristic, we assume that network bandwidth will be 10% of p2p user bandwidth. NOTE: can we find some high-quality information about p2p network throughput relative to the data consumed by p2p node operators?

As a result, the MVP data throughput could scale up to $100 * \text{proof_size}$ bytes of data per second.

For an MVP we wish to target a maximum bandwidth of 8MB per second (i.e. a good broadband connection). This gives us a network bandwidth of 0.8MB/s.

This sets the proof size limit to 819.2 kb per second per 100 transactions => 82 kilobytes of data per transaction.

As a rough estimate, we can assume the non-proof tx data will be irrelevant

compared to 82kb, so we target a proof size of 80 kilobytes for the MVV.

To support 100 transactions per second we would require a proof size of 8 kilobytes.

Prover time

The critical UX factor. To measure prover time for a transaction, we must first define a baseline transaction we wish to measure and the execution environment of the Prover.

As we build+refine our MVP, we want to avoid optimising the best-case scenario (i.e. the most basic tx type, a token transfer). Instead we want to ensure that transactions of a "moderate" complexity are possible with consumer hardware.

As a north star, we consider a private swap, and transpose it into an Aztec contract.

To perform a private swap, the following must occur:

1. Validate the user's account contract (1 kernel call)
2. Call a swap contract (1 kernel call)
3. The swap contract will initiate `transfer` calls on two token contracts (2 kernel calls)
4. A fee must be paid via our fee abstraction spec (1 kernel call)
5. A final "cleanup" proof is generated that evaluates state reads and processes the queues that have been constructed by previous kernel circuits (1 kernel call + 1 function call; the cleanup proof)

In total we have 6 kernel calls and 6 function calls.

We can further abstract the above by making the following assumption:

1. The kernel circuit is 2^{17} constraints
2. The average number of constraints per function call is 2^{17} constraints, but the first

function called has 2^{19} constraints

Defining the first function to cost 2^{19} constraints is a conservative assumption due to the fact that the kernel circuit can support functions that have a max of 2^{19} constraints. We want to ensure that our benchmarks (and possible optimisations) capture the "heavy function" case and we don't just optimise for lightweight functions.

Summary of what we are measuring to capture Prover time

1. A mock kernel circuit has a size of 2^{17} constraints and folds *two* Honk instances into an accumulator (the prev. kernel and the function being called)
2. The Prover must prove 5 mock function circuit proofs of size 2^{17} and one mock function proof of size 2^{19}
3. The Prover must iteratively prove 6 mock kernel circuit proofs

Execution environment

For the MVP we can assume the user has reasonable hardware. For the purpose we use a 2-year-old macbook with 16gb RAM. The proof must be generated in a web browser

Performance targets

For an MVP, we target a 1 minute proof generation time. This is a substantial amount of time to ask a user to wait and we are measuring on good hardware.

In an ideal world, a 10 second proof generation time would be much better for UX.

Verifier time

This matters because verifying a transaction is effectively free work being performed by sequencers and network nodes that propagate txns to the mempool. If verification time becomes too large it opens up potential DDOS attacks.

If we reserve 10% of the block production time for verifying user proofs, at 10 transaction per seconds this gives us 0.01s per transaction. i.e. 10ms per proof.

If the block producer has access to more than one physical machine that they can use to parallelise verification, we can extend the maximum tolerable verification time. For an MVP that requires 20ms to verify each proof, each block producer would require at least 2 physical machines to successfully build blocks.

100tps with one physical machine would require a verification time of 1ms per proof.

Memory consumption

This is *critical*. Users can tolerate slow proofs, but if Honk consumes too much memory, a user cannot make a proof at all.

safari on iPhone will purge tabs that consume more than 1gb of RAM. The WASM memory cap is 4gb which defines the upper limit for an MVP.

Kernel circuit size

Not a critical metric, but the prover time + prover memory metrics are predicated on a kernel circuit costing about 2^{17} constraints!

AVM Prover time

Our goal is to hit main-net with a network that can support 10 transactions per second. We need to estimate how many VM computation steps will be needed per transaction to determine the required speed of the VM Prover. The following uses very conservative estimations due to the difficulty of estimating this.

An Ethereum block consists of approximately 1,000 transactions, with a block gas limit of roughly 10 million gas. Basic computational steps in the Ethereum Virtual

Machine consume 3 gas. If the entire block gas limit is consumed with basic computation steps (not true but let's assume for a moment), this implies that 1,000 transactions consume 3.33 million computation steps. i.e. 10 transactions per second would require roughly 33,000 steps per second and 3,330 steps per transaction.

As a conservative estimate, let us assume that every tx in a block will consume 10,000 AVM steps.

Our AVM model is currently to evaluate a transaction's public function calls within a single AVM circuit. This means that a block of n transactions will require n public kernel proofs and n AVM proofs to be generated (assuming all txns have a public component).

If public VM proof construction consumes 20% of block time, we must generate 10 AVM proofs and 10 public kernel proofs in 2 seconds.

When measuring 2-to-1 rollup prover time, we assume we have access to a Prover network with 500 physical devices available for computation.

i.e. 10 proofs in 2 seconds across 500 devices => 1 AVM + public kernel proof in 25 seconds per physical device.

If we assume that ~10 seconds is budgeted to the public kernel proof, this would give a 15 second prover time target for a 10,000 step AVM circuit.

100 tps requires 1.5 seconds per proof.

AVM Memory consumption

A large AWS instance can consume 128Gb of memory which puts an upper limit for AVM RAM consumption. Ideally consumer-grade hardware can be used to generate AVM proofs i.e. 16 Gb.

2-to-1 rollup proving time

For a rollup block containing 2^d transactions, we need to compute 2-to-1 rollup proofs across d layers (i.e. 2^{d-1} 2-to-1 proofs, followed by 2^{d-2} proofs, followed by... etc down to requiring 1 2-to-1 proof). To hit 10tps, we must produce 1 block in $\frac{2^d}{10}$ seconds.

Note: this excludes network coordination costs, latency costs, block construction costs, public VM proof construction costs (must be computed before the 2-to-1 rollup proofs), cost to compute the final UltraPlonk proof.

To accomodate the above costs, we assume that we can budget 40% of block production time towards making proofs. Given these constraints, the following table describes maximum allowable proof construction times for a selection of block sizes.

block size	number of successive 2-to-1 rollup proofs	number of parallel Prover machines required for base layer proofs	time required to construct a rollup proof
1,024	10	512	4.1s
2,048	11	1,024	7.4s
4,096	12	2,048	13.6s
8,192	13	4,096	25.2s
16,384	14	8,192	46.8s

We must also define the maximum number of physical machines we can reasonably

expect to be constructing proofs across the Prover network. If we can assume we can expect 1,024 machines available, this caps the MVP proof construction time at 7.4 seconds.

Supporting a proof construction time of 4.1s would enable us to reduce minimum hardware requirements for the Prover network to 512 physical machines.

2-to-1 rollup memory consumption

Same rationale as the public VM proof construction time.

Proving System Components

Interactive Proving Systems

Ultra Plonk

UltraPlonk is a variant of the [PLONK](#) protocol - a zkSNARK with a universal trusted setup.

UltraPlonk utilizes the "Ultra" circuit arithmetisation. This is a configuration with four wires per-gate, and the following set of gate types:

- arithmetic gate
- elliptic curve point addition/doubling gate
- range-check gate
- plookup table gate
- memory-checking gates
- non-native field arithmetic gates

Honk

Honk is a variant of the PLOK protocol. Honk performs polynomial testing via checking a polynomial relation is zero modulo the vanishing polynomial of a multiplicative subgroup. Honk performs the polynomial testing via checking, using a sumcheck protocol, that a relation over multilinear polynomials vanishes when summed over a boolean hypercube.

The first protocol to combine Plonk and the sumcheck protocol was [HyperPlonk](#)

Honk uses a custom arithmetisation that extends the Ultra circuit arithmetisation (not yet finalized, but includes efficient Poseidon2 hashing)

Incrementally Verifiable Computation Subprotocols

An Incrementally Verifiable Computation (IVC) scheme describes a protocol that defines some concept of persistent state, and enables multiple successive proofs to evolve the state over time.

IVC schemes are used by Aztec in two capacities:

1. to compute a client-side proof of one transaction execution.
2. to compute a proof of a "rollup" circuit, that updates rollup state based on a block of user transactions

Both use IVC schemes. Client-side, each function call in a transaction is a "step" in the IVC scheme. Rollup-side, aggregating two transaction proofs is a "step" in the IVC scheme.

The client-side IVC scheme is substantially more complex than the rollup-side scheme due to performance requirements.

Rollup-side, each "step" in the IVC scheme is a Honk proof, which are recursively verified. As a result, no protocols other than Honk are required to execute rollup-side IVC.

We perform one layer of "[proof-system compression](#)" in the rollup. The final proof of block-correctness is constructed as a Honk proof. An UltraPlonk circuit is used to verify the correctness of the Honk proof, so that the proof that is verified on-chain is an UltraPlonk proof. Verification gas costs are lower for UltraPlonk vs Honk due to the

following factors:

1. Fewer precomputed selector polynomials, reducing Verifier G1 scalar multiplications
2. UltraPlonk does not use multilinear polynomials, which removes 1 pairing from the Verifier, as well as $O(\log n)$ G1 scalar multiplications.

The following sections list the protocol components required to implement client-side IVC. We make heavy use of folding schemes to build an IVC scheme. A folding scheme enables instances of a relation to be folded into a single instance of the original relation, but in a "relaxed" form. Depending on the scheme, restrictions may be placed on the instances that can be folded.

The main two families of folding schemes are derived from the [Nova](#) protocol and the [Protostar](#) protocol respectively.

Protogalaxy

The [Protogalaxy](#) protocol efficiently supports the ability to fold multiple Honk instances (describing different circuits) into the same accumulator. To contrast, the Nova/Supernova/Hypernova family of folding schemes assume that a single circuit is being repeatedly folded (each Aztec function circuit is a distinct circuit, which breaks this assumption).

It is a variant of [Protostar](#). Unlike Protostar, Protogalaxy enables multiple instances to be efficiently folded into the same accumulator instance.

The Protogalaxy protocol is split into two subprotocols, each modelled as interactive protocols between a Prover and a Verifier.

Protogalaxy Fold

The "Fold" Prover/Verifier validates that k instances of a defined relation (in our case the Honk relation) have been correctly folded into an accumulator instance.

Protogalaxy Decider

The "Decider" Prover/Verifier validate whether an accumulator instance correctly satisfies the accumulator relation. The accumulator being satisfiable inductively shows that all instances that have been folded were satisfied as well. (additional protocol checks are required to reason about *which* instances have been folded into the accumulator. See the [IVC specification](#) for more information. (note to zac: put this in the protocol specs!)

Goblin Plonk

[Goblin Plonk](#) is a computation delegation scheme that improves Prover performance when evaluating complex algorithms.

In the context of an IVC scheme, Goblin Plonk enables a Prover to defer non-native group operations required by a Verifier algorithm, across multiple recursive proofs, to a single step evaluated at the conclusion of the IVC Prover algorithm.

Goblin Plonk is composed of three subcomponents:

Transcript Aggregation Subprotocol

This subprotocol aggregates deferred computations from two independent instances, into a single instance

Elliptic Curve Virtual Machine (ECCVM) Subprotocol

The ECCVM is a Honk circuit with a custom circuit arithmetisation, designed to optimally evaluate elliptic curve arithmetic computations that have been deferred. It is defined over the Grumpkin elliptic curve.

Translator Subprotocol

The Translator is a Honk circuit, defined over BN254, with a custom circuit arithmetisation, designed to validate that the input commitments of an ECCVM circuit align with the delegated computations described by a Goblin Plonk transcript commitment.

Plonk Data Bus

When passing data between successive IVC steps, the canonical method is to do so via public inputs. This adds significant costs to an IVC folding verifier (or recursive verifier when not using a folding scheme). Public inputs must be hashed prior to generating Fiat-Shamir challenges. When this is performed in-circuit, this adds a cost linear in the number of public inputs (with unpleasant constants ~30 constraints per field element).

The Data Bus protocol eliminates this cost by representing cross-step data via succinct commitments instead of raw field elements.

The Plonk Data Bus protocol enables efficient data transfer between two Honk instances within a larger IVC protocol.

Polynomial Commitment Schemes

The UltraPlonk, Honk, Goblin Plonk and Plonk Data Bus protocols utilize Polynomial

Interactive Oracle Proofs as a core component, thus requiring the use of polynomial commitment schemes (PCS).

UltraPlonk and Honk utilize multilinear PCS. The Plonk Data Bus and Goblin Plonk also utilize univariate PCS.

For multilinear polynomial commitment schemes, we use the [ZeroMorph](#) protocol, which itself uses a univariate PCS as a core component.

Depending on context we use the following two univariate schemes within our cryptography stack.

KZG Commitments

The [KZG](#) polynomial commitment scheme requires a universal setup and is instantiated over a pairing-friendly elliptic curve.

Computing an opening proof of a degree- n polynomial requires n scalar multiplications, with a constant proof size and a constant verifier time.

Inner Product Argument

The [IPA](#) PCS has worse asymptotics than KZG but can be instantiated over non-pairing friendly curves.

We utilize the Grumpkin elliptic curve as part of the Goblin Plonk protocol, where we utilize the curve cycle formed between BN254 and Grumpkin to translate expensive non-native BN254 group operations in a BN254 circuit, into native group operations in a Grumpkin circuit.

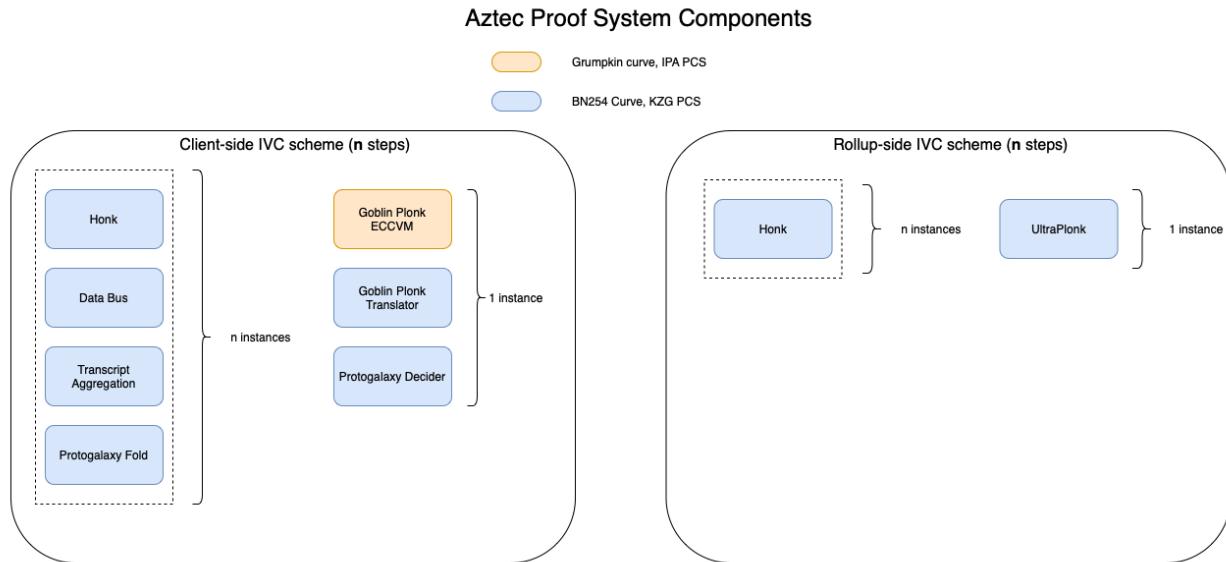
Computing an opening proof of a degree- n polynomial requires $2n$ scalar

multiplications, with a $O(\log n)$ proof size and an $O(n)$ verifier time.

To batch-verify multiple opening proofs, we use the technique articulated in the [Halo](#) protocol. To compute a proof of a single rollup block, only one linear-time PCS opening proof is verified despite multiple IPA proofs being generated as part of constructing the rollup proof.

Combined IVC + Proving System Protocol

The following block diagrams describe the components used by the client-side and server-side Provers when computing client proofs and rollup proofs respectively.



data-bus

hashing

Desirable properties

There are two main properties we might desire from the various hash functions defined in aztec:

Collision Resistance

Poseidon2 is the predominant hash function in the Aztec protocol. It is assumed to be collision resistant.

Domain Separators

To minimize the potential for collisions between distinct hashing contexts, all hashes are domain-separated by passing a `string` which describes the context of the hash. Each such string in this spec begins with the prefix `az_`, to domain-separate all hashes specifically to the Aztec protocol.

The strings provided in this spec are mere suggestions at this stage; we might find that the strings should be smaller bit-lengths, to reduce in-circuit constraints.

In the case of using Poseidon2 for hashing (which is the case for most hashing in the Aztec protocol), the string is converted from a big-endian byte representation into a `Field` element, and passed as a first argument into the hash. In the case of using non-algebraic hash functions (such as sha256), the string is converted from a big-endian byte representation into bits, and passed as the first bits into the hash. These details are conveyed more clearly as pseudocode in the relevant sections of the spec.

For some hashes there is further domain-separation. For example, [Merkle tree hashing](#) of the tree.

Pseudo-randomness

Sometimes we desire the output of a hash function to be pseudo-random.

Throughout the Aztec protocol, it is assumed that Poseidon2 can be used as a pseudo-random function.

Pseudo-randomness is required in cases such as:

- Fiat-Shamir challenge generation.
- Expanding a random seed to generate additional randomness.
 - See the derivation of [master secret keys](#).
- Deriving a nullifier, and siloing a nullifier.
 - See [deriving a nullifier](#).

poseidon2

pedersen

Merkle Trees

Notation

Orientation

A tree is always visualised as a "triangle" with its point at the top (the root) and its base at the bottom (the leaves). Like this: Δ .

Hopefully this gives a clear intuition whenever the terms "left", "right", "up", "down", "above", "below" are used when describing trees.

Arity

Trees in Aztec are currently all binary Merkle trees (2-ary).

Height

The `height` of a tree with l leaves is $\lceil \log_2(l) \rceil$.

Layers

The `layers` of a tree are enumerated from `0`. The leaves are at layer `0`; the root is at layer `height`.

Levels

Synonymous with `layers`.

Rows

Synonymous with [layers](#) and [levels](#).

Leaf Index

The leaves of a tree are indexed from `0`. The first, [left-most](#) leaf is at `leaf_index = 0`.

Node Index

All nodes of the tree (including the leaves) can be indexed. The method of indexing might depend on the algorithm being applied to the tree.

Path

The path from (or "of") a particular node is a vector of that node's ancestors. That is, the node's parent, then its parent's parent, and so on, all the way up to and including the root.

Sibling Path

The sibling path of a particular node is, loosely, a vector of the siblings of the nodes in its [path](#), except it also includes the node's sibling, and excludes the root (which has no sibling). The first element in the sibling path is the node's sibling. Then, the node's parent's sibling, then its parent's parent's sibling, and so on.

Membership Witness

The membership witness for a particular leaf, is the minimum data needed to prove

that leaf value's existence in the tree. That is:

- The leaf's [leaf index](#)
- The leaf's [sibling path](#)

(and the leaf value itself, of course, but we don't include that in this "membership witness" definition).

Hashing

Used for computing the parent nodes of all merkle trees.

```
enum TreeId {
    Archive,
    NoteHash,
    Nullifier,
    PrivateFunction,
    L1ToL2Msgs,
    PublicData
}

fn merkle_crh(
    tree_id: TreeId,
    layer: u64,
    left: Field,
    right: Field
) -> Field {
    let tree_id_domain_separator: string = match tree_id {
        TreeId::Archive => "archive",
        TreeId::NoteHash => "note_hash",
        TreeId::Nullifier => "nullifier",
        TreeId::PrivateFunction => "private_function",
        TreeId::L2ToL2Msgs => "l1_to_l2_msgs",
        TreeId::PublicData => "public_data",
    }
}
```

`tree_id` reflects the various [trees in the protocol](#). The `PrivateFunction` tree is discussed in the [contract classes](#) section. `layer` is the `layer` of the `left` and `right` children being hashed. For example, when hashing two leaves, `layer = 0`.

🔥 DANGER

- Q: Do we need the domain separator "az_merkle" + `tree_id`, for each of the trees?
- Q: do we need domain separation between different layers of the tree?
- Q: Can we optimise the two domain separators to take up 1 Field, instead of 2, or does squashing them together add too many constraints?
- Note: if it helps with optimisation, we can reduce the bit-length of the domain separator strings.
- Q: Can we specify the arguments to Poseidon as Fields, or do we need to specify them as bit-sequences?

Append-only Merkle Tree

TODO

Indexed Merkle Tree

TODO

Addresses and Keys

Aztec has no concept of externally-owned accounts. Every address identifies a smart contract in the network.

For end users to interact with the network, they'll most-likely need to [deploy](#) a so-called "account contract".

Addresses are then a commitment to a contract class, a list of constructor arguments, and a set of keys.

Keys in Aztec are used both for authorization and privacy. Authorization keys are managed by account contracts, and not mandated by the protocol. Each account contract may use different authorization keys, if at all, with different signing mechanisms.

Privacy keys are used for note encryption, tagging, and nullifying. These are also not enforced by the protocol. However, for facilitating compositability, the protocol enshrines a set of enshrined encryption and tagging mechanisms, that can be leveraged by applications as they interact with accounts.

The [requirements](#) section outlines the features that were sought when designing Aztec's addresses and keys. We then specify how [addresses](#) are derived, as well as the default way in which [keys](#) will be derived. The [precompiles](#) section describes enshrined contract addresses, with implementations defined by the protocol, used for note encryption and tagging.

Last, the [diversified and stealth accounts](#) sections describe application-level recommendations for diversified and stealth accounts.

Address

An address is computed as the hash of the following fields:

Requirements

Requirements which motivated Aztec's design for addresses and keys.

Default Keys Specification

Specification for default privacy keys format and derivation, and nullifier derivation.

Example Usage of Keys

4 items

Precompiles

Precompiled contracts, which borrow their name from Ethereum's, are contracts not deployed by users but defined at the protocol level. These contract instances and their c...

Diversified and Stealth Accounts

The keys specification describes derivation mechanisms for diversified and stealth public keys. However, the protocol requires users to interact with addresses.

Address

An address is computed as the hash of the following fields:

Field	Type	Description
salt	Field	User-generated pseudorandom value for uniqueness.
deployer	AztecAddress	Optional address of the deployer of the contract.
contract_class_id	Field	Identifier of the contract class for this instance.
initialization_hash	Field	Hash of the selector and arguments to the constructor.
portal_contract_address	EthereumAddress	Address of the L1 portal contract, zero if none.
public_keys_hash	Field	Hash of the struct of public keys used for encryption and nullifying by this contract, zero if no public keys.

Storing these fields in the address preimage allows any part of the protocol to check

them by recomputing the hash and verifying that the address matches. Examples of these checks are:

- Sending an encrypted note to an undeployed account, which requires the sender app to check the recipient's public key given their address. This scenario also requires the recipient to share with the sender their public key and rest of preimage.
- Having the kernel circuit verify that the code executed at a given address matches the one from the class.
- Asserting that the initialization hash matches the function call in the contract constructor.
- Checking the portal contract address when sending a cross-chain message.

⚠️ WARNING

We may remove the `portal_contract_address` as a first-class citizen.

The hashing scheme for the address should then ensure that checks that are more frequent can be done cheaply, and that data shared out of band is kept manageable. We define the hash to be computed as follows:

⚠️ WARNING

Some of these draft domain separators might be too many bits; they need to fit inside a single field element. Version numbers might not be needed until we roll the *next* version.

```
address_crh()  
    version: Field,  
    salt: Field,  
    deployer: AztecAddress,
```

The `public_keys` array can vary depending on the format of keys used by the address, but it is suggested it includes the master keys defined in the [keys section](#). For example:

```
let public_keys_hash: Field = poseidon2(  
    be_string_to_field("az_public_keys_hash"), // TODO: does this  
    need some unique ID, to disambiguate from other approaches people  
    might have for other public keys?  
  
    nullifier_pubkey.x,  
    nullifier_pubkey.y,  
    tagging_pubkey.x,  
    tagging_pubkey.y,  
    incoming_view_pubkey.x,  
    incoming_view_pubkey.y,  
    outgoing_view_pubkey.x,  
    outgoing_view_pubkey.y  
);
```

This recommended hash format is compatible with the [encryption precompiles](#) initially defined in the protocol and advertised in the canonical [registry](#) for private message delivery. An address that chooses to use a different format for its keys will not be compatible with apps that rely on the registry for note encryption. Nevertheless, new precompiles introduced in future versions of the protocol could use different public keys formats.

Requirements

Requirements for Keys

Scenario

A common illustration in this document is Bob sending funds to Alice, by:

- creating a "note" for her;
- committing to the contents of that note (a "note hash");
- inserting that note hash into a utxo tree;
- encrypting the contents of that note for Alice;
- optionally encrypting the contents of that note for Bob's future reference;
- optionally deriving an additional "tag" (a.k.a. "clue") to accompany the ciphertexts for faster note discovery;
- broadcasting the resulting ciphertext(s) (and tag(s));
- optionally identifying the tags;
- decrypting the ciphertexts; storing the note; and some time later spending (nullifying) the note.

Note: there is nothing to stop an app and wallet from implementing its own key derivation scheme. Nevertheless, we're designing a 'canonical' scheme that most developers and wallets can use.

Authorization keys

Aztec has native account abstraction, so tx authentication is done via an account contract, meaning tx authentication can be implemented however the user sees fit. That is, authorization keys aren't specified at the protocol level.

A tx authentication secret key is arguably the most important key to keep private, because knowledge of such a key could potentially enable an attacker to impersonate the user and execute a variety of functions on the network.

Requirements:

- A tx authentication secret key SHOULD NOT enter Aztec software, and SHOULD NOT enter a circuit.
 - Reason: this is just best practice.

Master & Siloed Keys

Requirements:

- All keys must be re-derivable from a single `seed` secret.
- Users must have the option of keeping this `seed` offline, e.g. in a hardware wallet, or on a piece of paper.
- All master keys (for a particular user) must be linkable to a single address for that user.
- For each contract, a siloed set of all secret keys MUST be derivable.
 - Reason: secret keys must be siloed, so that a malicious app circuit cannot access and emit (as an unencrypted event or as args to a public function) a user's master secret keys or the secret keys of other apps.
- Master *secret* keys must not be passed into an app circuit, except for precompiles.
 - Reason: a malicious app could broadcast these secret keys to the world.
- Siloed secret keys *of other apps* must not be passed into an app circuit.
 - Reason: a malicious app could broadcast these secret keys to the world.
- The PXE must prevent an app from accessing master secret keys.
- The PXE must prevent an app from accessing siloed secret keys that belong to another contract address.

- Note: To achieve this, the PXE simulator will need to check whether the bytecode being executed (that is requesting secret keys) actually exists at the contract address.
- There must be one and only one way to derive all (current*) master keys, and all siloed keys, for a particular user address.
 - For example, a user should not be able to derive multiple different outgoing viewing keys for a single incoming viewing key (note: this was a 'bug' that was fixed between ZCash Sapling and Orchard).
 - *"current", alludes to the possibility that the user might wish to support rotating keys, but only if one and only one set of keys is derivable as "current".
- All app-siloed keys can all be deterministically linked back to the user's address, without leaking important secrets to the app.

Security assumptions

- The Aztec private execution client (PXE), precompiled contracts (vetted application circuits), and the kernel circuit (a core protocol circuit) can be trusted with master secret keys (*except for* the tx authorization secret key, whose security assumptions are abstracted-away to wallet designers).

Encryption and decryption

Definitions (from the point of view of a user ("yourself")):

- Incoming data: Data which has been created by someone else, and sent to yourself.
- Outgoing data: Data which has been sent to somebody else, from you.
- Internal Incoming data: Data which has been created by you, and has been sent to yourself.
 - Note: this was an important observation by ZCash. Before this distinction, whenever a 'change' note was being created, it was being broadcast as

incoming data, but that allowed a 3rd party who was only meant to have been granted access to view "incoming" data (and not "outgoing" data), was also able to learn that an "outgoing" transaction had taken place (including information about the notes which were spent). The addition of "internal incoming" keys enables a user to keep interactions with themselves private and separate from interactions with others.

Requirements:

- A user can derive app-siloed incoming internal and outgoing viewing keys.
 - Reason: Allows users to share app-siloed keys to trusted 3rd parties such as auditors, scoped by app.
 - Incoming viewing keys are not considered for siloed derivation due to the lack of a suitable public key derivation mechanism.
- A user can encrypt a record of any actions, state changes, or messages, to *themselves*, so that they may re-sync their entire history of actions from their seed.

Nullifier keys

Derivation of a nullifier is app-specific; a nullifier is just a `field` (siloed by contract address), from the pov of the protocol.

Many private application devs will choose to inject a secret "nullifier key" into a nullifier. Such a nullifier key would be tied to a user's public identifier (e.g. their address), and that identifier would be tied to the note being nullified (e.g. the note might contain that identifier). This is a common pattern in existing privacy protocols. Injecting a secret "nullifier key" in this way serves to hide what the nullifier is nullifying, and ensures the nullifier can only be derived by one person (assuming the nullifier key isn't leaked).

| Note: not all nullifiers require injection of a secret *which is tied to a user's identity*

in some way. Sometimes an app will need just need a guarantee that some value will be unique, and so will insert it into the nullifier tree.

Requirements:

- Support use cases where an app requires a secret "nullifier key" (linked to a user identity) to be derivable.
 - Reason: it's a very common pattern.

Is a nullifier key *pair* needed?

I.e. do we need both a nullifier secret key and a nullifier public key? Zcash sapling had both, but Zcash orchard (an upgrade) replaced the notion of a keypair with a single nullifier key. The [reason](#) being:

- *"[The nullifier secret key's (nsk's)] purpose in Sapling was as defense-in-depth, in case RedDSA [(the scheme used for signing txs, using the authentication secret key ask)] was found to have weaknesses; an adversary who could recover ask would not be able to spend funds. In practice it has not been feasible to manage nsk much more securely than a full viewing key [(dk, ak, nk, ovk)], as the computational power required to generate Sapling proofs has made it necessary to perform this step [(deriving nk from nsk)] on the same device that is creating the overall transaction (rather than on a more constrained device like a hardware wallet). We are also more confident in RedDSA now."*

A nullifier public key might have the benefit (in Aztec) that a user could (optionally) provide their nullifier key nk to some 3rd party, to enable that 3rd party to see when the user's notes have been nullified for a particular app, without having the ability to nullify those notes.

- This presumes that within a circuit, the nk (not a public key; still secret!) would be derived from an nsk, and the nk would be injected into the nullifier.
- BUT, of course, it would be BAD if the nk were derivable as a bip32 normal child,

because then everyone would be able to derive the nk from the master key, and be able to view whenever a note is nullified!

- The nk would need to be a hardened key (derivable only from a secret).

Given that it's acceptable to ZCash Orchard, we accept that a nullifier master secret key may be 'seen' by Aztec software.

Auditability

Some app developers will wish to give users the option of sharing private transaction details with a trusted 3rd party.

Note: The [archive](#) will enable a user to prove many things about their transaction history, including historical encrypted logs. This feature will open up exciting audit patterns, where a user will be able to provably respond to questions without necessarily revealing their private data. However, sometimes this might be an inefficient pattern; in particular when a user is asked to prove a negative statement (e.g. "prove that you've never owned a rock NFT"). Proving such negative statements might require the user to execute an enormous recursive function to iterate through the entire tx history of the network, for example: proving that, out of all the encrypted events that the user *can* decrypt, none of them relate to ownership of a rock NFT. Given this (possibly huge) inefficiency, these key requirements include the more traditional ability to share certain keys with a trusted 3rd party.

Requirements:

- "Shareable" secret keys.
 - A user can optionally share "shareable" secret keys, to enable a 3rd party to decrypt the following data:
 - Outgoing data, across all apps
 - Outgoing data, siloed for a single app

- Incoming internal data, across all apps
 - Incoming internal data, siloed for a single app
 - Incoming data, across all apps
 - Incoming data, siloed for a single app, is not required due to lack of a suitable derivation scheme
- Shareable nullifier key.
 - A user can optionally share a "shareable" nullifier key, which would enable a trusted 3rd party to see *when* a particular note hash has been nullified, but would not divulge the contents of the note, or the circumstances under which the note was nullified (as such info would only be gleanable with the shareable viewing keys).
 - Given one (or many) shareable keys, a 3rd part MUST NOT be able to derive any of a user's other secret keys; be they shareable or non-shareable.
 - Further, they must not be able to derive any relationships *between* other keys.
- No impersonation.
 - The sharing of any (or all) "shareable" key(s) MUST NOT enable the trusted 3rd party to perform any actions on the network, on behalf of the user.
 - The sharing of a "shareable" outgoing viewing secret (and a "shareable" *internal* incoming viewing key) MUST NOT enable the trusted 3rd party to emit encrypted events that could be perceived as "outgoing data" (or internal incoming data) originating from the user.
- Control over incoming/outgoing data.
 - A user can choose to only give incoming data viewing rights to a 3rd party. (Gives rise to incoming viewing keys).
 - A user can choose to only give outgoing data viewing rights to a 3rd party. (Gives rise to outgoing viewing keys).
 - A user can choose to keep interactions with themselves private and distinct from the viewability of interactions with other parties. (Gives rise to *internal* incoming viewing keys).

Sending funds before deployment

Requirements:

- A user can generate an address to which funds (and other notes) can be sent, without that user having ever interacted with the network.
 - To put it another way: A user can be sent money before they've interacted with the Aztec network (i.e. before they've deployed an account contract). e.g their incoming viewing key can be derived.
- An address (user identifier) can be derived deterministically, before deploying an account contract.

Note Discovery

Requirements:

- A user should be able to discover which notes belong to them, without having to trial-decrypt every single note broadcasted on chain.
- Users should be able to opt-in to using new note discovery mechanisms as they are made available in the protocol.

Tag Hopping

Given that this is our best-known approach, we include some requirements relating to it:

Requirements:

- A user Bob can non-interactively generate a sequence of tags for some other user Alice, and non-interactively communicate that sequence of tags to Alice.
- If a shared secret (that is used for generating a sequence of tags) is leaked, Bob

can non-interactively generate and communicate a new sequence of tags to Alice, without requiring Bob nor Alice to rotate their keys.

- Note: if the shared secret is leaked through Bob/Alice accidentally leaking one of their keys, then they might need to actually rotate their keys.

Constraining key derivations

- An app has the ability to constrain the correct encryption and/or note discovery tagging scheme.
- An app can *choose* whether or not to constrain the correct encryption and/or note discovery tagging scheme.
 - Reason: constraining these computations (key derivations, encryption algorithms, tag derivations) will be costly (in terms of constraints), and some apps might not need to constrain it (e.g. zcash does not constrain correct encryption).

Rotating keys

- A user should be able to rotate their set of keys, without having to deploy a new account contract.
 - Reason: keys can be compromised, and setting up a new identity is costly, since the user needs to migrate all their assets. Rotating encryption keys allows the user to regain privacy for all subsequent interactions while keeping their identity.
 - This requirement causes a security risk when applied to nullifier keys. If a user can rotate their nullifier key, then the nullifier for any of their notes changes, so they can re-spend any note. Rotating nullifier keys requires the nullifier public key, or at least an identifier of it, to be stored as part of the note. Alternatively, this requirement can be removed for nullifier keys, which are not allowed to be rotated.

Diversified Keys

- Alice can derive a diversified address; a random-looking address which she can (interactively) provide to Bob, so that Bob may send her funds (and general notes).
 - Reason: By having the recipient derive a distinct payment address *per counterparty*, and then interactively provide that address to the sender, means that if two counterparties collude, they won't be able to convince the other that they both interacted with the same recipient.
- Random-looking addresses can be derived from a 'main' address, so that private to public function calls don't reveal the true `msg_sender`. These random-looking addresses can be provably linked back to the 'main' address.
 - | Note: both diversified and stealth addresses would meet this requirement.
- Distributing many diversified addresses must not increase the amount of time needed to scan the blockchain (they must all share a single set of viewing keys).

Stealth Addresses

Not to be confused with diversified addresses. A diversified address is generated by the recipient, and interactively given to a sender, for the sender to then use. But a stealth address is generated by the *sender*, and non-interactively shared with the recipient.

Requirement:

- Random-looking addresses can be derived from a 'main' address, so that private → public function calls don't reveal the true `msg_sender`. These random-looking addresses can be provably linked back to the 'main' address.
 - | Note: both diversified and stealth addresses would meet this requirement.
- Unlimited random-looking addresses can be non-interactively derived by a

sender for a particular recipient, in such a way that the recipient can use one set of keys to decrypt state changes or change states which are 'owned' by that stealth address.

Default Keys Specification

Cheat Sheet

The protocol does not enforce the usage of any of the following keys, and does not enforce the keys to conform to a particular length or algorithm. Users are expected to pick a set of keys valid for the encryption and tagging precompile they choose for their account.

Cat.	Key	Derivation	Link
Seed	seed	$\xleftarrow{\$} \mathbb{F}$	Seed
	sk	$\xleftarrow{\$} \mathbb{F}$	Master Secret Key
Master Secret Keys	nsk_m	$\text{poseidon2}(\text{"az_nsk_m"}, \text{sk})$	Master Nullifier Secret Key
	ovsk_m	$\text{poseidon2}(\text{"az_ovsk_m"}, \text{sk})$	Master Outgoing Viewing Secret Key
	ivsk_m	$\text{poseidon2}(\text{"az_ivsk_m"}, \text{sk})$	Master Incoming Viewing Secret Key
	tsk_m	$\text{poseidon2}(\text{"az_tsk_m"}, \text{sk})$	Master Tagging Secret Key
Master Public Keys	Npk_m	$\text{nsk}_m \cdot G$	Master Nullifier Public Key
	Ovpk_m	$\text{ovsk}_m \cdot G$	Master Outgoing Viewing Public Key
	Ivpk_m	$\text{ivsk}_m \cdot G$	Master Incoming Viewing Public Key
	Tpk_m	$\text{tsk}_m \cdot G$	Master Tagging Public Key

Cat.	Key	Derivation	Link
Hardened App-Siloed Secret Keys	nsk_{app}	poseidon2("az_nsk_app", app_address, nsk_m)	Hardened, App-siloed Nullifier Secret Key
	$ovsk_{app}$	poseidon2("az_ovsk_app", app_address, $ovsk_m$)	Hardened, App-siloed Outgoing Viewing Secret Key
Other App-siloed Keys	Nk_{app}	poseidon2("az_nk_app", nsk_{app})	App-siloed Nullifier Key

Colour Key

- **green** = Publicly shareable information.
- **red** = Very secret information. A user MUST NOT share this information.
 - TODO: perhaps we distinguish between information that must not be shared to prevent theft, and information that must not be shared to preserve privacy?
- **orange** = Secret information. A user MAY elect to share this information with a *trusted* 3rd party, but it MUST NOT be shared with the wider world.
- **violet** = Secret information. Information that is shared between a sender and recipient (and possibly with a 3rd party who is entrusted with viewing rights by the recipient).

Diagrams



Diagram is out of date vs the content on this page



The red boxes are uncertainties, which are explained later in this doc.

Preliminaries

\mathbb{F}_r denotes the AltBN254 scalar field (i.e. the Grumpkin base field).

\mathbb{F}_q denotes the AltBN254 base field (i.e. the Grumpkin scalar field).

Let $\mathbb{G}_{\text{Grumpkin}}$ be the Grumpkin elliptic curve group ($E(\mathbb{F}_r)$).

Let $G \in \mathbb{G}_{\text{Grumpkin}}$ be a generator point for the public key cryptography outlined below. TODO: decide on how to generate this point.

Elliptic curve operators $+$ and \cdot are used to denote addition and scalar multiplication, respectively.

$\text{poseidon2} : \mathbb{F}_r^t \rightarrow \mathbb{F}$ is the Poseidon2 hash function (and t can take values as per the [Poseidon2 spec](#)).

Note that $q > r$. Below, we'll often define secret keys as an element of \mathbb{F}_r , as this is most efficient within a snark circuit. We'll then use such secret keys in scalar multiplications with Grumpkin points ($E(\mathbb{F}_r)$) whose affine points are of the form $\mathbb{F}_r \times \mathbb{F}_r$. Strictly speaking, such scalars in Grumpkin scalar multiplication should be in \mathbb{F}_q .

A potential consequence of using elements of \mathbb{F}_r as secret keys could be that the resulting public keys are not uniformly-distributed in the Grumpkin group, so we should check this. The distribution of such public keys will have a statistical distance of $\frac{2(q-r)}{q}$ from uniform. It turns out that $\frac{1}{2^{126}} < \frac{2(q-r)}{q} < \frac{1}{2^{125}}$, so the statistical distance from uniform is broadly negligible, especially considering that the AltBN254 curve has fewer than 125-bits of security.

Key Derivation

Derive Master Secret Key from Secret Key

```
derive_master_secret_key_from_secret_key : string ×  $\mathbb{F}_r \rightarrow \mathbb{F}_r$ 
derive_master_secret_key_from_secret_key(domain_separator_string, secret_key)
:= poseidon2(be_string_to_field(domain_separator_string), secret_key)
```

Note: Here, poseidon2 is assumed to be a pseudo-random function.

Derive Hardened App-siloed Secret Key

```
derive_hardened_app_siloed_secret_key : string ×  $\mathbb{F}_r \times \mathbb{F}_r \rightarrow \mathbb{F}_r$ 
derive_hardened_app_siloed_secret_key(domain_separator_string, app_address, parent_secret_key)
:= poseidon2(be_string_to_field(domain_separator_string), app_address, parent_secret_key)
```

Note: Here, poseidon2 is assumed to be a pseudo-random function.

Note: this deviates significantly from the 'conventional' [BIP-32 style method](#) for deriving a "hardened child secret key", to reduce complexity and as an optimization. Such a deviation will need to be validated as secure. In particular:

- the notion of a "chain code" has been removed;
- the notion of an "index" has been replaced by an app_address;
- HMAC-SHA512 has been replaced with Poseidon2. Note: we don't need a 512-bit output, since we've removed the notion of a "chain code", and so we don't need to split the output of the Poseidon2 function into two outputs.

Derive Public Key (from Secret Key)

$$\begin{aligned}\text{derive_public_key} : \mathbb{F}_r &\rightarrow \mathbb{G}_{\text{Grumpkin}} \\ \text{derive_public_key}(\text{secret_key}) &:= \text{secret_key} \cdot G\end{aligned}$$

Seed

A seed secret from which all of a user's other keys may be derived. The **seed** can live on an offline device, such as a hardware wallet.

$$\text{seed} \xleftarrow{\$} \mathbb{F}_r$$

Master Secret Key

This **sk** must never enter a circuit. A user or wallet may wish to derive this **sk** from a cold wallet **seed**.

$$\text{sk} \xleftarrow{\$} \mathbb{F}_r$$

Note: Often $\text{sk} = \text{hash}(\text{seed})$ for some hardware-wallet-supported hash function, would be recommended. Although, care would need to be taken if the hardware wallet doesn't support hashing directly to \mathbb{F}_r , since a truncated hash output could be non-uniformly distributed in \mathbb{F}_r .
For example, if the hardware wallet only supports sha256, then it would not be acceptable to compute **sk** as $\text{sha256}(\text{seed}) \bmod r$, since the resulting output (of reducing a 256-bit number modulo r) would be biased towards smaller values in \mathbb{F}_r . More uniformity might be achieved by instead computing **sk** as $(\text{sha256}(\text{seed}, 1) \parallel \text{sha256}(\text{seed}, 2)) \bmod r$, for example, as a modulo reduction of a 512-bit number is closer to being uniformly distributed in \mathbb{F}_r .

This note is informal, and expert advice should be sought before adopting this approach.

Nullifier Keys

[App-siloed Nullifier Keys](#) can be used by app developers when deriving their apps' nullifiers. By inserting some secret nullifier key into a nullifier's preimage, it makes the resulting nullifier look random, meaning observers cannot determine which note has been nullified.

Note that not all nullifiers will require a secret key in their computation, e.g. plume nullifiers, or state variable initialization nullifiers. But the keys outlined in this section should prove useful to many app developers.

Master Nullifier Secret Key

$$nsk_m \in \mathbb{F}_r$$

$nsk_m = \text{derive_master_secret_key_from_secret_key}(\text{"az_nsk_m"}, \text{seed})$

See [derive_master_secret_key_from_secret_key](#).

nsk_m MUST NOT enter an app circuit.

nsk_m MAY enter the kernel circuit.

Master Nullifier Public Key

The Master Nullifier Public Key is only included so that other people can derive the user's address from some public information (i.e. from Npk_m), in such a way that nsk_m is tied to the user's address.

$$Npk_m \in \mathbb{G}_{\text{Grumpkin}}$$

$Npk_m = \text{derive_public_key}(nsk_m)$

See [derive_public_key](#).

App-siloed Nullifier Secret Key

The App-siloed Nullifier Secret Key is a **hardened** child key, and so is only derivable by the owner of the master nullifier secret key. It is hardened so as to enable the nsk_{app} to be passed into an app circuit, without the threat of nsk_m being reverse-derivable by a malicious app. Only when an app-siloed public key needs to be derivable by the general public is a normal (non-hardened) key derivation scheme used.

$$nsk_{app} \in \mathbb{F}_r$$

$nsk_{app} = \text{derive_hardened_app_siloed_secret_key}(\text{"az_nsk_app"}, \text{app_address}, nsk_m)$

See [derive_hardened_app_siloed_secret_key](#).

App-siloed Nullifier Key

If an app developer thinks some of their users might wish to have the option to enable some *trusted* 3rd party to see when a particular user's notes are nullified, then this nullifier key might be of use. This Nk_{app} can be used in a nullifier's preimage, rather than nsk_{app} in such cases, to enable said 3rd party to brute-force identify nullifications.

Note: this key can be optionally shared with a trusted 3rd party, and they would not be able to derive the user's secret keys.

Note: knowledge of this key enables someone to identify when an emitted nullifier belongs to the user, and

to identify which note hashes have been nullified.

Note: knowledge of this key would not enable a 3rd party to view the contents of any notes; knowledge of the $ivsk$ / $ovsk_{app}$ would be needed for that.

Note: this is intentionally not named as a "public" key, since it must not be shared with the wider public.

$$\begin{aligned} Nk_{app} &\in \mathbb{F}_r \\ Nk_{app} &= \text{poseidon2}(\text{"az_nk_app"}, nsk_{app}) \end{aligned}$$

⚠ TODO

We could also have derived Nk_{app} as $nsk_{app} \cdot G$, but this would have resulted in a Grumpkin point, which is more cumbersome to insert into the preimage of a nullifier. We might still change our minds to adopt this scalar-multiplication approach, since it might enable us to prove knowledge of nsk_m to the app circuit without having to add key derivation logic to the kernel circuit.

Outgoing Viewing Keys

App-siloed Outgoing Viewing Secret Keys can be used to derive ephemeral symmetric encryption keys, which can then be used to encrypt/decrypt data which *the user has created for their own future consumption*. i.e. these keys are for decrypting "outgoing" data from the pov of a sender. This is useful if the user's DB is wiped, and they need to sync from scratch (starting with only **seed**).

Master Outgoing Viewing Secret Key

$$\begin{aligned} ovsk_m &\in \mathbb{F}_r \\ ovsk_m &= \text{derive_master_secret_key_from_seed}(\text{"az_ovsk_m"}, \text{seed}) \end{aligned}$$

See [derive_master_secret_key_from_seed](#).

$ovsk_m$ MUST NOT enter an app circuit.

$ovsk_m$ MAY enter the kernel circuit.

Master Outgoing Viewing Public Key

The Master Outgoing Viewing Public Key is only included so that other people can derive the user's address from some public information (i.e. from $Ovpk_m$), in such a way that $ovsk_m$ is tied to the user's address.

$$\begin{aligned} Ovpk_m &\in \mathbb{G}_{\text{Grumpkin}} \\ Ovpk_m &= \text{derive_public_key}(ovsk_m) \end{aligned}$$

See [derive_public_key](#).

App-siloed Outgoing Viewing Secret Key

The App-siloed Outgoing Viewing Secret Key is a hardened child key, and so is only derivable by the owner of the Master Outgoing Viewing Secret Key. It is hardened so as to enable the $ovsk_{app}$ to be passed into an app circuit, without the threat of $ovsk_m$ being reverse-derivable by a malicious app. Only when an app-siloed public key needs to be derivable by the general public is a normal (non-hardened) key derivation scheme used.

$$ovsk_{app} \in \mathbb{F}_r$$

$$ovsk_{app} = \text{derive_hardened_app_siloed_secret_key}(\text{"az_ovsk_app"}, \text{app_address}, ovsks_m)$$

See [derive_hardened_app_siloed_secret_key](#).

Incoming Viewing Keys

If a sender wants to send some recipient a private message or note, they can derive an ephemeral symmetric encryption key from the recipient's Master Incoming Viewing Public Key. I.e. these keys are for decrypting "incoming" data from the pov of a recipient.

Master Incoming Viewing Secret Key

$$ivsk_m \in \mathbb{F}_r$$

$$ivsk_m = \text{derive_master_secret_key_from_seed}(\text{"az_ivsk_m"}, \text{seed})$$

See [derive_master_secret_key_from_seed](#).

$ivsk_m$ MUST NOT enter an app circuit.

Master Incoming Viewing Public Key

The Master Incoming Viewing Public Key can be used by a sender to encrypt messages and notes to the owner of this key.

$$Ivpk_m \in \mathbb{G}_{\text{Grumpkin}}$$

$$Ivpk_m = \text{derive_public_key}(ivsk_m)$$

See [derive_public_key](#).

App-siloed Incoming Viewing Secret Key

An App-siloed Incoming Viewing Secret Key is not prescribed in this spec, because depending on how an app

developer wishes to make use of such a key, it could have implications on the security of the Master Incoming Viewing Secret Key.

| TODO: more discussion needed here, to explain everything we've thought about.

Tagging Keys

The "tagging" key pair can be used to flag "this ciphertext is for you", without requiring decryption.

Master Tagging Secret Key

$$\mathbf{tsk}_m \in \mathbb{F}_r$$

$\mathbf{tsk}_m = \text{derive_master_secret_key_from_seed}(\text{"az_tvsk_m"}, \mathbf{seed})$

| See [derive_master_secret_key_from_seed](#).

| \mathbf{ivsk}_m MUST NOT enter an app circuit.

Master Tagging Public Key

$$\mathbf{Tpk}_m \in \mathbb{G}_{\text{Grumpkin}}$$

$\mathbf{Tpk}_m = \text{derive_public_key}(\mathbf{tsk}_m)$

| See [derive_public_key](#).

Acknowledgements

Much of this is inspired by the [ZCash Sapling and Orchard specs](#).

nullifier

Deriving a nullifier within an app contract

Let's assume a developer wants a nullifier of a note to be derived as:

```
nullifier = h(note_hash, nullifier_key);
```

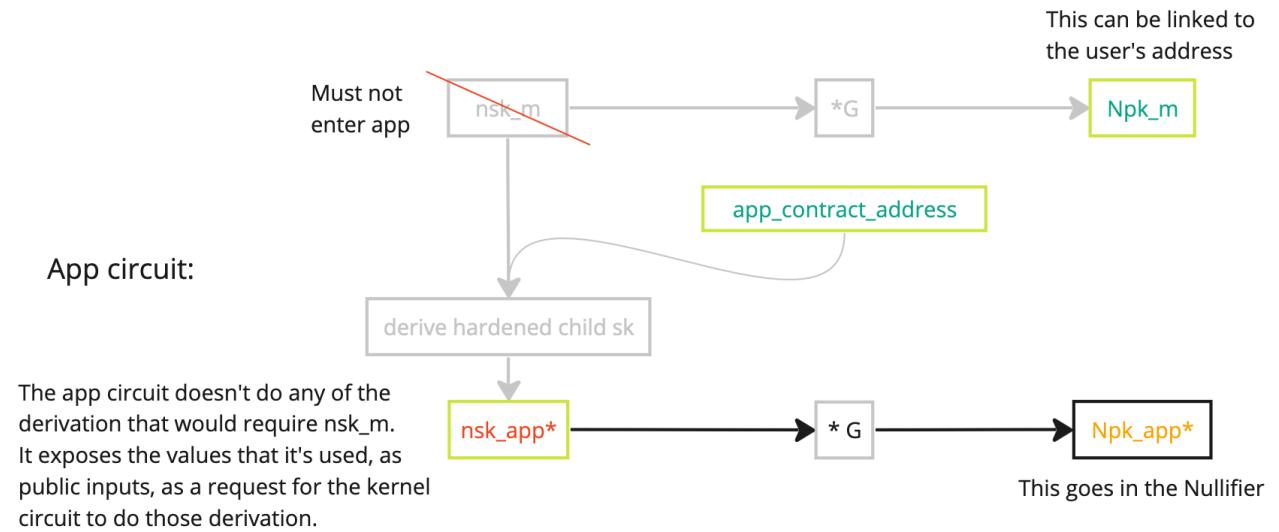
... where the `nullifier_key` (Nk_{app}) belongs to the 'owner' of the note, and where the 'owner' is some `address`.

Here's example for how an app circuit *could* constrain the nullifier key to be correct:

Diagram

It's easiest to take a look at this first:

Option 1



Within the app circuit

Within the app, we can prove links between:

- the user's `nsk_app` and their `Nk_app`; and between
- the user's `Npk_m` and their `address`.

The link that's missing is to prove that `Npk_m` relates to `nsk_app`. To compute this missing link requires the `nsk_m`, which MUST NOT be passed into an app circuit, and may only be passed into a kernel circuit. See the next 'Within the kernel circuit' section for details of this logic.

The logic

```

 $Nk_{app}$  = poseidon2( $nsk_{app}$ )
nullifier = poseidon2(note_hash,  $Nk_{app}$ )
public_keys_hash = poseidon2(be_string_to_field("az_public_keys_hash"),  $Npk_m$ ,  $Tpk_m$ ,  $Ivpk_m$ ,  $Ovpk_m$ )
address = poseidon2(be_string_to_field("az_contract_address_v1"), public_keys_hash, partial_address)

```

Note: the passing of points directly into the poseidon function is lazy notation: the keys would need to be serialized appropriately as fields into the poseidon function.

Recall an important point: the app circuit MUST NOT be given nsk_m . Indeed, nsk_{app} is derived (see earlier) as a *hardened* child of nsk_m , to prevent nsk_m from being reverse-derived by a malicious circuit. The linking of nsk_{app} to nsk_m is deferred to the kernel circuit (which can be trusted more so than an app).

Recall also: Nk_{app} is used (instead of nsk_{app}) solely as a way of giving the user the option of sharing Nk_{app} with a trusted 3rd party, to give them the ability to view when a note has been nullified (although I'm not sure how useful this is, given that it would require brute-force effort from that party to determine which note hash has been nullified, with very little additional information).

The app circuit exposes, as public inputs, a "nullifier key validation request":

```

let nullifier_validation_request = KeyValidationRequest {
    app_address: app_address,
    claimed_hardened_child_sk: nsk_app,
    claimed_parent_pk: Npk_m,
}

```

Within the Kernel Circuit

The kernel circuit can then validate the request (having been given nsk_m as a private input to the kernel circuit):

```

 $nsk_{app}$  = derive_hardened_app_siloed_secret_key("az_nsk_app", app_address,  $nsk_m$ )
 $Npk_m$  =  $nsk_m \cdot G$ 
 $nsk_{app}$  == claimed_hardened_child_sk
 $Npk_m$  == claimed_parent_pk

```

If the kernel circuit succeeds in these calculations, then the Nk_{app} has been validated as having a known secret key, and belonging to the *address*.

diversified-and-stealth-keys

Deriving diversified public keys

A diversified public key can be derived from Alice's keys, to enhance Alice's transaction privacy. If Alice's counterparties' databases are compromised, it enables Alice to retain privacy from such leakages. Diversified public keys are used for generating diversified addresses.

Basically, Alice must personally derive and provide Bob and Charlie with random-looking addresses (for Alice). Because Alice is the one deriving these Diversified Addresses (they can *only* be derived by Alice), if Bob and Charlie chose to later collude, they would not be able to convince each-other that they'd interacted with Alice.

This is not to be confused with 'Stealth Addresses', which 'flip' who derives: Bob and Charlie would each derive a random-looking Stealth Address for Alice. Alice would then discover her new Stealth Addresses through decryption.

All of the key information below is Alice's

Alice derives a 'diversified' incoming viewing public key, and sends it to Bob:

Thing	Derivation	Name	Comments
d	$\xleftarrow{rand} \mathbb{F}$	diversifier	
G_d	$d \cdot G$	diversified generator	
$Ivpk_{m,d}$	$ivsk_m \cdot G_d$	Diversified incoming viewing public key	

Notice: when $d = 1$, $Ivpk_{m,d} = Ivpk_m$. Often, it will be unnecessary to diversify the below data, but we keep d around for the most generality.

Deriving stealth public keys

All of the key information below is Alice's

Stealth Public Keys are used for generating Stealth Addresses. For Bob to derive a Stealth Address for Alice, Bob derives:

Thing	Derivation	Name	Comments
d	Given by Alice	(Diversifier)	Remember, in most cases, $d = 1$ is sufficient.
G_d	$d \cdot G$	(Diversified) generator	Remember, when $d = 1$, $G_d = G$.
$esk_{stealth}$	$\xleftarrow{rand} \mathbb{F}$	ephemeral secret, for deriving the stealth key shared secret	
$Epk_{d,stealth}$	$esk_{stealth} \cdot G_d$	(Diversified) Ephemeral public key, for deriving the stealth key shared secret	

Thing	Derivation	Name	Comments
$S_{m,\text{stealth}}$	$\textcolor{red}{esk}_{\text{stealth}} \cdot \textcolor{violet}{Ivpk}_{m,d}$	Stealth key shared secret	
h_{stealth}	$\text{pos2}(\text{"az_stealth_key"}, S_{m,\text{stealth}})$	stealth key	
$Ivpk_{m,d,\text{stealth}}$	$h_{\text{stealth}} \cdot G_d + Ivpk_{m,d}$	(Diversified) Stealth viewing public key	

Having derived a Stealth Address for Alice, Bob can now share it with Alice as follows:

Thing	Derivation	Name	Comments
$\text{tag}_{m,i}^{Bob \rightarrow Alice}$	See earlier in this doc.		<p>Derive the next tag in the $Bob \rightarrow Alice$ sequence.</p> <p>Note: we illustrate with a <i>master</i> tag sequence, but an app-specific tag sequence could also be used (in</p>

Thing	Derivation	Name	Comments
			which case an encryption of the app_address in a ciphertext header wouldn't be required; it could just be inferred from the tag used).
$\textcolor{red}{esk}_{\text{header}}$	$\xleftarrow{\textit{rand}} \mathbb{F}$	ephemeral secret key, for deriving the ciphertext header shared secret	
$Epk_{d,\text{header}}$	$\textcolor{red}{esk}_{\text{header}} \cdot G_d$	(Diversified) Ephemeral public key, for deriving the ciphertext	

Thing	Derivation	Name	Comments
		header shared secret	
$S_{m,header}$	$esk_{header} \cdot Ivpk_m$	Ciphertext header shared secret	TODO: we might need to use a different ephemeral keypair from the one used to derive the stealth address.
$h_{m,enc,header}$	$\text{pos2}(\text{"az_enc_key"}, S_{m,header})$	ciphertext header encryption key	
ciphertext_header	$\text{encrypt}_{h_{m,enc,header}}^{Ivpk_m}(\text{app_address})$		TODO: diversify this?
payload	$[\ tag_{m,i}^{Bob \rightarrow Alice}, Epk_{d,header},$ $\text{ciphertext_header},$ $Epk_{d,stealth}]$		

Alice can learn about her new Stealth Address as follows. First, she would identify the transaction has intended for her, either by observing $tag_{m,i}^{Bob \rightarrow Alice}$ on-chain herself (and then downloading the rest of the payload which accompanies the tag), or by making a

privacy-preserving request to a server, to retrieve the payload which accompanies the tag. Assuming the **payload** has been identified as Alice's, we proceed:

Thing	Derivation	Name
$S_{m,header}$	$ivsk_m \cdot Epk_{d,header}$	Ciphertext header shared secret
$h_{m,enc,header}$	$\text{pos2}(\text{"az_enc_key"}, S_{m,header})$	ciphertext header encryption key
app_address	$\text{decrypt}_{h_{m,enc,header}}^{ivsk_m}(\text{ciphertext_header})$	
$ivsk_m$	See derivations above. Use the decrypted app_address in the derivation.	app-specific incoming viewing secret key
$S_{m,stealth}$	$ivsk_m \cdot Epk_{d,stealth}$	Stealth key shared secret
$h_{stealth}$	$\text{pos2}(\text{"az_stealth_key"}, S_{m,stealth})$	stealth key
$ivsk_{m,stealth}$	$h_{stealth} + ivsk_m$	
$Ivpk_{m,d,stealth}$	$ivsk_{m,stealth} \cdot G_d$	(Diversified) Stealth viewing public key

Handshaking for tag-hopping

Deriving a sequence of tags for tag-hopping.

Deriving a sequence of tags between Alice and Bob across all apps

For Bob to derive a shared secret for Alice:

Thing	Derivation	Name	Comments
esk_{hs}	$\xleftarrow{rand} \mathbb{F}$	ephemeral secret key, for handshaking	hs = handshake.
Epk_{hs}	$esk_{hs} \cdot G$	Ephemeral public key, for handshaking	
$S_{m,tagging}^{Bob \rightarrow Alice}$	$esk_{hs} \cdot Ivpk_m$	Shared secret, for tagging	Here, we're illustrating the derivation of a shared secret (for tagging) using <i>master keys</i> .

Having derived a Shared Secret, Bob can now share it with Alice as follows:

Thing	Derivation	Name	Comments
Tag_{hs}	$\text{esk}_{hs} \cdot \text{Tpk}_m$	Handshake message identification tag	<p>Note: the tagging public key Tpk_m exists as an optimization, seeking to make brute-force message identification as fast as possible. In many cases, handshakes can be performed offchain via traditional web2 means, but in the case of on-chain handshakes, we have no preferred alternative over simply brute-force attempting to reconcile every 'Handshake message identification tag'. Note: this optimization reduces the recipient's work by 1 cpu-friendly hash per message (at the cost of 255-bits to broadcast a compressed encoding of Tag_{hs}). We'll need to decide whether this is the right speed/communication trade-off.</p>
payload	$[\text{Tag}_{hs}, \text{Epk}_{hs}]$	Payload	<p>This can be broadcast via L1. Curve points can be compressed in the payload.</p>

Alice can identify she is the intended the handshake recipient as follows:

Thing	Derivation	Name	Comments
Tag_{hs}	$tsk_m \cdot Epk_{hs}$	Handshake message identification tag	<p>Alice can extract Tag_{hs} and Epk_{hs} from the payload and perform this scalar multiplication on every handshake message. If the computed Tag_{hs} value matches that of the payload, then the message is intended for Alice.</p> <p>Clearly, handshake transactions will need to be identifiable as such (to save Alice time), e.g. by revealing the contract address of some canonical handshaking contract alongside the payload.</p> <p>Recall: this step is merely an optimization, to enable Alice to do a single scalar multiplication before moving on (in cases where she is not the intended recipient).</p>

If Alice successfully identifies that she is the intended the handshake recipient, she can proceed with deriving the shared secret (for tagging) as follows:

Thing	Derivation	Name	Comments
$S_{m,tagging}^{Bob \rightarrow Alice}$	$ivsk_m \cdot Epk_{hs}$	Shared secret, for tagging	

A sequence of tags can then be derived by both Alice and Bob as:

Thing	Derivation	Name	Comments
$\text{tag}_{m,i}^{Bob \rightarrow Alice}$	$\text{pos2}(\text{"az_tag_ss_m"}, \text{S}_{m,\text{tagging}}^{Bob \rightarrow Alice}, i)$	The i-th tag in the sequence.	

This tag can be used as the basis for note retrieval schemes. Each time Bob sends Alice a **ciphertext**, he can attach the next unused $\text{tag}_{m,i}^{Bob \rightarrow Alice}$ in the sequence. Alice - who is also able to derive the next $\text{tag}_{m,i}^{Bob \rightarrow Alice}$ in the sequence - can make privacy-preserving calls to a server, requesting the **ciphertext** associated with a particular $\text{tag}_{m,i}^{Bob \rightarrow Alice}$.

The colour key isn't quite clear for $\text{tag}_{m,i}^{Bob \rightarrow Alice}$. It will be a publicly-broadcast piece of information, but no one should learn that it relates to Bob nor Alice (except perhaps some trusted 3rd party whom Alice has entrusted with her ivsk_m).

Deriving a sequence of tags from Bob to himself across all apps

The benefit of Bob deriving a sequence of tags for himself, is that he can re-sync his *outgoing* transaction data more quickly, if he ever needs to in future.

This can be done by either:

- Copying the approach used to derive a sequence of tags between Bob and Alice (but this time do it between Bob and Bob, and use Bob's outgoing keys).
- Generating a very basic sequence of tags $\text{tag}_{app,i}^{Bob \rightarrow Bob} = \text{pos2}(\text{"az_tag_ovsk_app"}, \text{ovsk}_{app}, i)$ (at the app level) and $\text{tag}_{m,i}^{Bob \rightarrow Bob} = \text{pos2}(\text{"az_tag_ovsk_m"}, \text{ovsk}_m, i)$ (at the master level).

- Note: In the case of deriving app-specific sequences of tags, Bob might wish to also encrypt the app*address as a ciphertext header (and attach a master tag $\text{tag} * m, i^{Bob \rightarrow Bob}$), to remind himself of the apps that he should derive tags *for*.

encrypt-and-tag

Encrypt and tag an incoming message

Bob wants to send Alice a private message, e.g. the contents of a note, which we'll refer to as the **plaintext**. Bob and Alice are using a "tag hopping" scheme to help with note discovery. Let's assume they've already handshaked to establish a shared secret $S_{m,tagging}^{Bob \rightarrow Alice}$, from which a sequence of tags $tag_{m,i}^{Bob \rightarrow Alice}$ can be derived.

Thing	Derivation	Name	Comments
d	Given by Alice	(Diversifier)	Remember, in most cases, $d = 1$ is sufficient.
G_d	$d \cdot G$	(Diversified) generator	Remember, when $d = 1$, $G_d = G$.
esk_{header}	$\xleftarrow{rand} \mathbb{F}$	ephemeral secret key	
$Epk_{d,header}$	$esk_{header} \cdot G_d$	(Diversified) Ephemeral public key	
$S_{m,header}$	$esk_{header} \cdot Ivpk_m$	Shared secret, for ciphertext header encryption	TODO: can we use the same ephemeral keypair for both the ciphertext header and the ciphertext? TODO: diversify the $Ivpk_m$?
$h_{m,enc,header}$	$h("?", S_{m,header})$	Ciphertext header	

Thing	Derivation	Name	Comments
		encryption key	
ciphertext_header	$enc_{\text{hm}, \text{enc}, \text{header}}^{Ivpk_m}(\text{app_address})$	Ciphertext header	
esk	$\xleftarrow{\text{rand}} \mathbb{F}$	ephemeral secret key	
Epk_d	$esk \cdot G_d$	(Diversified) Ephemeral public key	
$S_{app, enc}$	$esk \cdot Ivpk_{app, d, stealth}$	Shared secret, for ciphertext encryption	
$h_{app, enc}$	$h("?", S_{app, enc})$	Incoming data encryption key	
ciphertext	$enc_{h_{app, enc}}^{Ivpk_{app, d, stealth}}(\text{plaintext})$	Ciphertext	
payload	$[\text{tag}_{m,i}^{Bob \rightarrow Alice}, \text{ciphertext_header}, \text{ciphertext}, Epk_{d, header}, Epk_d]$	Payload	

Alice can learn about her new payload as follows. First, she would identify the transaction has intended for her, either by observing $\text{tag}_{m,i}^{Bob \rightarrow Alice}$ on-chain herself (and then downloading the rest of the payload which accompanies the tag), or by making a privacy-preserving request to a server,

to retrieve the payload which accompanies the tag. Assuming the **payload** has been identified as Alice's, and retrieved by Alice, we proceed.

Given that the tag in this illustration was derived from Alice's master key, the tag itself doesn't convey which app_address to use, to derive the correct app-siloed incoming viewing secret key that would enable decryption of the ciphertext. So first Alice needs to decrypt the **ciphertext_header** using her master key:

Thing	Derivation	Name
$S_{m,header}$	$ivsk_m \cdot Epk_{d,header}$	Shared secret, for encrypting the ciphertext header
$h_{m,enc,header}$	$h("?", S_{m,header})$	Incoming encryption key
app_address	$decrypt^{ivsk_m}_{h_{m,enc,header}}(\text{ciphertext_header})$	App address
$ivsk_{stealth}$	See derivations above. Use the decrypted app_address.	Incoming viewing secret key
$S_{app,enc}$	$ivsk_{stealth} \cdot Epk_d$	Shared secret, for ciphertext encryption
$h_{app,enc}$	$h("?", S_{app,enc})$	Ciphertext encryption key
plaintext	$decrypt^{ivsk_{stealth}}_{h_{app,enc}}(\text{ciphertext})$	Plaintext

Encrypt and tag an outgoing message

Bob wants to send himself a private message (e.g. a record of the outgoing notes that he's created for other people) which we'll refer to as the **plaintext**. Let's assume Bob has derived a sequence of tags $\text{tag}_{m,i}^{Bob \rightarrow Alice}$ for himself (see earlier).

Note: this illustration uses *master* keys for tags, rather than app-specific keys for tags. App-

specific keys for tags could be used instead, in which case a 'ciphertext header' wouldn't be needed for the 'app_address', since the address could be inferred from the tag.

Note: rather than copying the 'shared secret' approach of Bob sending to Alice, we can cut a corner (because Bob is the sender and recipient, and so knows his own secrets).

Note: if Bob has sent a private message to Alice, and he also wants to send himself a corresponding message:

- he can likely re-use the ephemeral keypairs for himself.
- he can include esk in the plaintext that he sends to himself, as a way of reducing the size of his ciphertext (since the esk will enable him to access all the information in the ciphertext that was sent to Alice).

Note: the violet symbols should actually be orange here.

Thing	Derivation	Name	Comments
d	Given by Alice	(Diversifier)	Remember, in most cases, $d = 1$ is sufficient.
G_d	$d \cdot G$	(Diversified) generator	Remember, when $d = 1$, $G_d = G$.
esk_{header}	$\xleftarrow{rand} \mathbb{F}$	ephemeral secret key	
$Epk_{d,header}$	$esk_{header} \cdot G_d$	(Diversified) Ephemeral public key	
$h_{m,enc,header}$	$h("?", ovsk_m, Epk_{d,header})$	Header encryption key	This uses a master secret key $ovsk_m$, which MUST NOT be given to an app nor to an app circuit. However, it can still be given to a trusted precompile, which can handle this

Thing	Derivation	Name	Comments
			derivation securely.
ciphertext_header	$enc_{\text{hm}, \text{enc}, \text{header}}(\text{app_address})$	Ciphertext header encryption key	
esk	$\xleftarrow{\text{rand}} \mathbb{F}$	ephemeral secret key	
Epk_d	$esk \cdot G_d$	(Diversified) Ephemeral public key	
$h_{\text{app}, \text{enc}}$	$h("?", ovsk_{\text{app}}, Epk_d)$	Outgoing data encryption key	Since $ovsk_{\text{app}}$ is a <i>hardened</i> app-siloed secret key, it may be safely given to the dapp or passed into the app's circuit.
ciphertext	$enc_{\text{happ}, \text{enc}}(\text{plaintext})$	Ciphertext	
payload	$[\text{tag}_{m,i}^{Bob \rightarrow Bob}, \text{ciphertext_header}, \text{ciphertext}, Epk_{d, \text{header}}, Epk_d]$	Payload	

Alice can learn about her new payload as follows. First, she would identify the transaction has intended for her, either by observing $\text{tag}_{m,i}^{Bob \rightarrow Alice}$ on-chain herself (and then downloading the rest of the payload which accompanies the tag), or by making a privacy-preserving request to a server, to retrieve the payload which accompanies the tag. Assuming the payload has been identified as Alice's, and retrieved by Alice, we proceed.

Given that the tag in this illustration was derived from Alice's master key, the tag itself doesn't

convey which app_address to use, to derive the correct app-siloed incoming viewing secret key that would enable decryption of the ciphertext. So first Alice needs to decrypt the **ciphertext_header** using her master key:

Thing	Derivation	Name
$h_{m,enc,header}$	$h("?", ovs k_m, Epk_{d,header})$	
app_address	$decrypt_{h_{m,enc,header}}(\text{ciphertext_header})$	
$ovsk_{app}$	See derivations above. Use the decrypted app_address.	
$h_{app,enc}$	$h("?", ovs k_m, Epk_d)$	
plaintext	$decrypt_{h_{app,enc}}(\text{ciphertext})$	

Doing this inside an app circuit

Here's how an app circuit could constrain the app-siloed outgoing viewing secret key ($ovsk_{app}$) to be correct:

The app circuit exposes, as public inputs, an "outgoing viewing key validation request":

Thing	Derivation	Name	Comments
outgoing_viewing_key_validation_request	app_address: app_address, hardened_child_sk: nsk_{app} , claimed_parent_pk: Npk_m		

The kernel circuit can then validate the request (having been given $ovsk_m$ as a private input to the kernel circuit):

Thing	Derivation	Name	Comments
$\textcolor{red}{ovsk}_{\text{app}}$	$h(\textcolor{red}{ovsk}_m, \text{app_address})$		
$Ovpk_m$	$\textcolor{red}{ovsk}_m \cdot G$	Outgoing viewing public key	
			Copy-constrain $\textcolor{red}{ovsk}_m$ with $\textcolor{red}{ovsk}_m$.

If the kernel circuit succeeds in these calculations, then the $\textcolor{red}{ovsk}_{\text{app}}$ has been validated as the correct app-siled secret key for $Ovpk_m$.

Encrypt and tag an internal incoming message

Internal incoming messages are handled analogously to outgoing messages, since in both cases the sender is the same as the recipient, who has access to the secret keys when encrypting and tagging the message.

Precompiles

Precompiled contracts, which borrow their name from Ethereum's, are contracts not deployed by users but defined at the protocol level. These contract [instances](#) and their [classes](#) are assigned well-known low-number addresses and identifiers, and their implementation is subject to change via protocol upgrades. Precompiled contracts in Aztec are implemented as a set of circuits, one for each function they expose, like user-defined private contracts. Precompiles may make use of the local PXE oracle.

Note that, unlike user-defined contracts, the address of a precompiled [contract instance](#) and the [identifier of its class](#) both have no known preimage.

The rationale for precompiled contracts is to provide a set of vetted primitives for [note encryption](#) and [tagging](#) that applications can use safely. These primitives are guaranteed to be always-satisfiable when called with valid arguments. This allows account contracts to choose their preferred method of encryption and tagging from any primitive in this set, and application contracts to call into them without the risk of calling into a untrusted code, which could potentially halt the execution flow via an unsatisfiable constraint. Furthermore, by exposing these primitives in a reserved set of well-known addresses, applications can be forward-compatible and incorporate new encryption and tagging methods as accounts opt into them.

Constants

- `ENCRYPTION_BATCH_SIZES=[4, 8, 16, 32]`: Defines what max [batch sizes](#) are supported in precompiled encryption methods.
- `ENCRYPTION_PRECOMPILE_ADDRESS_RANGE=0x00..0xFFFF`: Defines the range of addresses reserved for precompiles used for encryption and tagging.

- `MAX_PLAINTEXT_LENGTH`: Defines the maximum length of a plaintext to encrypt.
- `MAX_CIPHERTEXT_LENGTH`: Defines the maximum length of a returned encrypted ciphertext.
- `MAX_TAGGED_CIPHERTEXT_LENGTH`: Defines the maximum length of a returned encrypted ciphertext prefixed with a note tag.

Encryption and tagging precompiles

All precompiles in the address range `ENCRYPTION_PRECOMPILE_ADDRESS_RANGE` are reserved for encryption and tagging. Application contracts are expected to call into these contracts with note plaintext(s), recipient address(es), and public key(s). To facilitate forward compatibility, all unassigned addresses within the range expose the functions below as no-ops, meaning that no actions will be executed when calling into them.

All functions in these precompiles accept a `PublicKeys` struct which contains the user-advertised public keys. The structure of each of the public keys included can change from one encryption method to another, with the exception of the `nullifier_key` which is always restricted to a single field element. For forward compatibility, the precompiles interface accepts a hash of the public keys, which can be expanded within each method via an oracle call.

```
struct PublicKeys:
    nullifier_key: Field
    incoming_encryption_key: PublicKey
    outgoing_encryption_key: PublicKey
    incoming_internal_encryption_key: PublicKey
    tagging_key: PublicKey
```

To identify which public key to use in the encryption, precompiles also accept an

enum:

```
enum EncryptionType:  
    incoming = 1  
    outgoing = 2  
    incoming_internal = 3
```

Precompiles expose the following private functions:

```
validate_keys(public_keys_hash: Field): bool
```

Returns true if the set of public keys represented by `public_keys` is valid for this encryption and tagging mechanism. The precompile must guarantee that any of its methods must succeed if called with a set of public keys deemed as valid. This method returns `false` for undefined precompiles.

```
encrypt(public_keys_hash: Field, encryption_type: EncryptionType,  
recipient: AztecAddress, plaintext: Field[MAX_PLAINTEXT_LENGTH]):  
Field[MAX_CIPHERTEXT_LENGTH]
```

Encrypts the given plaintext using the provided public keys, and returns the encrypted ciphertext.

```
encrypt_and_tag(public_keys_hash: Field, encryption_type:  
EncryptionType, recipient: AztecAddress, plaintext:  
Field[MAX_PLAINTEXT_LENGTH]): Field[MAX_TAGGED_CIPHERTEXT_LENGTH]
```

Encrypts and tags the given plaintext using the provided public keys, and returns the encrypted note prefixed with its tag for note discovery.

```
encrypt_and_broadcast(public_keys_hash: Field, encryption_type:  
EncryptionType, recipient: AztecAddress, plaintext:  
Field[MAX_PLAINTEXT_LENGTH]): Field[MAX_TAGGED_CIPHERTEXT_LENGTH]
```

Encrypts and tags the given plaintext using the provided public keys, broadcasts them as an event, and returns the encrypted note prefixed with its tag for note discovery.

```
encrypt<N>([call_context: CallContext, public_keys_hash: Field,  
encryption_type: EncryptionType, recipient: AztecAddress,  
plaintext: Field[MAX_PLAINTEXT_LENGTH] ][N]):  
Field[MAX_CIPHERTEXT_LENGTH][N]  
encrypt_and_tag<N>([call_context: CallContext, public_keys_hash:  
Field, encryption_type: EncryptionType, recipient: AztecAddress,  
plaintext: Field[MAX_PLAINTEXT_LENGTH] ][N]):  
Field[MAX_TAGGED_CIPHERTEXT_LENGTH][N]  
encrypt_and_broadcast<N>([call_context: CallContext,  
public_keys_hash: Field, encryption_type: EncryptionType,  
recipient: AztecAddress, plaintext: Field[MAX_PLAINTEXT_LENGTH]  
][N]): Field[MAX_TAGGED_CIPHERTEXT_LENGTH][N]
```

Batched versions of the methods above, which accept an array of `N` tuples of public keys, recipient, and plaintext to encrypt in batch. Precompiles expose instances of this method for multiple values of `N` as defined by `ENCRYPTION_BATCH_SIZES`. Values in the batch with zeroes are skipped. These functions are intended to be used in [batched calls](#).

```
decrypt(public_keys_hash: Field, encryption_type: EncryptionType,  
owner: AztecAddress, ciphertext: Field[MAX_CIPHERTEXT_LENGTH]):  
Field[MAX_PLAINTEXT_LENGTH]
```

Decrypts the given ciphertext, encrypted for the provided owner. Instead of receiving

the decryption key, this method triggers an oracle call to fetch the private decryption key directly from the local PXE and validates it against the supplied public key, in order to avoid leaking a user secret to untrusted application code. This method is intended for provable decryption use cases.

Encryption strategies

List of encryption strategies implemented by precompiles:

AES128

Uses AES128 for encryption, by generating an AES128 symmetric key and an IV from a shared secret derived from the recipient's public key and an ephemeral keypair. Requires that the recipient's keys are points in the Grumpkin curve. The output of the encryption is the concatenation of the encrypted ciphertext and the ephemeral public key.

Pseudocode for the encryption process:

```
encrypt(plaintext, recipient_public_key):
    ephemeral_private_key, ephemeral_public_key =
    grumpkin_random_keypair()
    shared_secret = recipient_public_key * ephemeral_private_key
    [aes_key, aes_iv] = sha256(shared_secret ++ [0x01])
    return ephemeral_public_key ++ aes_encrypt(aes_key, aes_iv,
    plaintext)
```

Pseudocode for the decryption process:

```
decrypt(ciphertext, recipient_private_key):
    ephemeral_public_key = ciphertext[0:64]
```

Note tagging strategies

List of note tagging strategies implemented by precompiles:

Trial decryption

Trial decryption relies on the recipient to brute-force trial-decrypting every note emitted by the chain. Every note is attempted to be decrypted with the associated decryption scheme. If decryption is successful, then the note is added to the local database. This requires no note tags to be emitted along with a note.

In AES encryption, the plaintext is prefixed with the first 8 bytes of the IV. Decryption is deemed successful if the first 8 bytes of the decrypted plaintext matches the first 8 bytes of the IV derived from the shared secret.

This is the cheapest approach in terms of calldata cost, and the simplest to implement, but puts a significant burden on the user. Should not be used except for accounts tied to users running full nodes.

Delegated trial decryption

Delegated trial decryption relies on a tag added to each note, generated using the recipient's tagging public key. The holder of the corresponding tagging private key can trial-decrypt each tag, and if decryption is successful, proceed to decrypt the contents of the note using the associated decryption scheme.

This allows a user to share their tagging private key with a trusted service provider, who then proceeds to trial decrypt all possible note tags on their behalf. This scheme is simple for the user, but requires trust on a third party.

Tag hopping

Tag hopping relies on establishing a one-time shared secret through a handshake between each sender-recipient pair, advertise the handshake through a trial-decrypted brute-forced channel, and then generate tags by combining the shared secret and an incremental counter. Recipients need to trial-decrypt events emitted by a canonical `Handshake` contract to detect new channels established with them, and then scan for the next tag for each open channel. Note that the handshake contract leaks whenever a new shared secret has been established, but the participants of the handshake are kept hidden.

This method requires the recipient to be continuously trial-decrypting the handshake channel, and then scanning for a number of tags equivalent to the number of handshakes they had received. While this can get to too large amounts for particularly active addresses, it is still far more efficient than trial decryption.

When Alice wants to send a message to Bob for the first time:

1. Alice creates a note, and calls into Bob's encryption and tagging precompile.
2. The precompile makes an oracle call to `getSharedSecret(Alice, Bob)`.
3. Alice's PXE looks up the shared secret which doesn't exist since this is their first interaction.
4. Alice's PXE generates a random shared secret, and stores it associated Bob along with `counter=1`.
5. The precompile makes a call to the `Handshake` contract that emits the shared secret, encrypted for Bob and optionally Alice.
6. The precompile computes `new_tag = hash(alice, bob, secret, counter)`, emits it as a nullifier, and prepends it to the note ciphertext before broadcasting it.

For all subsequent messages:

1. Alice creates a note, and calls into Bob's encryption and tagging precompile.
2. The precompile makes an oracle call to `getSharedSecret(Alice, Bob)`.
3. Alice's PXE looks up the shared secret and returns it, along with the current value for `counter`, and locally increments `counter`.
4. The precompile computes `previous_tag = hash(alice, bob, secret, counter)`, and performs a merkle membership proof for it in the nullifier tree. This ensures that tags are incremental and cannot be skipped.
5. The precompile computes `new_tag = hash(alice, bob, secret, counter + 1)`, emits it as a nullifier, and prepends it to the note ciphertext before broadcasting it.

Defined precompiles

List of precompiles defined by the protocol:

Address	Encryption	Note Tagging	Comments
0x01	Noop	Noop	Used by accounts to explicitly signal that they cannot receive encrypted payloads. Validation method returns <code>true</code> only for an empty list of public keys. All other methods return empty.
0x02	AES128	Trial decryption	
0x03	AES128	Delegated	

Address	Encryption	Note Tagging	Comments
		trial decryption	
0x04	AES128	Tag hopping	

Diversified and Stealth Accounts

The [keys specification](#) describes derivation mechanisms for diversified and stealth public keys. However, the protocol requires users to interact with addresses.

Computing Addresses

To support diversified and stealth accounts, a user may compute the deterministic address for a given account contract that is deployed using a diversified or stealth public key, so a sender can interact with the resulting so-called "diversified" or "stealth" address even before the account contract is deployed.

When the user wants to access the notes that were sent to the diversified or stealth address, they can deploy the contract at their address, and control it privately from their main account.

Account Contract Pseudocode

As an example implementation, account contracts for diversified and stealth accounts can be designed to require no private constructor or state, and delegate entrypoint access control to their master address.

```
contract DiversifiedAccount

    private fn entrypoint(payload: action[])
        assert msg_sender == get_owner_address()
        execute(payload)
```

Given the contract does not require initialization since it has no constructor, it can be used by its owner without being actually deployed, which reduces the setup cost.

Discarded Approaches

An alternative approach was to introduce a new type of call, a diversified call, that would allow the caller to impersonate any address they can derive from their own, for an enshrined derivation mechanism. Account contracts could use this opcode, as opposed to a regular call, to issue calls on behalf on their diversified and stealth addresses. However, this approach failed to account for calls made back to the account contracts, in particular authwit checks. It also required protocol changes, introducing a new type of call which could be difficult to reason about, and increased attack surface. The only benefit over the approach chosen is that it would require one less extra function call to hop from the user's main account contract to the diversified or stealth one.

State

The global state is the set of data that makes up Aztec - it is persistent and only updates when new blocks are added to the chain.

The state consists of multiple different categories of data with varying requirements. What all of the categories have in common is that they need strong integrity guarantees and efficient membership proofs. Like most other blockchains, this can be enforced by structuring the data as leaves in Merkle trees.

However, unlike most other blockchains, our contract state cannot use a Merkle tree as a key-value store for each contract's data. The reason for this is that we have both private and public state; while public state could be stored in a key-value tree, private state cannot, as doing so would leak information whenever the private state is updated, even if encrypted.

To work around this, we use a two-tree approach for state that can be used privately. Namely we have one (or more) tree(s) where data is added to (sometimes called a data tree), and a second tree where we "nullify" or mark the data as deleted. This allows us to "update" a leaf by adding a new leaf to the data tree, and add the nullifier of the old leaf to the second tree (the nullifier tree). That way we can show that the new leaf is the "active" one, and that the old leaf is "deleted".

When dealing with private data, only the hash of the data is stored in the leaf in our data tree and we must set up a derivation mechanism that ensures nullifiers can be computed deterministically from the pre-image (the data that was hashed). This way, no-one can tell what data is stored in the leaf (unless they already know it), and therefore won't be able to derive the nullifier and tell if the leaf is active or deleted.

Convincing someone that a piece of data is active can then be done by proving its membership in the data tree, and that it is not deleted by proving its non-membership in the nullifier tree. This ability to efficiently prove non-membership is one of the extra requirements we have for some parts of our state. To support the requirements most efficiently, we use two families of Merkle trees:

- The [Append-only Merkle tree](#), which supports efficient membership proofs,
- The [Indexed Merkle tree](#), which supports efficient membership and non-membership proofs but increases the cost of adding leaves.

Private State Access

Whenever a user is to read or use data, they must then convince the "rollup" that their data is active. As mentioned above, they must prove that the data is in the data tree (membership proof) and that it is still active (non-membership proof). However, there are nuances to this approach!

One important aspect to consider is *when* state can be accessed. In most blockchains, state is always accessed at the head of the chain and changes are only made by the sequencer as new blocks are added.

However, since private execution relies on proofs generated by the user, this would be very impractical - one user's transaction could invalidate everyone else's.

While proving inclusion in the data tree can be done using historical state, the non-membership proof in the nullifier tree cannot.

Membership can be proven using historical state because we are using an append-only tree, so anything that was there in the past must still be in the append-only tree now.

However, this doesn't work for the non-membership proof, as it can only prove that the data was active at the time the proof was generated, not that it is still active today! This would allow a user to create multiple transactions spending the same data and then send those transactions all at once, creating a double spend.

To solve this, we need to perform the non-membership proofs at the head of the chain, which only the sequencer knows! This means that instead of the user proving that the nullifier of the data is not in the nullifier tree, they provide the nullifier as part of their transaction, and the sequencer then proves non-membership **AND** inserts it into the nullifier tree. This way, if multiple transactions include the same nullifier, only one of them will be included in the block as the others will fail the non-membership proof.

Why does it need to insert the nullifier if I'm reading? Why can't it just prove that the nullifier is not in the tree? Well, this is a privacy concern. If you just make the non-membership proof, you are leaking that you are reading data for nullifier x , so if you read that data again at a later point, x is seen by the sequencer again and it can infer that it is the same actor reading data. By emitting the nullifier the read is indistinguishable from a write, and the sequencer cannot tell what is happening and there will be no repetitions.

This however also means, that whenever data is only to be read, a new note with the same data must be inserted into the data tree. This note have new randomness, so anyone watching will be unable to tell if it is the same data inserted, or if it is new data. This is good for privacy, but comes at an additional cost.

A side-effect of this also means that if multiple users are "sharing" their notes, any one of them reading the data will cause the note to be updated, so pending transaction that require the note will fail.

State Categories

Below is a short description of the state catagories (trees) and why they have the type they have.

- **Note Hashes:** A set of hashes (commitments) of the individual blobs of contract data (we call these blobs of data notes). New notes can be created and their hashes inserted through contract execution. We need to support efficient membership proofs as any read will require one to prove validity. The set is represented as an [Append-only Merkle tree](#), storing the note hashes as leaves.
- **Nullifiers:** A set of nullifiers for notes that have been spent. We need to support efficient non-membership proofs since we need to check that a note has not been spent before it can be used. The set is represented as an [Indexed Merkle tree](#).
- **Public Data:** The key-value store for public contract state. We need to support both efficient membership and non-membership proofs! We require both, since the tree is "empty" from the start. Meaning that if the key is not already stored (non-membership), we need to insert it, and if it is already stored (membership) we need to just update the value.
- **L1 to L2 Messages:** The set of messages sent from L1 to L2. The set itself only needs to support efficient membership proofs, so we can ensure that the message was correctly sent from L1. However, it utilizes the Nullifier tree from above to ensure that the message cannot be processed twice. The set is represented as an [Append-only Merkle tree](#). For more information on how the L1 to L2 messages are used, see the [L1 Smart Contracts](#) page.
- **Archive Tree:** The set of block headers that have been processed. We need to support efficient membership proofs as this is used in private execution to get the roots of the other trees. The set is represented as an [Append-only Merkle tree](#).

To recall, the global state in Aztec is represented by a set of Merkle trees: the [Note Hash tree](#), [Nullifier tree](#), and [Public Data tree](#) reflect the latest state of the chain, while the L1 to L2 message tree allows for [cross-chain communication](#) and the [Archive Tree](#) allows for historical state access.

Tree implementations

Aztec relies on two Merkle tree implementations in the protocol: append-only and indexed Merkle trees.

Archive Tree

The Archive Tree is an append-only Merkle tree that stores the Headers (see the diagram below) of all previous blocks in the chain as its leaves.

Note Hash Tree

The Note Hash tree is an append-only Merkle tree that stores siloed note hashes as its elements. Each element in the tree is a 254-bit altBN-254 scalar field element. This tree...

Nullifier Tree

The Nullifier tree is an indexed Merkle tree that stores nullifier values. Each value stored in the tree is a 254-bit altBN-254 scalar field element. This tree is part of the global sta...

Public Data Tree

The Public Data tree is an indexed Merkle tree that stores public-state. Each item stored in the tree is a key-value pair, where both key and value are 254-bit altBN-254 scalar ...

Wonky Tree

A 'wonky' tree is an append-only unbalanced merkle tree, filled from left to right. It is used to construct rollup proofs without padding empty transactions.

Tree implementations

Aztec relies on two Merkle tree implementations in the protocol: append-only and indexed Merkle trees.

Append-only Merkle trees

In an append-only Merkle tree, new leaves are inserted in order from left to right. Existing leaf values are immutable and cannot be modified. These trees are useful to represent historical data, as historical data is not altered, and new entries can be added as new transactions and blocks are processed.

Append-only trees allow for more efficient syncing than sparse trees, since clients can sync from left to right starting with their last known value. Updates to the tree root, when inserting new leaves, can be computed from the rightmost "frontier" of the tree (i.e., from the sibling path of the rightmost nonzero leaf). Batch insertions can be computed with fewer hashes than in a sparse tree. The historical snapshots of append-only trees also enable efficient membership proofs; as older roots can be computed by completing the merkle path from a past left subtree with an empty right subtree.

Wonky Merkle Trees

We also use a special type of append-only tree to structure the rollup circuits. Given n leaves, we fill from left to right and attempt to pair them to produce the next layer. If n is a power of 2, this tree looks exactly like a standard append-only merkle tree. Otherwise, once we reach an odd-sized row we shift the final node up until we reach another odd row to combine them.

This results in an unbalanced tree where there are no empty leaves. For rollups, this

means we don't have to pad empty transactions and process them through the rollup circuits. A full explanation is given [here](#).

Indexed Merkle trees

Indexed Merkle trees, introduced [here](#), allow for proofs of non-inclusion more efficiently than sparse Merkle trees. Each leaf in the tree is a tuple of: the leaf value, the next-highest value in the tree, and the index of the leaf where that next-highest value is stored. New leaves are inserted from left to right, as in the append-only tree, but existing leaves can be *modified* to update the next-highest value and next-highest index (a.k.a. the "pointer") if a new leaf with a "closer value" is added to the tree. An Indexed Merkle trees behaves as a Merkle tree over a sorted linked list.

With an Indexed Merkle tree, proving non-membership of a value x then requires a membership proof of the node with value lower than x and a next-highest value greater than x . The cost of this proof is proportional to the height of the tree, which can be set according to the expected number of elements to be stored in the tree. For comparison, a non-membership proof in a sparse tree requires a tree with height proportional to the size of the elements, so when working with 256-bit elements, 256 hashes are required for a proof.

Refer to [this page](#) for more details on how insertions, updates, and membership proofs are executed on an Indexed Merkle tree.

Siloing leaves

In several trees in the protocol we indicate that its leaves are "siloed". This refers to hashing the leaf value with some other "siloing" value before inserting it into the tree. The siloing value is typically the contract address of the contract that produced the value. This allows us to store disjoint "domains" within the same tree, ensuring that a

value emitted from one domain cannot affect others.

To guarantee the siloing of leaf values, siloing is performed by a trusted protocol circuit, such as a kernel or rollup circuit, and not by an application circuit.

Archive Tree

The Archive Tree is an [append-only Merkle tree](#) that stores the Headers (see the diagram below) of all previous blocks in the chain as its leaves.

For most chains this is not required since they are always executing at the head of the chain. However, private execution relies on proofs generated by the user, and since users don't know the current head they must base their proofs on historical state. By including all prior headers (which include commitments to the state) the Archive Tree allows us to easily prove that the historic state that a transaction was proven upon is valid.

Furthermore, since each Header includes a snapshot of the Archive Tree as at the time of insertion, as well as commitments to the Header's block content and global variables, we can use the Archive Tree to prove statements about the state at any given block or even transactions that occurred at specific blocks.

Note Hash Tree

The Note Hash tree is an [append-only Merkle tree](#) that stores [siloed](#) note hashes as its elements. Each element in the tree is a 254-bit altBN-254 scalar field element. This tree is part of the global state, and is used to prove existence of private notes via Merkle membership proofs.

Note commitments are immutable once created. Still, notes can be consumed ("read") by functions. To preserve privacy, a consumed note is not removed from the tree, otherwise it would be possible to link the transaction that created a note with the one that consumed it. Instead, a note is consumed by emitting a deterministic [nullifier](#).

Contracts emit new note commitments via the `new_note_hashes` in the `CircuitPublicInputs`, which are subsequently [siloed](#) by contract address by the Kernel circuit. Siloing the commitment ensures that a malicious contract cannot create notes for (that is, modify the state of) another contract.

The Kernel circuit also guarantees uniqueness of commitments by hashing them with a nonce, derived from the transaction identifier and the index of the commitment within the transaction's array of newly-created note hashes. Uniqueness means that a note with the same contents can be emitted more than once, and each instance can be independently nullified. Without uniqueness, two notes with the same content would yield the same commitment and nullifier, so nullifying one of them would render the second one as nullified as well.

The pseudocode for siloing and making a commitment unique is the following, where each `hash` operation is a Pedersen hash with a unique generator index, indicated by the constant in all caps.

```
fn compute_siloed_note_hash(commitment, contract, transaction):
    let index = index_of(commitment, transaction.commitments)
    let nonce = hash([transaction.tx_hash, index], NOTE_HASH_NONCE)
    let unique_note_hash = hash([nonce, commitment],
UNIQUE_NOTE_HASH);
    return hash([contract, unique_note_hash], SILOED_NOTE_HASH)
```

The unique siloed commitment of a note is included in the `transaction` `data`, and then inserted into the Note Hash tree by the sequencer as the transaction is included in a block.

The protocol does not enforce any constraints on any note hashes emitted by an application. This means that applications are responsible for including a `randomness` field in the note hash to make the commitment *hiding* in addition to *binding*. If an application does not include randomness, and the note preimage can be guessed by an attacker, it makes the note vulnerable to preimage attacks, since the siloing and uniqueness steps do not provide hiding.

Furthermore, since there are no constraints to the commitment emitted by an application, an application can emit any value whatsoever as a `new_note_hash`, including values that do not map to a note hash.

Nullifier Tree

The Nullifier tree is an [indexed Merkle tree](#) that stores nullifier values. Each value stored in the tree is a 254-bit altBN-254 scalar field element. This tree is part of the global state, and is primarily used to prove non-existence of a nullifier when a note is consumed (as a way of preventing double-spend).

In addition to storing nullifiers of notes, the nullifier tree is more generally useful to prevent any action from being repeated twice. This includes preventing re-initialization of state variables, and [re-deployment of contracts](#).

Nullifiers are asserted to be unique during insertion, by checking that the inserted value is not equal to the value and next-value stored in the prior leaf in the indexed tree. Any attempt to insert a duplicated value is rejected.

Contracts emit new nullifiers via the `new_nullifiers` field of the `CircuitPublicInputs` ABI. Similarly to elements in the [Note Hash tree](#), nullifiers are [siloed](#) by contract address, by the Kernel circuit, before being inserted into the tree. This ensures that a contract cannot emit the nullifiers of other contracts' state variables!

```
fn compute_siloed_nullifier(nullifier, contract):  
    return hash([contract, nullifier], OUTER_NULLIFIER)
```

Nullifiers are primarily used for privately marking notes as consumed. When a note is consumed in an application, the application computes and emits a deterministic nullifier associated to the note. If a user attempts to consume the same note more than once, the same nullifier will be generated, and will be rejected on insertion by the nullifier tree.

Nullifiers provide privacy by being computed using a deterministic secret value, such

as the owner siloed nullifier secret key, or a random value stored in an encrypted note. This ensures that, without knowledge of the secret value, it is not possible to calculate the associated nullifier, and thus it is not possible to link a nullifier to its associated note commitment.

Applications are not constrained by the protocol on how the nullifier for a note is computed. It is responsibility of the application to guarantee determinism in calculating a nullifier, otherwise the same note could be spent multiple times.

Furthermore, nullifiers can be emitted by an application just to ensure that an action can be executed only once, such as initializing a value, and are not required to be linked to a note commitment.

Public Data Tree

The Public Data tree is an [indexed Merkle tree](#) that stores public-state. Each item stored in the tree is a key-value pair, where both key and value are 254-bit altBN-254 scalar field elements. Items are sorted based on their key, so each indexed tree leaf contains a tuple with the key, the value, the next-highest key, and the index in the tree for the next-highest key. This tree is part of the global state, and is updated by the sequencer during the execution of public functions.

An indexed Merkle tree is used instead of a sparse Merkle tree in order to reduce the tree height. A lower height means shorter membership proofs.

Keys in the Public Data tree are [siloed](#) using the contract address, to prevent a contract from overwriting the public state of another contract.

```
fn compute_siloed_public_data_item(key, value, contract):  
    let siloed_key = hash([contract, key], PUBLIC_DATA_LEAF)  
    return [siloed_key, value]
```

When attempting to read a key from the Public Data tree, the key may or may not be present. If the key is not present, then a non-membership proof can be produced. When a key is written to, either a new node is appended to the tree if the key was not present, or its value is overwritten if it was.

Public functions can read from or write to the Public Data tree by emitting `contract_storage_read` and `contract_storage_update_requests` in the `PublicCircuitPublicInputs`. The Kernel circuit then siloes these requests per contract.

Contracts can store arbitrary data at a given key, which is always stored as a single field element. Applications are responsible for interpreting this data. Should an application need to store data larger than a single field element, they are responsible

for partitioning it across multiple keys.

Wonky Tree

A 'wonky' tree is an append-only unbalanced merkle tree, filled from left to right. It is used to construct [rollup](#) proofs without padding empty transactions.

For example, using a balanced merkle tree to rollup 5 transactions requires padding of 3 empty transactions:

Where each node marked with `*` indicates a circuit proving entirely empty information. While the above structure does allow us to easily construct balanced trees later on consisting of `out_hashes` and `tx_effects_hashes`, it will lead to wasted compute and higher block processing costs unless we provide a number of transactions equal to a power of 2.

Our wonky tree implementation instead gives the below structure for 5 transactions:

Here, each circuit is proving useful transaction information with no wasted compute. We can construct a tree like this one for any number of transactions by greedy filling from left to right. Given the required 5 base circuits:

...we then pair these base circuits up to form merges:

Since we have an odd number of transactions, we cannot pair up the final base. Instead, we continue to pair the next layers until we reach a layer with an odd number of members. In this example, that's when we reach merge 2:

Once paired, the base layer has length 4, the next merge layer has 2, and the final merge layer has 1. After reaching a layer with odd length, the orchestrator can now pair base 4:

Since we have processed all base circuits, this final pair will be input to a root circuit.

Filling from left to right means that we can easily reconstruct the tree only from the number of transactions n . The above method ensures that the final tree is a combination of *balanced* subtrees of descending size. The widths of these subtrees are given by the decomposition of n into powers of 2. For example, 5 transactions:

```
Subtrees: [4, 1] ->
    left_subtree_root = balanced_tree(txs[0..4])
    right_subtree_root = balanced_tree(txs[4]) = txs[4]
    root = left_subtree_root | right_subtree_root
```

For 31 transactions:

```
Subtrees: [16, 8, 4, 2, 1] ->
    Merge D: left_subtree_root = balanced_tree(txs[0..16])
    right_subtree_root = Subtrees: [8, 4, 2, 1] --> {
        Merge C: left_subtree_root = balanced_tree(txs[16..24])
        right_subtree_root = Subtrees: [4, 2, 1] --> {
            Merge B: left_subtree_root = balanced_tree(txs[24..28])
            right_subtree_root = Subtrees: [2, 1] --> {
                Merge A: left_subtree_root =
                balanced_tree(txs[28..30])
                right_subtree_root = balanced_tree(txs[30]) =
                txs[30]
                Merge 0: root = left_subtree_root |
                right_subtree_root
            }
            Merge 1: root = left_subtree_root | right_subtree_root
        }
        Merge 2: root = left_subtree_root | right_subtree_root
    }
    root = left_subtree_root | right_subtree_root
```

An unrolled recursive algorithm is not the easiest thing to read. This diagram represents the 31 transactions rolled up in our wonky structure, where each **Merge**

<num> is a 'subroot' above:

The tree is reconstructed to check the txs_effects_hash (= the root of a wonky tree given by leaves of each tx's tx_effects) on L1. We also reconstruct it to provide a membership path against the stored out_hash (= the root of a wonky tree given by leaves of each tx's L2 to L1 message tree root) for consuming a L2 to L1 message.

Currently, this tree is built via the orchestrator given the number of transactions to rollup. Each 'node' is assigned a level (0 at the root) and index in that level. The below function finds the parent level:

```
// Calculates the index and level of the parent rollup circuit
public findMergeLevel(currentLevel: bigint, currentIndex: bigint) {
    const moveUpMergeLevel = (levelSize: number, index: bigint,
nodeToShift: boolean) => {
        levelSize /= 2;
        if (levelSize & 1) {
            [levelSize, nodeToShift] = nodeToShift ? [levelSize + 1,
false] : [levelSize - 1, true];
        }
        index >>= 1n;
        return { thisLevelSize: levelSize, thisIndex: index,
shiftUp: nodeToShift };
    };
    let [thisLevelSize, shiftUp] = this.totalNumTxs & 1 ?
[this.totalNumTxs - 1, true] : [this.totalNumTxs, false];
    const maxLevel = this.numMergeLevels + 1n;
    let placeholder = currentIndex;
    for (let i = 0; i < maxLevel - currentLevel; i++) {
        ({ thisLevelSize, thisIndex: placeholder, shiftUp } =
moveUpMergeLevel(thisLevelSize, placeholder, shiftUp));
    }
    let thisIndex = currentIndex;
    let mergeLevel = currentLevel;
```

For example, `Base 4` above starts with `level = 3` and `index = 4`. Since we have an odd number of transactions at this level, `thisLevelSize` is set to 4 with `shiftUp = true`.

The while loop triggers and shifts up our node to `level = 2` and `index = 2`. This level (containing `Merge 0` and `Merge 1`) is of even length, so the loop continues. The next iteration shifts up to `level = 1` and `index = 1` - we now have an odd level, so the loop stops. The actual position of `Base 4` is therefore at `level = 1` and `index = 1`. This function returns the parent level of the input node, so we return `level = 0, index = 0`, correctly indicating that the parent of `Base 4` is the root.

Flexible wonky trees

We can also encode the structure of *any* binary merkle tree by tracking `number_of_branches` and `number_of_leaves` for each node in the tree. This encoding was originally designed for `logs` before they were included in the `txs_effects_hash`, so the below explanation references the leaves stored in relation to logs and transactions.

The benefit of this method as opposed to the one above is allowing for any binary structure and therefore allowing for 'skipping' leaves with no information. However, the encoding grows as the tree grows, by at least 2 bytes per node. The above implementation only requires the number of leaves to be encoded, which will likely only require a single field to store.

Encoding

1. The encoded logs data of a transaction is a flattened array of all logs data within the transaction:

```
tx_logs_data = [number_of_logs, ...log_data_0, ...log_data_1,  
...]
```

- The encoded logs data of a block is a flatten array of a collection of the above *tx_logs_data*, with hints facilitating hashing replay in a binary tree structure:

```
block_logs_data = [number_of_branches, number_of_transactions,
...tx_logs_data_0, ...tx_logs_data_1, ...]
```

- number_of_transactions* is the number of leaves in the left-most branch, restricted to either 1 or 2.
- number_of_branches* is the depth of the parent node of the left-most leaf.

Here is a step-by-step example to construct the *block_logs_data*:

- A rollup, *R01*, merges two transactions: *tx0* containing *tx_logs_data_0*, and *tx1* containing *tx_logs_data_1*:

```
block_logs_data: [0, 2, ...tx_logs_data_0, ...tx_logs_data_1]
```

Where 0 is the depth of the node *R01*, and 2 is the number of aggregated *tx_logs_data* of *R01*.

- Another rollup, *R23*, merges two transactions: *tx3* containing *tx_logs_data_3*, and *tx2* without any logs:

```
block_logs_data: [0, 1, ...tx_logs_data_3]
```

Here, the number of aggregated *tx_logs_data* is 1.

- A rollup, *RA*, merges the two rollups *R01* and *R23*:

```
block_logs_data: [1, 2, ...tx_logs_data_0, ...tx_logs_data_1, 0, 1,
...tx_logs_data_3]
```

The result is the *block_logs_data* of *R01* concatenated with the *block_logs_data*

of $R23$, with the *number_of_branches* of $R01$ incremented by 1. The updated value of *number_of_branches* ($0 + 1$) is also the depth of the node $R01$.

4. A rollup, RB , merges the above rollup RA and another rollup $R45$:

```
block_logs_data: [2, 2, ...tx_logs_data_0, ...tx_logs_data_1, 0, 1,
...tx_logs_data_3, 0, 2, ...tx_logs_data_4, ...tx_logs_data_5]
```

The result is the concatenation of the *block_logs_data* from both rollups, with the *number_of_branches* of the left-side rollup, RA , incremented by 1.

Verification

Upon receiving a proof and its encoded logs data, the entity can ensure the correctness of the provided *block_logs_data* by verifying that the *accumulated_logs_hash* in the proof can be derived from it:

```
const accumulated_logs_hash =
compute_accumulated_logs_hash(block_logs_data);
assert(accumulated_logs_hash == proof.accumulated_logs_hash);
assert(block_logs_data.accumulated_logs_length ==
proof.accumulated_logs_length);

function compute_accumulated_logs_hash(logs_data) {
    const number_of_branches = logs_data.read_u32();

    const number_of_transactions = logs_data.read_u32();
    let res = hash_tx_logs_data(logs_data);
    if number_of_transactions == 2 {
        res = hash(res, hash_tx_logs_data(logs_data));
    }

    for (let i = 0; i < number_of_branches; ++i) {
        const res_right = compute_accumulated_logs_hash(logs_data);
        res = hash(res, res_right);
    }
}
```


Transactions

A transaction is the minimal action that changes the state of the network. Transactions in Aztec have a private and a public component, where the former is executed in the user's private execution environment (PXE) and the latter by the sequencer.

A transaction is also split into three phases to support authorization abstraction and fee payments: a validation and fee preparation phase, a main execution phase, and fee distribution phase.

Users initiate a transaction by sending a `transaction_request` to their local PXE, which locally simulates and proves the transaction and returns a `transaction_object` identified by a `transaction_hash`. This transaction object is then broadcast to the network via an Aztec Node, which checks its validity, and is eventually picked up by a sequencer who executes the public component of the transaction and includes it in a block.

Local Execution

Transactions are initiated via a transaction execution request sent from the user to their local private execution environment (PXE). The PXE first executes the transaction local...

Public execution

Transactions have a public execution component. Once a transaction is picked up by a sequencer to be included in a block, the sequencer is responsible for executing all en...

Transaction object

The transaction object is the struct broadcasted to the p2p network, generated by local execution by the user's PXE. Sequencers pick up transactions from the p2p network t...

Validity conditions

The validity conditions of a transaction define when a transaction object is valid. Nodes should check the validity of a transaction when they receive it either directly or throu...

Local Execution

Transactions are initiated via a *transaction execution request* sent from the user to their local *private execution environment* (PXE). The PXE first executes the transaction locally in a *simulation* step, and then generates a *zero-knowledge proof* of correct execution. The PXE is then responsible for converting a *transaction execution request* into a *transaction* ready to be broadcasted to the network.

Execution request

A transaction execution request has the following structure. Note that, since Aztec uses full native account abstraction where every account is backed by a contract, a transaction execution request only needs to provide the contract address, function, and arguments of the initial call; nonces and signatures are arguments to the call, and thus opaque to the protocol.

Field	Type	Description
origin	AztecAddress	Address of the contract where the transaction is initiated.
functionSelector	u32	Selector (identifier) of the function to be called as entrypoint in the origin contract.
argsHash	Field	Hash of the arguments to be used for calling the entrypoint function.

Field	Type	Description
txContext	TxContext	Includes chain id, protocol version, and gas settings.
packedArguments	PackedValues[]	Preimages for argument hashes. When executing a function call with the hash of the arguments, the PXE will look for the preimage of that hash in this list, and expand the arguments to execute the call.
authWitnesses	AuthWitness[]	Authorization witnesses. When authorizing an action identified by a hash, the PXE will look for the authorization witness identified by that hash and provide that value to the account contract.

Simulation step

Upon receiving a transaction execution request to *simulate*, the PXE will locally execute the function identified by the given `functionSelector` in the given `origin` contract with the arguments committed to by `argsHash`. We refer to this function as the *entrypoint*. During execution, contracts may request authorization witnesses or expanded arguments from the *execution oracle*, which are answered with the `packedArguments` and `authWitnesses` from the request.

The *entrypoint* may enqueue additional function calls, either private or public. The

simulation step will always execute all private functions in the call stack until emptied. The result of the simulation is a *transaction* object without an associated *proof* which is returned to the application that requested the simulation.

In terms of circuitry, the simulation step must execute all application circuits that correspond to private function calls, and then execute the private kernel circuit until the private call stack is empty. Note that circuits are only executed, there is no witness generation or proving involved.

Proving step

The proving step is similar to the simulation step, though witnesses are generated for all circuits and proven. Note that it is not necessary to execute the simulation step before the proving step, though it is desirable in order to provide the user with info on their transaction and catch any failed assertions early.

The output of the proving step is a *transaction* object with a valid *proof* associated, ready to be broadcasted to the network.

Public execution

Transactions have a *public execution* component. Once a transaction is picked up by a sequencer to be included in a block, the sequencer is responsible for executing all enqueued public function calls in the transaction. These are defined by the `data.accumulatedData.publicCallStack` field of the [transaction object](#), which are commitments to the preimages of the `enqueuedPublicFunctionCalls` in the transaction. The sequencer pops function calls from the stack, and pushes new ones as needed, until the public call stack is empty.

Bytecode

Unlike private functions, which are native circuits, public functions in the Aztec Network are specified in AVM bytecode . This bytecode is executed and proven in the Aztec Virtual Machine. Each enqueued public function spawns a new instance of the AVM, and a *public kernel circuit* aggregates these calls and produces a final proof of the transaction, which also includes the *private kernel circuit* proof of the transaction generated during [local execution](#).

State

Since public execution is run by the sequencer, it is run on the very-latest state of the chain as it is when the transaction is included in the block. Public functions operate on [public state](#), an updateable key-value mapping, instead of notes.

Reverts

Note that, unlike local private execution, public execution can *revert* due to a failed

assertion, running out of gas, trying to call a non-existing function, or other failures. If this happens, the sequencer halts execution and discards all side effects from the [transaction payload phase](#). The transaction is still included in the block and pays fees, but is flagged as reverted.

Transaction object

The transaction object is the struct broadcasted to the p2p network, generated by [local execution](#) by the user's PXE. Sequencers pick up transactions from the p2p network to include in a block.

Transaction object struct

The fields of a transaction object are the following:

Field	Type	Description
data	PrivateKernelPublicInputsFinal	Public inputs (ie output) of the last iteration of the private kernel circuit for this transaction.
proof	Buffer	Zero-knowledge honk proof for the last iteration of the private kernel circuit for this transaction.

Field	Type	Description
encryptedLogs	Buffer[][]	Encrypted logs emitted per function in this transaction. Position i contains the encrypted logs emitted by the i -th function execution.
unencryptedLogs	Buffer[][]	Equivalent to the above but for unencrypted logs.
queuedPublicFunctionCalls	PublicCallRequest[]	List of public function calls to run during public execution.

Private kernel public inputs final

Output of the last iteration of the private kernel circuit. Includes *accumulated data*

after recursing through all private function calls, as well as *constant data* composed of *block header* reflecting the state of the chain when such functions were executed, and the global *transaction context*. Refer to the circuits section for more info.

Accumulated data

Field	Type	Description
noteHashes	Field[]	The new note hashes made in this transaction.
nullifiers	Field[]	The new nullifiers made in this transaction.
nullifiedNoteHashes	Field[]	The note hashes which are nullified by a nullifier in the above list.
privateCallStack	Field[]	Current private call stack.
publicCallStack	Field[]	Current public call stack.
I2ToL1Msgs	Field[]	All the new L2 to L1 messages created in this transaction.
encryptedLogsHash	Field[]	Accumulated encrypted logs hash from all the previous kernel iterations.
unencryptedLogsHash	Field[]	Accumulated unencrypted logs hash from all the previous kernel

Field	Type	Description
		iterations.
encryptedLogPreimagesLength	Field	Total accumulated length of the encrypted log preimages emitted in all the previous kernel iterations.
unencryptedLogPreimagesLength	Field	Total accumulated length of the unencrypted log preimages emitted in all the previous kernel iterations.
maxBlockNum	Field	Maximum block number (inclusive) for inclusion of this transaction in a block.

Block header

Field	Type	Description
noteHashTreeRoot	Field	Root of the note hash tree at the time of when this information was assembled.
nullifierTreeRoot	Field	Root of the nullifier tree at the time of when this information was assembled.
contractTreeRoot	Field	Root of the contract tree at the time of when this information was assembled.

Field	Type	Description
I1ToL2MessageTreeRoot	Field	Root of the L1 to L2 message tree at the time of when this information was assembled.
archiveRoot	Field	Root of the archive at the time of when this information was assembled.
privateKernelVkTreeRoot	Field	Root of the private kernel VK tree at the time of when this information was assembled (future enhancement).
publicDataTreeRoot	Field	Current public state tree hash.
globalVariablesHash	Field	Previous globals hash, this value is used to recalculate the block hash.

Public call request

Each *public call request* is the preimage of a public call stack item in the transaction's `data`, and has the following fields:

Field	Type	Description
contractAddress	AztecAddress	Address of the contract on which the function is invoked.
callContext	CallContext	Includes function selector and caller.

Field	Type	Description
args	Field[]	Arguments to the function call.
sideEffectCounter	number?	Optional counter for ordering side effects of this function call.

Extended contract data

Each *extended contract data* corresponds to a contract being deployed by the transaction, and has the following fields:

Field	Type	Description
address	AztecAddress	Address where the contract is to be deployed.
portalAddress	EthereumAddress	Portal address on L1 for this contract (zero if none).
bytecode	Buffer	Encoded Brilliq bytecode for all public functions in the contract.
publicKey	PublicKey	Master public encryption key for this contract (zero if none).
partialAddress	Field	Hash of the constructor arguments, salt, and bytecode.

Transaction hash

A transaction is identified by its `transaction_hash`. In order to be able to identify a transaction before it has been locally executed, the hash is computed from its *transaction execution request* by hashing:

- `origin`
- `functionSelector`
- `argsHash`
- `txContent`

The resulting transaction hash is always emitted during local execution as the first nullifier of the transaction, in order to prevent replay attacks. This is enforced by the private kernel circuit.

Validity conditions

The *validity conditions* of a transaction define when a *transaction object* is valid.

Nodes should check the validity of a transaction when they receive it either directly or through the p2p pool, and if they find it to be invalid, should drop it immediately and not broadcast it.

In addition to being well-formed, the transaction object needs to pass the following checks:

- **Proof is valid:** The `proof` for the given public `data` should be valid according to a protocol-wide verification key for the final private kernel circuit.
- **No duplicate nullifiers:** No `nullifier` in the transaction `data` should be already present in the nullifier tree.
- **No pending private function calls:** The `data` private call stack should be empty.
- **Valid historic data:** The tree roots in the block header of `data` must match the tree roots of a historical block in the chain.
- **Maximum block number not exceeded:** The transaction must be included in a block with height no greater than the value specified in `maxBlockNum` within the transaction's `data`.
- **Preimages must match commitments in `data`:** The expanded fields in the transaction object should match the commitments (hashes) to them in the public `data`.
 - The `encryptedLogs` should match the `encryptedLogsHash` and `encryptedLogPreimagesLength` in the transaction `data`.
 - The `unencryptedLogs` should match the `unencryptedLogsHash` and `unencryptedLogPreimagesLength` in the transaction `data`.
 - Each public call stack item in the transaction `data` should have a corresponding preimage in the `enqueuedPublicFunctionCalls`.

- Each new contract data in transaction `data` should have a corresponding preimage in the `newContracts`.
- Able to pay fee: The [fee can be paid](#).

Note that all checks but the last one are enforced by the base rollup circuit when the transaction is included in a block.

This section describes how contracts are represented within the protocol for execution.

In the context of Aztec, a contract is a set of functions which can be of one of three types:

- Private functions: The functions that run on user's machines. They are circuits that must be individually executed by the [ACVM](#) and proved by barretenberg.
- Public functions: The functions that are run by sequencers. They are aggregated in a bytecode block that must be executed and proven by the AVM.
- Unconstrained functions: Helper functions that are run on users' machines but are not constrained. They are represented individually as bytecode that is executed by the ACVM.
 - Unconstrained functions are often used to fetch and serialize private data, for use as witnesses to a circuit.
 - They can also be used to convey how dapps should handle a particular contract's data.

When a contract is compiled, private and unconstrained functions are compiled individually. Public functions are compiled together to a single bytecode with an initial dispatch table based on function selectors. Since public functions are run in a VM, we do not incur a huge extra proving cost for the branching that is required to execute different functions.

If a private function needs unconstrained hints, the bytecode that generates the unconstrained hints is embedded in the private circuit. This allows the ACVM to compute the hints during witness generation.

There are three different (but related) bytecode standards that are used in Aztec: AVM bytecode, Brillig bytecode and ACIR bytecode.

AVM Bytecode

The AVM bytecode is the compilation target of the public functions of a contract. It's specified in the [AVM section](#). It allows control flow and uses a flat memory model which tracks bit sizes of values stored in memory via tagging of memory indexes. Sequencers run the AVM bytecode of the public functions of a contract using the AVM and prove the correct execution of it.

Brillig Bytecode

Brillig bytecode is the compilation target of all the unconstrained code in a contract. Any unconstrained hint used by a private function is compiled to Brillig bytecode. Also, contracts' top level unconstrained functions are entirely compiled to Brillig bytecode. In the case of Noir, it compiles public functions entirely to a single block of brillig bytecode that is then converted to AVM bytecode. Similarly to AVM bytecode, Brillig bytecode allows control flow.

Brillig bytecode will be very similar to AVM bytecode. While AVM bytecode is specifically designed to be executed by the AVM, brillig bytecode is meant to be more general and allow the use of arbitrary oracles.

Oracles allow nondeterminism during the execution of a given function, allowing the simulator entity to choose the value that an oracle will return during the simulation process. Oracles are heavily used by aztec.nr to fetch data during simulation of private and unconstrained functions, such as fetching notes. They are also used to notify the simulator about events arising during execution, such as a nullified note so that it's not offered again during the simulation.

However, AVM bytecode doesn't allow arbitrary oracles, any nondeterminism introduced is done in a way that the protocol can ensure that the simulator entity (the sequencer) cannot manipulate the result of an oracle. As such, when transforming brillig bytecode to AVM bytecode, all the oracles are replaced by the specific

opcodes that the AVM supports for nondeterminism, like `TIMESTAMP`, `ADDRESS`, etc. Any opcode that requires the simulator entity to provide data external to the AVM memory is non-deterministic.

The current implementation of Brillig can be found [in the noir repository](#). It's actively being changed to become "AVM bytecode without arbitrary oracles" and right now the differences are handled by a transpiler.

ACIR Bytecode

ACIR bytecode is the compilation target of contract private functions. ACIR expresses arithmetic circuits and thus has no control flow. Control flow in regular functions is either unrolled (for loops) or flattened (by inlining and adding predicates), resulting in a single function with no control flow to be transformed to ACIR.

The types of opcodes that can appear in ACIR are:

- Arithmetic: They can express any degree-2 multivariate relation between witness indices. They are the most common opcodes in ACIR.
- BlackBoxFuncCall: They assign the witnesses of the parameters and the witnesses of the return values of black box functions. Black box functions are commonly used operations that are treated as a black box, meaning that the underlying backend chooses how to prove them efficiently.
- Brillig: This opcode contains a block of brillig bytecode, witness indices of the parameters and witness indices of the return values. When ACIR bytecode needs an unconstrained hint, the bytecode that is able to generate the hint at runtime is embedded in a Brillig opcode, and the result of running the hint is assigned to the return witnesses specified in the opcode. The simulator entity is the one responsible for executing the brillig bytecode. The results of the execution of the function are assigned to the witnesses of the return values and they should be constrained to be correct by the ACIR bytecode.
- MemoryOp: They handle memory operations. When accessing arrays with

indices unknown at compile time, the compiler cannot know which witness index is being read. The memory abstraction allows acir to read and write to dynamic positions in arrays in an efficient manner, offloading the responsibility of proving the correct access to the underlying backend.

This implies that a block of ACIR bytecode can represent more than one program, since it can contain any number of Brillig opcodes each one containing a full Brillig program that computes a hint that the circuit needs at runtime.

Usage of the bytecode

Compiling a contract

When a contract is compiled, an artifact will be generated. This artifact needs to be hashed in a specific manner [detailed in the deployment section](#) for publishing.

The exact form of the artifact is not specified by the protocol, but it needs at least the following information:

Contract artifact

Field	Type	Description
<code>name</code>	<code>string</code>	The name of the contract.
<code>compilerVersion</code>	<code>string</code>	Version of the compiler that generated the bytecode. This is a string to convey extra information like the version of Aztec.nr used.

Field	Type	Description
functions	FunctionEntry[]	The functions of the contract.
publicBytecode	string	The AVM bytecode of the public functions, converted to base64.
events	EventAbi[]	The events of the contract.

Event ABI

Field	Type	Description
name	string	The event name.
fields	ABIVariable	The fields of the event.

Function entry

If the function is public, the entry will be its ABI. If the function is private or unconstrained, the entry will be the ABI + the artifact.

Function artifact

Field	Type	Description
bytecode	string	The ACIR bytecode of the function, converted to base64.

Function ABI

Field	Type	Description
name	string	The name of the function.
functionType	string	private, public or unconstrained.
parameters	ABIParameter[]	Function parameters.
returnTypes	AbiType[]	The types of the return values.

ABI Variable

Field	Type	Description
name	string	The name of the variable.
type	AbiType	The type of the variable.

ABI Parameter

Field	Type	Description
name	string	The name of the variable.

Field	Type	Description
<code>type</code>	<code>AbiType</code>	The type of the variable.
<code>visibility</code>	<code>string</code>	<code>public</code> or <code>secret</code> .

ABI Type

Field	Type	Description
<code>kind</code>	<code>string</code>	<code>field</code> , <code>boolean</code> , <code>integer</code> , <code>array</code> , <code>string</code> or <code>struct</code>
<code>sign?</code>	<code>string</code>	The sign of the integer. Applies to integers only.
<code>width?</code>	<code>number</code>	The width of the integer in bits. Applies to integers only.
<code>length?</code>	<code>number</code>	The length of the array or string. Applies to arrays and strings only.
<code>type?</code>	<code>AbiType</code>	The types of the array elements. Applies to arrays only.
<code>fields?</code>	<code>ABIVariable[]</code>	the fields of the struct. Applies to structs only.

Bytecode in the artifact

The protocol mandates that public bytecode needs to be published to a data availability solution, since the sequencers need to have the data available to run the public functions. Also, it needs to use an encoding that is friendly to the public VM, such as the one specified in the [AVM section](#).

The bytecode of private and unconstrained functions doesn't need to be published, instead, users that desire to use a given contract can add the artifact to their PXE before interacting with it. Publishing it is [supported but not required](#) by the protocol. However, the verification key of a private function is hashed into the function's leaf of the contract's function tree, so the user can prove to the protocol that he executed the function correctly. Also, contract classes contain an [artifact hash](#) so the PXE can verify that the artifact corresponds with the contract class.

The encoding of private and unconstrained functions is not specified by the protocol, but it's recommended to follow [the encoding](#) that Barretenberg and the ACVM share that is serialization using bincode and gzip for compression.

This implies that the encoding of private and unconstrained functions does not need to be friendly to circuits, since when publishing it the protocol only sees a [generic array of field elements](#).

Executing a private function

When executing a private function, its ACIR bytecode will be executed by the PXE using the ACVM. The ACVM will generate the witness of the execution. The proving system can be used to generate a proof of the correctness of the witness.

The fact that the correct function was executed is checked by the protocol by verifying that the [contract class ID](#) contains one leaf in the function tree with this

selector and the verification key of the function.

Executing an unconstrained function

When executing an unconstrained function, its Brillig bytecode will be executed by the PXE using the ACVM, similarly to private functions, but the PXE will not prove the execution. Instead, the PXE will return the result of the execution of the function to the user.

Executing a public function

When executing a public function, its AVM bytecode will be executed by the sequencer with the specified selector and arguments. The sequencer will generate a public VM proof of the correct execution of the AVM bytecode.

The fact that the correct bytecode was executed is checked by the protocol by verifying that the [contract class ID](#) contains the [commitment](#) to the bytecode used.

Contract Deployment

Contracts in Aztec are deployed as *instances* of a contract *class*. Deploying a new contract then requires first registering the *class*, if it has not been registered before, and then creating an *instance* that references the class. Both classes and instances are committed to in the nullifier tree in the global state, and are created via a call to a canonical class registry or instance deployer contract respectively.

Contract classes

A contract class is a collection of state variable declarations, and related unconstrained, private, and public functions. Contract classes don't have any initialized state, they j...

Contract instances

A contract instance is a concrete deployment of a contract class. A contract instance always references a contract class, which dictates what code it executes when called. ...

Contract classes

A contract class is a collection of state variable declarations, and related unconstrained, private, and public functions. Contract classes don't have any initialized state, they just define code. A contract class cannot be called; only a contract instance can be called.

Rationale

Contract classes simplify the process of reusing code by enshrining implementations as a first-class citizen at the protocol. Given multiple [contract instances](#) that rely on the same class, the class needs to be declared only once, reducing the deployment cost for all contract instances. Classes also simplify the process of upgradeability; classes decouple state from code, making it easier for an instance to switch to different code while retaining its state.

! INFO

Read the following discussions for additional context:

- [Abstracting contract deployment](#)
- [Implementing contract upgrades](#)
- [Contract classes, upgrades, and default accounts](#)

ContractClass

The structure of a contract class is defined as:

Field	Type	Description
version	u8	Version identifier. Initially one, bumped for any changes to the contract class struct.

Field	Type	Description
<code>artifact_hash</code>	<code>Field</code>	Hash of the contract artifact. The specification of this hash is not enforced by the protocol. Should include commitments to unconstrained code and compilation metadata. Intended to be used by clients to verify that an off-chain fetched artifact matches a registered class.
<code>private_functions</code>	<code>PrivateFunction[]</code>	List of individual private functions, constructors included.
<code>packed_public_bytecode</code>	<code>Field[]</code>	Packed bytecode representation of the AVM bytecode for all public functions in this contract.

The public function are sorted in ascending order by their function selector before being packed. This is to ensure consistent hashing later.

Note that individual public functions are not first-class citizens in the protocol, so the contract entire public function bytecode is stored in the class, unlike private or unconstrained functions which are differentiated individual circuits recognized by the protocol.

As for unconstrained functions, these are not used standalone within the protocol. They are either inlined within private functions, or called from a PXE as *getters* for a contract. Calling from a private function to an unconstrained one in a different contract is forbidden, since the caller would have no guarantee of the code run by the callee. Considering this, unconstrained functions are not part of a contract class at the protocol level.

`contract_class_id`

Also known as `contract_class_id`, the Class Identifier is both a unique identifier and a

commitment to the struct contents. It is computed as:

```
contract_class_id_crh(
    artifact_hash: Field
    private_functions: PrivateFunction[],
    packed_public_bytecode: bytes[],
) -> Field {
    let private_function_leaves: Field[] = private_functions.map(|f|
private_function_leaf_crh(f));

    // Illustrative function, not defined. TODO.
    let private_function_tree_root: Field =
merkleize(private_function_leaves);

    // Illustrative function, not defined. TODO.
    let public_bytecode_commitment: Point =
calculate_commitment(packed_public_bytecode);

    let contract_class_id = poseidon2(
        be_string_to_field("az_contract_class_id"),

        artifact_hash,
        private_function_tree_root,
        public_bytecode_commitment.x,
        public_bytecode_commitment.y,
    );
}

contract_class_id
}
```

See below for `private_function_leaf_crh`. Private Functions are sorted in ascending order by their selector, and then hashed into Function Leaves, before being merkleized into a tree of height `PRIVATE_FUNCTION_TREE_HEIGHT`. Empty leaves have value `0`. The AVM public bytecode commitment is calculated as [defined in the Public VM section](#).

PrivateFunction

The structure of each private function within the protocol is the following:

Field	Type	Description
function_selector	u32	Selector of the function. Calculated as the hash of the method name and parameters. The specification of this is not enforced by the protocol.
vk_hash	Field	Hash of the verification key associated to this private function.

Note the lack of visibility modifiers. Internal functions are specified as a macro, and the check is handled at the application circuit level by verifying that the `context.msg_sender` equals the contract current address.

Also note the lack of commitment to the function compilation artifact. Even though a commitment to a function is required so that the PXE can verify the execution of correct unconstrained Brilliig code embedded within private functions, this is handled entirely out of protocol. As such, PXEs are expected to verify it against the `artifact_hash` in the containing contract class.

Private Function Leaf Hash

```

private_function_leaf_crh(
    f: PrivateFunction
) -> Field {
    let private_function_leaf = poseidon2(
        be_string_to_field("az_private_function_leaf"),
        be_bits_to_field(f.function_selector),
        f.vk_hash
    );
    private_function_leaf
}

```

Artifact Hash

Even though not enforced by the protocol, it is suggested for the `artifact_hash` to follow this general structure, in order to be compatible with the definition of the `broadcast` function below.

Note: below, `sha256_modulo(x) = sha256(x) % FIELD_MODULUS`. This approach must not be used if seeking pseudo-randomness, but can be used for collision resistance.

```
artifact_crh(  
    artifact // This type is out of protocol, e.g. the format output by Nargo  
) -> Field {  
  
    let private_functions_artifact_leaves: Field[] =  
        artifact.private_functions.map(|f|  
            sha256_modulo(  
                be_string_to_bits("az_artifact_private_function_leaf"),  
  
                f.selector, // 32-bits  
                f.metadata_hash, // 256-bits  
                sha256(f.private_bytecode)  
            )  
        );  
    let private_functions_artifact_tree_root: Field =  
        merkleize(private_functions_artifact_leaves);  
  
    let unconstrained_functions_artifact_leaves: Field[] =  
        artifact.unconstrained_functions.map(|f|  
            sha256_modulo(  
                be_string_to_bits("az_artifact_unconstrained_function_leaf"),  
  
                f.selector, // 32-bits  
                f.metadata_hash, // 256-bits  
                sha256(f.unconstrained_bytecode)  
            )  
        );  
    let unconstrained_functions_artifact_tree_root: Field =  
        merkleize(unconstrained_functions_artifact_leaves);  
  
    let artifact_hash: Field = sha256_modulo(  
}
```

For the artifact hash merkleization and hashing is done using sha256, since it is computed and verified outside of circuits and does not need to be SNARK friendly, and then wrapped around the field's maximum value. Fields are left-padded with zeros to 256 bits before being hashed. Function leaves are sorted in ascending order before being merkleized, according to their function selectors. Note that a tree with dynamic height is built instead of having a tree with a fixed height, since the merkleization is done out of a circuit.

Bytecode for private functions is a mix of ACIR and Brillig, whereas unconstrained function bytecode is Brillig exclusively, as described on the [bytecode section](#).

The metadata hash for each function is suggested to be computed as the sha256 of all JSON-serialized fields in the function struct of the compilation artifact, except for bytecode and debug symbols. The metadata is JSON-serialized using no spaces, and sorting ascending all keys in objects before serializing them.

```
function_metadata_crh
  function // This type is out of protocol, e.g. the format output by Nargo
) -> Field {
  let function_metadata = omit(function, "bytecode", "debug_symbols");

  let function_metadata_hash: Field = sha256_modulo(
    be_string_to_bits("az_function_metadata"),

    json_serialize(function_metadata)
  );

  function_metadata_hash
}
```

The artifact metadata stores all data that is not contained within the contract functions and is not debug specific. This includes the compiler version identifier, events interface, and name. Metadata is JSON-serialized in the same fashion as the function metadata.

```
artifact_metadata_crh
  artifact // This type is out of protocol, e.g. the format output by Nargo
) -> Field {
  let artifact_metadata = omit(artifact, "functions", "file_map");
```

Versioning

A contract class has an implicit `version` field that identifies the schema of the struct. This allows to change the shape of a contract class in future upgrades to the protocol to include new fields or change existing ones, while preserving the structure for existing classes. Supporting new types of contract classes would require introducing new kernel circuits, and a transaction proof may require switching between different kernel circuits depending on the version of the contract class used for each function call.

Note that the `version` field is not directly used when computing the contract class id, but is implicit in the generator index. Bumping the `version` of a contract class struct would involve using a different generator index for computing its id.

Canonical Contract Class Registerer

A contract class is registered by calling a private `register` function in a canonical `ContractClassRegisterer` contract, which will emit a Registration Nullifier. The Registration Nullifier is defined as the `contract_class_id` itself of the class being registered. Note that the Private Kernel circuit will `silo` this value with the contract address of the `ContractClassRegisterer`, effectively storing the hash of the `contract_class_id` and `ContractClassRegisterer` address in the nullifier tree. As such, proving that a given contract class has been registered requires checking existence of this siloed nullifier.

The rationale for the Registerer contract is to guarantee that the public bytecode for a contract class is publicly available. This is a requirement for publicly [deploying a contract instance](#), which ultimately prevents a sequencer from executing a public function for which other nodes in the network may not have the code.

Register Function

The `register` function receives the artifact hash, private functions tree root, and packed public bytecode of a `ContractClass` struct as [defined above](#), and performs the following steps:

- Assert that `packed_public_bytecode` is valid according to the definition in the [Public VM](#)

section.

- Computes the `contract_class_id` as defined above.
- Emits the resulting `contract_class_id` as a nullifier to prevent the same class from being registered again.
- Emits an unencrypted event `ContractClassRegistered` with the contents of the contract class.

In pseudocode:

```
fn register(
    artifact_hash: Field,
    private_functions_root: Field,
    public_bytecode_commitment: Point,
    packed_public_bytecode: Field[],
) {
    assert(is_valid_packed_public_bytecode(packed_public_bytecode));

    let computed_bytecode_commitment: Point =
        calculate_commitment(packed_public_bytecode);

    assert(public_bytecode_commitment == computed_bytecode_commitment);

    let version: Field = 1;
    let contract_class_id = contract_class_id_crh(version, artifact_hash,
        private_functions_root, bytecode_commitment);

    emit_nullifier(contract_class_id);

    emit_unencrypted_event(ContractClassRegistered::new(
        contract_class_id,
        version,
        artifact_hash,
        private_functions_root,
        packed_public_bytecode
    ));
}
```

Upon seeing a `ContractClassRegistered` event in a mined transaction, nodes are expected to store the contract class, so they can retrieve it when executing a public function for that class.

Note that a class may be used for deploying a contract within the same transaction in which it is registered.

Note that emitting the `contract_class_id` as a nullifier (the `contract_class_id_nullifier`), instead of as an entry in the note hashes tree, allows nodes to prove non-existence of a class. This is needed so a sequencer can provably revert a transaction that includes a call to an unregistered class.

Genesis

The `ContractClassRegisterer` will need to exist from the genesis of the Aztec Network, otherwise nothing will ever be publicly deployable to the network. The Class Nullifier for the `ContractClassRegisterer` contract will be pre-inserted into the genesis nullifier tree at leaf index

`GENESIS_NULLIFIER_LEAF_INDEX_OF_CONTRACT_CLASS_REGISTERER_CLASS_ID_NULLIFIER`.

The canonical instance will be deployed at `CONTRACT_CLASS_REGISTERER_ADDRESS`, and its Deployment Nullifier will be inserted at

`GENESIS_NULLIFIER_LEAF_INDEX_OF_CONTRACT_CLASS_REGISTERER_DEPLOYMENT_NULLIFIER`.

Broadcast

The `ContractClassRegisterer` has an additional private `broadcast` functions that can be used for broadcasting on-chain the bytecode, both ACIR and Brillig, for private functions and unconstrained in the contract. Any user can freely call this function. Given that ACIR and Brillig do not have a circuit-friendly commitment, it is left up to nodes to perform this check.

Broadcasted function artifacts that do not match with their corresponding `artifact_hash`, or that reference a `contract_class_id` that has not been broadcasted, can be safely discarded.

```
fn broadcast_private_function(  
    contract_class_id: Field,  
    artifact_metadata_hash: Field,  
    unconstrained_functions_artifact_tree_root: Field,  
    private_function_tree_sibling_path: Field[],  
    private_function_tree_leaf_index: Field,  
    artifact_function_tree_sibling_path: Field[],
```

```

fn broadcast_unconstrained_function(
    contract_class_id: Field,
    artifact_metadata_hash: Field,
    private_functions_artifact_tree_root: Field,
    artifact_function_tree_sibling_path: Field[],
    artifact_function_tree_leaf_index: Field
    function: { selector: Field, metadata_hash: Field, bytecode: Field[] }[],
)
emit_unencrypted_event ClassUnconstrainedFunctionBroadcasted(
    contract_class_id,
    artifact_metadata_hash,
    private_functions_artifact_tree_root,
    artifact_function_tree_sibling_path,
    artifact_function_tree_leaf_index,
    function,
)

```

The broadcast functions are split between private and unconstrained to allow for private bytecode to be broadcasted, which is valuable for composability purposes, without having to also include unconstrained functions, which could be costly to do due to data broadcasting costs. Additionally, note that each broadcast function must include enough information to reconstruct the `artifact_hash` from the Contract Class, so nodes can verify it against the one previously registered.

A node that captures a `ClassPrivateFunctionBroadcasted` should perform the following validation steps before storing the private function information in its database:

```

// Load contract class from local db
contract_class = db.get_contract_class(contract_class_id)

// Compute function leaf and assert it belongs to the private functions
tree
function_leaf = pedersen([selector as Field, vk_hash],
GENERATOR__FUNCTION_LEAF)
computed_private_function_tree_root = compute_root(function_leaf,
private_function_tree_sibling_path, private_function_tree_leaf_index)
assert computed_private_function_tree_root ==
contract_class.private_function_root

```

The check for an unconstrained function is similar:

```
// Load contract class from local db
contract_class = db.get_contract_class(contract_class_id)

// Compute artifact leaf and assert it belongs to the artifact
artifact_function_leaf = sha256(selector, metadata_hash, sha256(bytecode))
computed_artifact_unconstrained_function_tree_root =
compute_root(artifact_function_leaf, artifact_function_tree_sibling_path,
artifact_function_tree_leaf_index)
computed_artifact_hash = sha256(private_functions_artifact_tree_root,
computed_artifact_unconstrained_function_tree_root, artifact_metadata_hash)
assert computed_artifact_hash == contract_class.artifact_hash
```

It is strongly recommended for developers registering new classes to broadcast the code for `compute_hash_and_nullifier`, so any private message recipients have the code available to process their incoming notes. However, the `ContractClassRegisterer` contract does not enforce this during registration, since it is difficult to check the multiple signatures for `compute_hash_and_nullifier` as they may evolve over time to account for new note sizes.

Encoding Bytecode

The `register`, `broadcast_unconstrained_function`, and `broadcast_private_function` functions all receive and emit variable-length bytecode in unencrypted events. In every function, bytecode is encoded in a fixed-length array of field elements, which sets a maximum length for each:

- `MAX_PACKED_PUBLIC_BYTECODE_SIZE_IN_FIELDS`: 3000 field elements, used for a contract's public bytecode in the `register` function.
- `MAX_PACKED_BYTECODE_SIZE_PER_PRIVATE_FUNCTION_IN_FIELDS`: 3000 field elements, used for the ACIR and Brillig bytecode of a broadcasted private function in `broadcast_private_function`.
- `MAX_PACKED_BYTECODE_SIZE_PER_UNCONSTRAINED_FUNCTION_IN_FIELDS`: 3000 field elements, used for the Brillig bytecode of a broadcasted unconstrained function in `broadcast_unconstrained_function`.

To encode the bytecode into a fixed-length array of Fields, the bytecode is first split into 31-byte chunks, and each chunk interpreted big-endian as a field element. The total length in bytes is then prepended as an initial element, and then right-padded with zeroes.

```
chunks = chunk bytecode into 31 bytes elements, last element right-padded  
with zeroes  
fields = right-align each chunk into 32 bytes and cast to a field element  
padding = repeat a zero-value field MAX_SIZE - fields.count - 1 times  
encoded = [bytecode.length as field, ...fields, ...padding]
```

Discarded Approaches

Bundling private function information into a single tree

Data about private functions is split across two trees: one for the protocol, that deals only with selectors and verification keys, and one for the artifact, which deals with bytecode and metadata. While bundling together both trees would simplify the representation, it would also pollute the protocol circuits and require more hashing there. In order to minimize in-circuit hashing, we opted for keeping non-protocol info completely out of circuits.

Contract instances

A contract instance is a concrete deployment of a [contract class](#). A contract instance always references a contract class, which dictates what code it executes when called. A contract instance has state (both private and public), as well as an address that acts as its identifier. A contract instance can be called into.

Requirements

- Users must be able to precompute the address of a given contract instance. This allows users to precompute their account contract addresses and receive funds before interacting with the chain, and also allows counterfactual deployments.
- An address must be linkable to its deployer address. This allows simple diversified and stealth account contracts. Related, a precomputed deployment may or may not be restricted to be executed by a given address.
- A user calling into an address must be able to prove that it has not been deployed. This allows the executor to prove that a given call in a transaction is unsatisfiable and revert accordingly.
- A user should be able to privately call into a contract without publicly deploying it. This allows private applications to deploy contracts without leaking information about their usage.

ContractInstance

The structure of a contract instance is defined as:

Field	Type	Description
version	u8	Version identifier. Initially one, bumped for any changes to the contract instance struct.
salt	Field	User-generated pseudorandom value for uniqueness.

Field	Type	Description
<code>deployer</code>	<code>AztecAddress</code>	Optional address of the deployer of the contract.
<code>contract_class_id</code>	<code>Field</code>	Identifier of the contract class for this instance.
<code>initialization_hash</code>	<code>Field</code>	Hash of the selector and arguments to the constructor.
<code>public_keys_hash</code>	<code>Field</code>	Optional hash of the struct of public keys used for encryption and nullifying by this contract.

Versioning

Contract instances have a `version` field that identifies the schema of the instance, allowing for changes to the struct in future versions of the protocol, same as the contract class [version](#).

Address

The address of the contract instance is computed as the hash of the elements in the structure above, as defined in [the addresses and keys section](#). This computation is deterministic, which allows any user to precompute the expected deployment address of their contract, including account contracts.

Deployer

The `deployer` address of a contract instance is used to restrict who can initialize the contract (ie call its constructor) and who can publicly deploy it. Note that neither of these checks are enforced by the protocol: the initialization is checked by the constructor itself, and the deployment by the `ContractInstanceDeployer` (described below). Furthermore, a contract class may choose to not enforce this restriction by removing the check from the constructor.

The `deployer` address can be set to zero to signal that anyone can initialize or publicly deploy an

instance.

Initialization

A contract instance at a given address can be either Initialized or not. An address by default is not initialized, and it is considered to be Initialized once it emits an Initialization Nullifier, meaning it can only be initialized once.

Uninitialized

The default state for any given address is to be uninitialized, meaning its constructor has not been called. A user who knows the preimage of the address can still issue a private call into a function in the contract, as long as that function does not assert that the contract has been initialized by checking the Initialization Nullifier.

All function calls to an Uninitialized contract that depend on the contract being initialized should fail, to prevent the contract from being used in an invalid state.

This state allows using a contract privately before it has been initialized or deployed, which is used in [diversified and stealth accounts](#).

Initialized

An instance is Initialized when a constructor for the instance has been invoked, and the constructor has emitted the instance's Initialization Nullifier. All private functions that require the contract to be initialized by checking the existence of the Initialization Nullifier can now be called by any user who knows the address preimage.

The Initialization Nullifier is defined as the contract address itself. Note that the nullifier later gets [siloed by the Private Kernel Circuit](#) before it gets broadcasted in a transaction.

WARNING

It may be the case that it is not possible to read a nullifier in the same transaction that it was emitted due to protocol limitations. That would lead to a contract not being callable in the same transaction as it is initialized. To work around this, we can emit an Initialization

Commitment along with the Initialization Nullifier, which *can* be read in the same transaction as it is emitted. If needed, the Initialization Commitment is defined exactly as the Initialization Nullifier.

Constructors

Contract constructors are not enshrined in the protocol, but handled at the application circuit level. Constructors are methods used for initializing a contract, either private or public, and a contract may have more than a single constructor. A contract must ensure the following requirements are met:

- A contract may be initialized at most once
- A contract must be initialized using the method and arguments defined in its address preimage
- A contract must be initialized by its `deployer` (if it's non-zero)
- All functions that depend on contract initialization cannot be invoked until the contract is initialized

These checks are embedded in the application circuits themselves. The constructor emits an Initialization Nullifier when it is invoked, which prevents it from being called more than once. The constructor code must also check that its own selector and the arguments for the call match the ones in the address preimage, which are supplied via an oracle call.

All non-constructor functions in the contract should require a merkle membership proof for the Initialization Nullifier, to prevent them from being called before the constructor is invoked. Nevertheless, a contract may choose to allow some functions to be called before initialization, such as in the case of [Diversified and Stealth account contracts](#).

Removing constructors from the protocol itself simplifies the kernel circuit, and decoupling Initialization from Public Deployments allows users to keep contract instances private if they wish to do so.

Public Deployment

A Contract Instance is considered to be Publicly Deployed when it has been broadcasted to the

network via a canonical `ContractInstanceDeployer` contract, which also emits a Deployment Nullifier associated to the deployed instance.

All public function calls to an Undeployed address *must* fail, since the Contract Class for it is not known to the network. If the Class is not known to the network, then an Aztec Node, whether it is the elected sequencer or a full node following the chain, may not be able to execute the bytecode for a public function call, which is undesirable.

The failing of public function calls to Undeployed addresses is enforced by having the Public Kernel Circuit check that the Deployment Nullifier for the instance has been emitted. Note that makes Public Deployment a protocol-level concern, whereas Initialization is purely an application-level concern. Also, note that this requires hardcoding the address of the `ContractInstanceDeployer` contract in a protocol circuit.

The Deployment Nullifier is defined as the address of the contract being deployed. Note that it later gets *siloed* using the `ContractInstanceDeployer` address by the Kernel Circuit, so this nullifier is effectively the hash of the deployed contract address and the `ContractInstanceDeployer` address.

Canonical Contract Instance Deployer

A new contract instance can be *Publicly Deployed* by calling a `deploy` function in a canonical `ContractInstanceDeployer` contract. This function receives the arguments for a `ContractInstance` struct as described [above](#):

- Validates the referenced `contract_class_id` exists. This can be done via either a call to the `ClassRegisterer` contract, or by directly reading the corresponding nullifier.
- Set `deployer` to zero or `msg_sender` depending on whether the `universal_deploy` flag is set.
- Computes the resulting `new_contract_address`.
- Emits the resulting address as the Deployment Nullifier to signal the public deployment, so callers can prove that the contract has or has not been publicly deployed.
- Emits an unencrypted event `ContractInstanceDeployed` with the address preimage.

The pseudocode for the process described above is the following:

```

fn deploy (
    salt: Field,
    contract_class_id: Field,
    initialization_hash: Field,
    public_keys_hash: Field,
    universal_deploy?: boolean,
)
    let contract_class_registerer: Contract =
ContractClassRegisterer::at(CONTRACT_CLASS_REGISTERER_ADDRESS);

    assert(nullifier_exists(silo(contract_class_id,
contract_class_registerer.address)));

    let deployer: Address = if universal_deploy { 0 } else { msg_sender };
    let version: Field = 1;

    let address = address_crh(
        version,
        salt,
        deployer,
        contract_class_id,
        initialization_hash,
        public_keys_hash
    );
    emit_nullifier(address);

    emit_unencrypted_event(ContractInstanceDeployed::new(address, version,
salt, contract_class_id, initialization_hash, public_keys_hash));

```

See [address](#) for `address_crh`.

Upon seeing a `ContractInstanceDeployed` event from the canonical `ContractInstanceDeployer` contract, nodes are expected to store the address and preimage, so they can verify executed code during public code execution as described in the next section.

The `ContractInstanceDeployer` contract provides two implementations of the `deploy` function: a private and a public one.

Genesis

The `ContractInstanceDeployer` will need to exist from the genesis of the Aztec Network, otherwise nothing will ever be deployable to the network. The Class Nullifier for the `ContractInstanceDeployer` contract will be pre-inserted into the genesis nullifier tree at leaf index

`GENESIS_NULLIFIER_LEAF_INDEX_OF_CONTRACT_INSTANCE_DEPLOYER_CLASS_ID_NULLIFIER`.

The canonical instance will be deployed at `CONTRACT_INSTANCE_DEPLOYER_ADDRESS`, and its Deployment Nullifier will be inserted at

`GENESIS_NULLIFIER_LEAF_INDEX_OF_CONTRACT_INSTANCE_DEPLOYER_DEPLOYMENT_NULLIFIER`.

Verification of Executed Code

The Kernel Circuit, both private and public, is responsible for verifying that the code loaded for a given function execution matches the expected one. This requires the following checks:

- The `contract_class_id` of the address called is the expected one, verified by hashing the address preimage that includes the `contract_class_id`.
- The `function selector` being executed is part of the `contract_class_id`, verified via a Merkle membership proof of the selector in the functions tree of the Contract Class.

Specific to private functions:

- The hash of the `verification_key` matches the `vk_hash` defined in the corresponding [Private Function](#) for the Contract Class. Note that the `verification_key` must include an identifier of the proving system used to compute it.

Specific to public functions:

- The bytecode loaded by the [AVM](#) for the contract matches the `bytecode_commitment` in the contract class, verified using the [bytecode validation circuit](#).
- The contract Deployment Nullifier has been emitted, or prove that it hasn't, in which case the transaction is expected to revert. This check is done via a merkle (non-)membership proof of the Deployment Nullifier. Note that a public function should be callable in the same

transaction in which its contract Deployment Nullifier was emitted.

Note that, since constructors are handled at the application level, the kernel circuit is not required to check the Initialization Nullifier before executing code.

Verifying Brilliq in Private Functions

Private functions may have unconstrained code, inlined as Brilliq bytecode. While unconstrained code, as its name implies, is not constrained within the protocol, a user PXE still needs a mechanism to verify that the code it has been delivered off-chain for a given function is correct.

This verification is done via the [contract class](#) `artifact_hash`, which contains a commitment to all bytecode in the contract. The PXE should receive the entire contract artifact, or at least the relevant sections to execute along with the commitments for the others to reconstruct the original `artifact_hash`, and verify that the resulting `artifact_hash` matches the one declared on-chain for the class of the contract being run.

Discarded Approaches

Contracts Tree

Earlier versions of the protocol relied on a dedicated contract tree, which required dedicated kernel code to process deployments, which had to be enshrined as new outputs from the application circuits. By abstracting contract deployment and storing deployments as nullifiers, the interface between the application and kernel circuits is simplified, and the kernel circuit has far fewer responsibilities. Furthermore, multiple contract deployments within a single transaction are now possible.

Requiring initialization for Public Deployment

An earlier version of this draft required contracts to be Initialized in order to be Publicly Deployed. While this was useful for removing the initialization check in public functions, it caused a mix of concerns where the `ContractInstanceDeployer` needed to read a nullifier emitted from another contract. It also coupled the `ContractInstanceDeployer` to the convention decided for Initialization Nullifiers, and forced every contract to have a constructor in order to be publicly

deployed even if they didn't need one. Furthermore, it required public constructors to be called via the `ContractInstanceDeployer` only.

Fully separating Initialization and Public Deployment leads to a cleaner `ContractInstanceDeployer`, and allows more flexibility to applications in handling their own initialization. The main downsides are that this opens the door for a contract to be simultaneously Publicly Deployed and Uninitialized, which is a state that does not seem to map to a valid use case. And it requires public functions to check the Initialization Nullifier on every call, which in the current approach is not needed as the presence of the Deployment Nullifier checked by the Public Kernel is enough of a guarantee that the contract was initialized.

Execute Initialization during Public Deployment only

While it is appealing to allow a user to privately create a new contract instance and not reveal it to the world, we have not yet validated this use case. We could simplify deployment by relying on a single nullifier to track Initialization, and couple it with Public Deployment. Private functions can check initialization via the Deployment Nullifier emitted by the `ContractInstanceDeployer`.

This approach requires that constructors are only invoked as part of Public Deployment, so constructors would require an additional check for `msg_sender` being the canonical `ContractInstanceDeployer`. Furthermore, to ensure that an instance constructor is properly run, the `ContractInstanceDeployer` would need to know the selector for the instance constructor, which now needs to be part of the Contract Class, re-enshrining it into the protocol. Last, being able to keep agreements (contracts) private among their parties is commonplace in the traditional world, so there is a compelling argument for keeping this requirement.

Alternatively, we could remove constructor abstraction altogether, and have the Private Kernel Circuit check for the Deployment Nullifier, much like the Public Kernel Circuit does. However, this hurts Diversified and Stealth account contracts, which now require an explicit deployment and cannot be used directly.

Calls

Functions in the Aztec Network can call other functions. There are several types of call:

- **Synchronous calls:** when a private function calls another private function; or when a public function calls another public function.
- **Enqueued calls:** when a private function calls a public function.
- **Batched calls:** when multiple calls to the same function are enqueued and processed as a single call on a concatenation of the arguments.

The protocol also supports alternative call methods, such as [static](#), and [unconstrained](#) calls.

In addition to function calls, the protocol allows for communication via message-passing back-and-forth between L1 and L2, as well as from public to private functions.

Synchronous calls

Calls from a private function to another private function, as well as calls from a public function to another public function, are synchronous. When a synchronous function call...

Enqueued calls

Calls from private functions to public functions are asynchronous. Since private and public functions are executed in different domains at different times and in different cont...

Batched calls

The low-level specifics of how batched calls will work is still being discussed.

Static calls

Synchronous calls, both private and public, can be executed as static calls. This means that the called function, and all nested calls within, cannot emit any modifying side ef...

Unconstrained calls

<!--

Inter-Layer Calls

Public-Private messaging

Synchronous calls

Calls from a private function to another private function, as well as calls from a public function to another public function, are *synchronous*. When a synchronous function call is found during execution, execution jumps to the target of the call, and returns to the caller with a return value from the function called. This allows easy composability across contracts.

At the protocol level, each call is represented as a `CallStackItem`, which includes the contract address and function being called, as well as the public inputs `PrivateCircuitPublicInputs` or `PublicCircuitPublicInputs` that are outputted by the execution of the called function. These public inputs include information on the call context, the side effects of the execution, and the block header.

At the contract level, a call is executed via an oracle call `callPrivateFunction` or `callPublicFunction`, both of which accept the contract address to call, the function selector, and a hash of the arguments. The oracle call prompts the executor to pause the current frame, jump to the target of the call, and return its result. The result is a `CallStackItem` that represents the nested execution.

The calling function is responsible for asserting that the function and arguments in the returned `CallStackItem` match the requested ones, otherwise a malicious oracle could return a `CallStackItem` for a different execution. The calling function must also push the hash of the returned `CallStackItem` into the private or public call stack of the current execution context, which is returned as part of the circuit's `PublicInputs` output. The end result is a top-level entrypoint `CallStackItem`, which itself contains (nested within) a stack of call stack items to process.

The kernel circuit is then responsible for iteratively processing each `CallStackItem`, pushing new items into the stack as it encounters nested calls, and popping one item

off the stack with each kernel iteration until the stack is empty. The private kernel circuit processes private function calls locally in the PXE, whereas the public kernel circuit processes public function calls on the sequencer's machine.

The private kernel circuit iterations begin with the entrypoint execution, empty output and proof. The public kernel circuit starts with the public call stack in the transaction object , and builds on top of the output and proof of the private kernel circuit.

```
let call_stack, kernel_public_inputs, proof
if is_private():
    call_stack = [top_level_execution]
    kernel_public_inputs = empty_inputs
    proof = empty_proof
else:
    call_stack = tx.public_call_stack
    kernel_public_inputs = tx.kernel_public_inputs
    proof = tx.proof

while call_stack is not empty:
    let call_stack_item = call_stack.pop()
    call_stack.push(...call_stack_item.call_stack)
    kernel_public_inputs, proof = kernel_circuit(call_stack_item,
kernel_public_inputs, proof)
```

The kernel circuit asserts that nested functions and their side effects are processed in order, and that the hash of each nested execution matches the corresponding hash outputted in the call stack by each `CircuitPublicInputs`.

For more information about how the private kernel circuit works, see [here](#).

Enqueued calls

Calls from private functions to public functions are asynchronous. Since private and public functions are executed in different domains at different times and in different contexts -- the former are run locally by the user in a PXE and the latter by the sequencer -- it is not possible for a private function to call a public one and await its result. Instead, private functions can *enqueue* public function calls.

The process is analogous to [synchronous calls](#), but relies on an `enqueuePublicFunctionCall` oracle call that accepts the same arguments. The object returned by the oracle is a `PublicCallStackItem` with a flag `is_execution_request` set, and empty side effects to reflect that the stack item has not been executed yet. As with synchronous calls, the caller is responsible for validating the function and arguments in the call stack item, and to push its hash to its public call stack, which represents the list of enqueued public function calls.

Once the transaction is received by the sequencer, the public kernel circuit can begin processing the enqueued public function calls from the transaction's public call stack, pushing new recursive calls to the stack as needed, and popping-off one call stack item at a time, until the public call stack is empty, as described in the [synchronous calls](#) section.

Batched calls

WARNING

The low-level specifics of how batched calls will work is still being discussed.

Calls to private functions can be *batched* instead of executed [synchronously](#). When executing a batched call to a private function, the function is not executed on the spot, but enqueued for execution at the end of local execution. Once the private call stack has been emptied, all batched execution requests are grouped by target (contract and function selector), and executed via a single call to each target.

Batched calls are implemented by pushing a [PrivateCallStackItem](#) with the flag `is_execution_request` into a [private_batched_queue](#) in the execution context, and require an oracle call to a [batchPrivateFunctionCall](#) function with the same argument types as for other oracle function calls.

Batched calls are processed by the private kernel circuit. On each kernel circuit iteration, if the private call stack is not empty, the kernel circuit pops and processes the topmost entry. Otherwise, if the batched queue is not empty, the kernel pops the first item, collects and deletes all other items with the same target, and calls into the target. Note that this allows batched calls to trigger further synchronous calls.

The arguments for the batched call are arranged in an array with one position for each individual call. Each position within the array is a nested array where the first element is the call context for that individual call, followed by the actual arguments of the call. A batched call is expected to return an array of [PrivateCircuitPublicInputs](#), where each public input's call context matches the call context from the corresponding individual call. This allows batched delegate calls, where each individual call processed has a context of its own. This can be used to emit logs on behalf of multiple contracts within a single batched call.

In pseudocode, the kernel circuit executes the following logic:

```
loop:  
    if next_call_stack_item = context.private_call_stack.pop():  
        execute(next_call_stack_item.address,  
next_call_stack_item.function_selector,  
next_call_stack_item.arguments)  
    else if next_batched_call = context.private_batched_queue.pop():  
        let calls = context.private_batched_queue.filter(call =>  
call.target == target)  
        context.private_batched_queue.delete_many(calls)  
        execute(target.address, target.function_selector,  
calls.map(call => [call.call_context, ...call.arguments]))  
    else:  
        break
```

The rationale for batched calls is to minimize the number of function calls in private execution, in order to reduce total proving times. Batched calls are mostly intended for usage with note delivery precompiles, since these do not require synchronous execution, and allows for processing all notes that are to be encrypted and tagged with the same mechanism using a single call. Batched calls can also be used for other common functions which do not require synchronous execution and which are likely to be invoked multiple times.

Static calls

Synchronous calls, both private and public, can be executed as *static* calls. This means that the called function, and all nested calls within, cannot emit any modifying side effects, such as creating or consuming notes, writing to storage, or emitting events. The purpose of a static call is to query another contract while ensuring that the call will not modify state. Static calls are based on [EIP214](#).

In particular, the following fields of the returned `CallStackItem` must be zero or empty in a static call:

- `new_note_hashes`
- `new_nullifiers`
- `nullified_commitments`
- `new_12_to_11_msgs`
- `encrypted_logs_hash`
- `unencrypted_logs_hash`
- `encrypted_log_preimages_length`
- `unencrypted_log_preimages_length`

From the moment a static call is made, every subsequent nested call is forced to be static by setting a flag in the derived `CallContext`, which propagates through the call stack.

At the protocol level, a static call is identified by a `is_static_call` flag in the `CircuitPublicInputs` of the `CallStackItem`. The kernel is responsible for asserting that the call and all nested calls do not emit any forbidden side effects.

At the contract level, a caller can initiate a static call via a `staticCallPrivateFunction` or `staticCallPublicFunction` oracle call. The

caller is responsible for asserting that the returned `CallStackItem` has the `is_static_call` flag correctly set.

Unconstrained calls

Private function calls can be executed as *unconstrained*. Unconstrained function calls execute the code at the target and return the result, but their execution is not constrained. It is responsibility of the caller to constrain the result, if needed. Unconstrained calls are a generalization of oracle function calls, where the call is not to a PXE function but to another contract. Side effects from unconstrained calls are ignored. Note that all calls executed from an unconstrained call frame will be unconstrained as well.

Unconstrained calls are executed via a `unconstrainedCallPrivateFunction` oracle call, which accepts the same arguments as a regular `callPrivateFunction`, and return the result from the function call. Unconstrained calls are not pushed into the `private_call_stack` and do not incur in an additional kernel iteration.

The rationale for unconstrained calls is to allow apps to consume results from functions that do not need to be provable. An example use case for unconstrained calls is unconstrained encryption and note tagging, which can be used in applications where constraining such encryption computations isn't necessary, e.g. if the sender is incentivized to ensure the recipient receives the correct data.

Another motivation for unconstrained calls is for retrieving or computing data where the end result can be more efficiently constrained by the caller.

Inter-Layer Calls

Public-Private messaging

Public state and private state exist in different [trees](#). In a private function you cannot reference or modify public state.

Yet, it should be possible for:

1. private functions to call private or public functions
2. public functions to call private or public functions

Private functions are executed locally by the user, so that the user can ensure privacy of their data. Public functions are executed by the sequencer, who is the only party with an up-to-date view of the latest public state. It's natural, then, that private functions be executed first, and public functions be executed after the user has submitted a [transaction object](#) (which contains proof of private execution) to the network. Since a user doesn't have an up-to-date view of the latest state, private functions are always executed on some historical snapshot of the network's state.

Given this natural flow from private-land to public-land, private functions can enqueue calls to public functions. But the opposite direction is not true. We'll see [below](#) that public functions cannot "call" private functions, but rather they must pass messages.

Since private functions execute first, they cannot 'wait' on the results of any of their calls to public functions.

By way of example, suppose a function makes a call to a public function, and then to a private function. The public function will not be executed immediately, but will instead be enqueued for the sequencer to execute later.

The order of execution will actually be:

And the order of proving will actually be:

Private to Public Messaging

When a private function calls a public function:

1. The arguments to the public function are hashed into an `args_hash`.
2. A `public_call_stack_item` is created, which includes the public function's `function_selector`, `contract_address`, and `args_hash`.
3. A hash of the `public_call_stack_item` gets enqueued into a separate `public_call_stack` and passed as inputs to the private kernel.
4. The private kernel pushes these hashes onto its own the `public_inputs`, which the sequencer can see.
5. The PXE creates a `transaction_object` which includes the kernel's `public_inputs`.
6. The PXE sends the `transaction_object` to the sequencer.
7. Sequencer then unpacks the `public_call_stack_item` and executes each of the functions.
8. The Public VM executes the enqueued public calls, and then verifies that the hash provided by the private kernel matches the current call stack item.

Handling Privacy Leakage and `msg.sender`

The sequencer only sees the data in the `transaction_object`, which shouldn't expose any private information. There are some practical caveats.

When making a private-to-public call, the `msg.sender` will become public. If this is

the actual user, then it leaks privacy. If `msg.sender` is some application's contract address, this leaks which contract is calling the public method and therefore leaks which contract the user was interacting with in private land.

An out-of-protocol option to randomizing `msg.sender` (as a user) would be to deploy a [diversified account contract](#) and route transactions through this contract.

Application developers might also be able to do something similar, to randomize the `msg.sender` of their app contract's address.

Reverts

If the private part of a transaction reverts, then public calls are never enqueued. But if the public part of the transaction reverts, it should still revert the entire transaction. I.e. the sequencer should drop the execution results of the private part of the transaction and not include those in the state transitioner smart contract. A fee can still be charged by the sequencer for their compute effort.

Public to Private Messaging

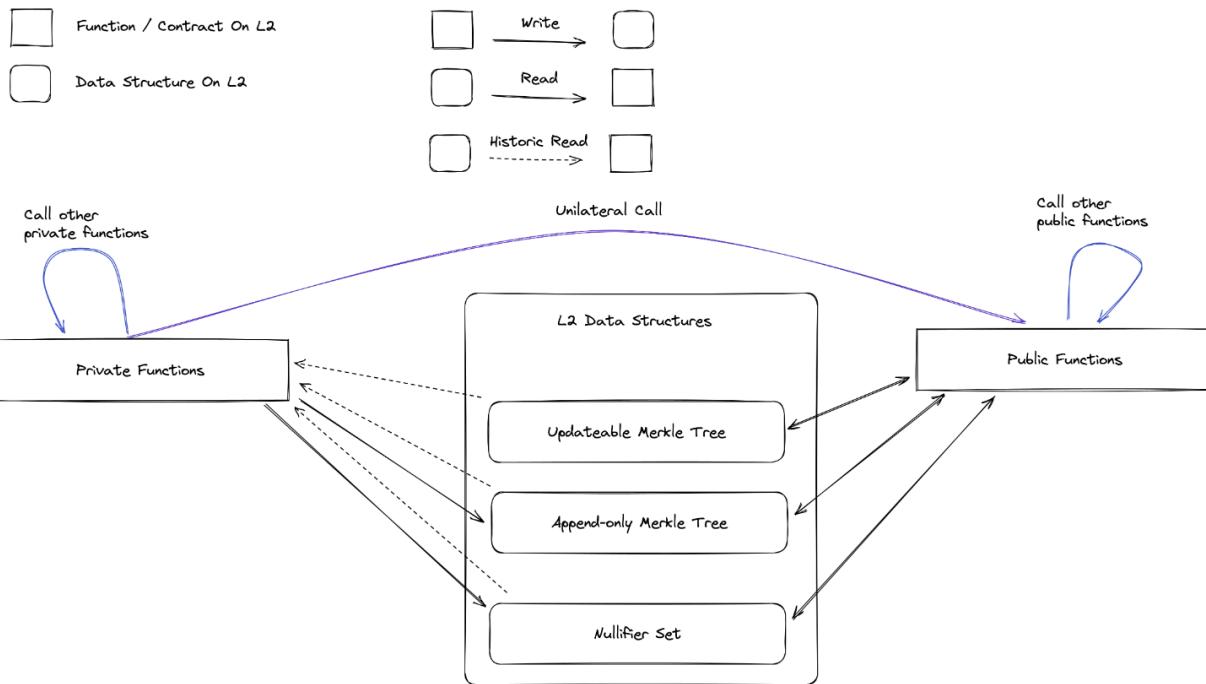
Since public functions execute after private functions, it isn't possible for a public function to call a private function in the same transaction. Nevertheless, it is quite useful for public functions to have a message passing system to private land. A public function can add messages to the [Note Hash Tree](#) to save messages from a public function call, that can later be consumed by a private function. Note: such a message can only be consumed by a *later* transaction. In practice this means that unless you are the sequencer (or have an out of protocol agreement with the sequencer) it cannot be consumed within the same rollup.

To elaborate, a public function may not have read access to encrypted private state in the Note Hash Tree, but it can write to it. You could create a note in the public domain, compute its note hash which gets passed to the inputs of the public VM which adds

the hash to the note hash tree. The user who wants to redeem the note can add the note preimage to their PXE and then redeem/nullify the note in the private domain at a later time.

In the picture below, it is worth noting that all data reads performed by private functions are historical in nature, and that private functions are not capable of modifying public storage. Conversely, public functions have the capacity to manipulate private storage (e.g., inserting new note hashes, potentially as part of transferring funds from the public domain to the private domain).

Legend:



Cross-chain communication

This section describes what our L1 contracts do, what they are responsible for and how they interact with the circuits.

Note that the only reason that we even have any contracts is to facilitate cross-chain communication. The contracts are not required for the rollup to function, but required to bridge assets and to reduce the cost of light nodes.

! PURPOSE OF CONTRACTS

The purpose of the L1 contracts are simple:

- Facilitate cross-chain communication such that L1 liquidity can be used on L2
- Act as a validating light node for L2 that every L1 node implicitly run

Overview

When presented with a new `ProvenBlock` and its proof, an Aztec node can be convinced of its validity if the proof passes and the `Header.last_archive` matches the `archive` of the node (archive here represents a root of `archive tree`). The `archive` used as public input is the archive after the new header is inserted (see `root rollup`).

```
def propose(block: ProvenBlock, proof: Proof):  
    header = block.header  
    block_number = header.global_variables.block_number
```

! WHY `math.ceil(log2(MAX_L2_T0_L1_MSGS_PER_TX))`?

The argument to the `insert` function is the `outbox` is the height of the message tree. Since every transaction can hold more than 1 message, it might add multiple layers to the tree. For a binary tree, the number of extra layers to add is computed as `math.ceil(log2(MAX_L2_T0_L1_MSGS_PER_TX))`. Currently, `MAX_L2_T0_L1_MSGS_PER_TX = 2` which means that we are simply adding 1 extra layer.

While the `ProvenBlock` must be published and available for nodes to build the state of the rollup, we can build the validating light node (the contract) such that as long as the node can be *convinced* that the data is available we can progress the state. This means our light node can be built to only require a subset of the `ProvenBlock` to be published to Ethereum L1 as calldata and use a different data availability layer for most of the block body. Namely, we need the cross-chain messages to be published to L1, but the rest of the block body can be published to a different data availability layer.

! VALIDIUM OR ROLLUP

If a different data availability layer than Ethereum is used for the block body, we are effectively building a Validium. If we use Ethereum for the block body, we are building a Rollup.

For more information around the requirements we have for the availability, see [Data Availability](#).

Using the data structures defined throughout the [rollup circuits](#) section, we can outline the validating light node structure as follows:

State transitioner

The state transitioner is the heart of the validating light node for the L2. The contract keeps track of the current state of the L2 and progresses this state when a valid L2 block is received. It also facilitates cross-chain communication (communication between the L1 inbox and outbox contracts).

 INFO

The following example shows a simplified case where proof and block are provided in the same transaction.

```
class StateTransitioner:

    struct BlockLog:
        archive: bytes32
        slot_number: uint128

    VERIFIER: immutable(IVerifier)
    INBOX: immutable(IInbox)
    OUTBOX: immutable(Outbox)
    VERSION: immutable(uint256)
    GENESIS_TIME: immutable(uint256)
    SLOT_DURATION: immutable(uint256)

    blocks: BlockLog[]

    def __init__(self, ...):
        ...
        Initialize the state transitioner
        ...
        self.blocks.append(BlockLog({archive: bytes32(0),
slot_number: 0}))
```

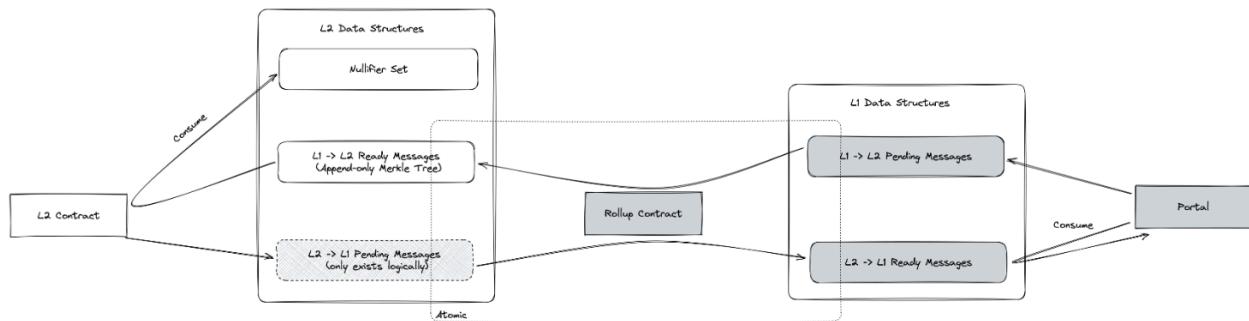
Registry

To keep one location where all the core rollup contracts can be found, we have a registry contract. The registry is a contract that holds the current and historical addresses of the core rollup contracts. The addresses of a rollup deployment are contained in a snapshot, and the registry is tracking version-snapshot pairs.

Depending on the upgrade scheme, it might be used to handle upgrades, or it could entirely be removed. It is generally the one address that a node MUST know about, as it can then tell the node where to find the remainder of the contracts. This is for example used when looking for the address new L2 blocks should be published to.

Message Bridges

To let users communicate between L1 and the L2, we are using message bridges, namely an L1 inbox that is paired to an L2 outbox, and an L2 inbox that is paired to an L1 outbox.



NAMING IS BASED FROM THE POV OF THE STATE TRANSITIONER.

While we logically have 4 boxes, we practically only require 3 of those. The L2 inbox is not real - but only logical. This is due to the fact that they are always inserted and then consumed in the same block! Insertions require a L2 transaction, and it is then to be

consumed and moved to the L1 outbox by the state transitioner in the same block.

Portals

Some contracts on L2 might wish to talk to contracts on L1 - these recipients on L1 are called portals.

Often it is desired to constrain where messages are sent to and received from, which can be done by keeping the portal address in the storage of the L2 contract, such that it can be loaded on demand.

Messages

Messages that are communicated between the L1 and L2 need to contain a minimum of information to ensure that they can correctly consumed by users. Specifically the messages should be as described below:

```
struct L1Actor {
    address: actor,
    uint256: chainId,
}

struct L2Actor {
    bytes32: actor,
    uint256: version,
}

struct L1ToL2Msg {
    L1Actor: sender,
    L2Actor: recipient,
    bytes32: content,
    bytes32: secretHash,
}
```

Beware, that while we speak of messages, we are practically passing around only their **hashes** to reduce cost. The `version` value of the `L2Actor` is the version of the rollup, which is intended to be used to specify which version of the rollup the message is intended for or sent from. This way, multiple rollup instances can use the same inbox/outbox contracts.

❗ WHY A SINGLE HASH?

Compute on L1 is expensive, but storage is extremely expensive! To reduce overhead, we trade storage for computation and only commit to the messages and then "open" these for consumption later. However, since computation also bears significant cost we need to use a hash function that is relatively cheap on L1, while still being doable inside a snark. For this purpose a modified SHA256 was chosen, modified by fitting the output value into a single field element using the modulo operator.

Some additional discussion/comments on the message structure can be found in the forum post, [The Republic](#).

Since any data that is moving from one chain to the other at some point will live on L1, it will be public. While this is fine for L1 consumption (which is always public), we want to ensure that the L2 consumption can be private. To support this, we use a nullifier scheme similar to what we are doing for the other [notes](#). As part of the nullifier computation we use a `secret` which hashes to a `secretHash`, which ensures that only actors with knowledge of the `secret` will be able to see when it is spent on L2.

Any message that is consumed on one side MUST be moved to the other side. This is to ensure that the messages exist AND are only consumed once. The L1 contracts handle one side and the circuits must handle the other.

❗ IS `secretHash` REQUIRED?

We are using the `secretHash` to ensure that the user can spend the message privately with a nullifier computation. However, as the nullifier computation is almost entirely controlled by the Aztec contract (the application circuit, except the contract siloing - see [Nullifier Tree](#)). Contracts could compute a custom nullifier to have the `secretHash` included as part of the computation. However, the chosen approach reduces the developer burden and reduces the likelihood of mistakes.

Inbox

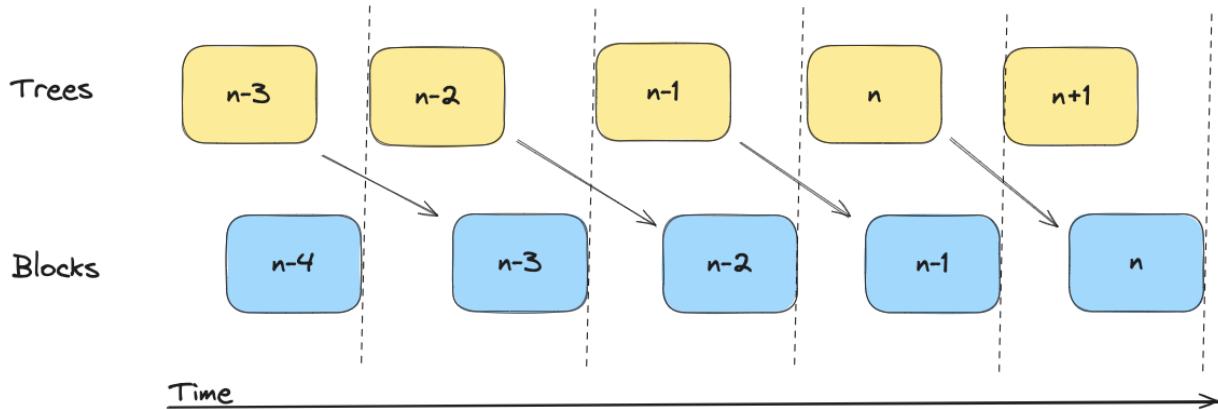
When we say inbox, we are generally referring to the L1 contract that handles the L1 to L2 messages.

The inbox takes messages from L1 contracts and inserts them into a series of message trees. We build multiple "trees" instead of a single tree, since we are building one tree per block and not one large tree with all messages for all blocks.

We need to split trees into epochs such that a sequencer can build a proof based on a tree that is not going to update in the middle of the proof building. Having one tree that updates across blocks would allow DOS attacks on the sequencer, which is undesirable.

In practice, we introduce a "lag" between when trees are built and when they must be included. Whenever a new block is published, we start building a new tree, essentially meaning that at block n we include tree n which was created earlier (during block $n - 1$).

Below, tree n is "fixed" when block n needs to be published. Tree $n + 1$ is being built upon until block n is published.



When the state transitioner processes a tree, it **MUST** insert the subtree into the "L2 outbox" ([message tree](#) included in global state).

When a message is inserted into the inbox, the inbox **MUST** fill in the `sender`:

- `L1Actor.actor`: The sender of the message (the caller), `msg.sender`
- `L1Actor.chainId`: The chainId of the L1 chain sending the message, `block.chainId`

We **MUST** populate these values in the inbox, since we cannot rely on user input.

From the `L1ToL2Msg` we compute a hash of the message. This hash is what is moved by the state transitioner to the L2 outbox.

Since message from L1 to L2 can be inserted independently of the L2 block, the message transfer (moving from L1 inbox into L2 outbox) is not synchronous as it is for L2 to L1 messages. This means that the message can be inserted into the inbox, but not yet moved to the outbox. The message will be moved to the outbox when the state transitioner processes the message as part of a block. Since sequencers are required to move the entire subtree at once, you can be sure that the message will be moved to the outbox. As mentioned earlier, segmenting updates is done to ensure that the messages are not used to DOS the state transitioner.

The message tree is built on L1, so we need to use a gas-friendly hash-function such as SHA256. However, we need to allow users to prove inclusion in this tree, so we cannot just insert the SHA256 tree into the rollup state, since it expensive to process in a zk circuit. Therefore, we need to "convert" the SHA256 tree into a tree that uses a more snark-friendly hash. This part is done in the [tree parity circuits](#).

Furthermore, to build the tree on L1, we can optimize storage on L1 such that the insertions don't require a lot of merkle tree related data which could be cumbersome and prone to race-conditions (e.g., two insertions based on inclusion paths that are created at the same time will invalidate each other).

The solution is to use a "frontier" merkle tree to store the messages. This is a special kind of append-only merkle tree that allows us to store very few elements in storage, while still being able to extend it and compute the root of the tree. See the [Frontier Merkle Tree](#) for more information on this.

Assuming that we have these trees, we can build an [Inbox](#) as follows. When a new block is published, we start building a new tree. Notice however, that if we have entirely filled the current tree, we can start building a new one immediately, and the blocks can then "catch up".

```
class Inbox:
    STATE_TRANSITIONER: immutable(address)
    ZERO: immutable(bytes32)

    HEIGHT: immutable(uint256)
    SIZE: immutable(uint256)

    trees: HashMap[uint256, FrontierTree]

    to_include: uint256 = 0
    in_progress: uint256 = 1
```

L2 Inbox

While the L2 inbox is not a contract, it is a logical concept that apply mutations to the data similar to the L1 inbox to ensure that the sender cannot fake his position. This logic is handled by the kernel and rollup circuits.

Just like the L1 variant, we must populate the `sender` :

- `L2Actor.actor`: The sender of the message (the caller)
- `L2Actor.version`: The version of the L2 chain sending the message

In practice, this is done in the kernel circuit of the L2, and the message hashes are then aggregated into a tree as outlined in the [Rollup Circuits section](#) before it is inserted into the L1 outbox.

Outbox

The outboxes are the location where a user can consume messages from on the destination chain. An outbox can only contain elements that have previously been removed from the paired inbox.

Our L1 outbox is pretty simple, like the L1 inbox, it is a series of trees. The trees are built from the messages of all the transactions in the block, and the root and height is then pushed to the L1 outbox.

Whenever a portal wishes to consume a message, it proves that it is included in one of these roots and that it has not been consumed before.

To address the nullifier (marking it is spent), we can simply use a bitmap and flip just 1 bit per message. This shares some of the cost of processing messages.

This structure is used in many merkle airdrop contracts. Nevertheless, it requires

some consideration from the developers side, as the portal needs to prepare the inclusion proof for the message before it can be consumed. The proof can be prepared based on the published data, so with good libraries it should be very straight forward for most cases.

🔥 CHECKING SENDER

When consuming a message on L1, the portal contract must check that it was sent from the expected contract given that it is possible for multiple contracts on L2 to send to it. If the check is not done this could go horribly wrong.

```
class Outbox:
    STATE_TRANSITIONER: immutable(address)

    struct RootData:
        root: bytes32
        height: uint256
        nullified: HashMap[uint256, bool]

    roots: HashMap[uint256, RootData]

    def __init__(self, _state_transitioner: address):
        self.STATE_TRANSITIONER = _state_transitioner

    def insert(index: uint256, root: bytes32, height: uint256):
        assert msg.sender == self.STATE_TRANSITIONER
        self.roots[index] = RootData(root, height, {})

    def consume(
        root_index: uint256,
        leaf_index: uint256,
        message: L2ToL1Message,
        inclusion_proof: bytes[]
    ):
        leaf = message.hash_to_field()
```

L2 Outbox

The L2 outbox is a merkle tree that is populated with the messages moved by the state transitioner through the converted tree as seen in [Rollup Circuits](#). The messages are consumed on L2 by emitting a nullifier from the application circuit (Aztec contract).

This means that all validation is done by the application circuit. The application should:

- Ensure that the message exists in the outbox (message tree)
- Ensure that the message sender is the expected contract
- Ensure that the message recipient is itself and that the version matches the expected version
- Ensure that the user knows `secret` that hashes to the `secretHash` of the message
- Compute a nullifier that includes the `secret` along with the message hash and the index of the message in the tree
 - The index is included to ensure that the nullifier is unique for each message

Validity conditions

While there are multiple contracts, they work in unison to ensure that the rollup is valid and that messages are correctly moved between the chains. In practice this means that the contracts ensure that the following constraints are met in order for the validating light node to accept a block.

Note that some conditions are marked as SHOULD, which is not strictly needed for security of the rollup, but need for the security of the individual applications or for UX. Some of the conditions are repetitions of what we saw earlier from the [state](#)

transitioner.

- **Data Availability:** The block content MUST be available. To validate this, the `AvailabilityOracle` is used.
- **Header Validation:** See the checks from the [state transitioner](#)
- **Proof validation:** The proof MUST be validated with the header and archive.
- **Inserting messages:** for messages that are inserted into the inboxes:
 - The `sender.actor` MUST be the caller
 - The `(sender | recipient).chainId` MUST be the chainId of the L1 where the state transitioner is deployed
 - The `(sender | recipient).version` MUST be the version of the state transitioner (the version of the L2 specified in the L1 contract)
 - The `content` MUST fit within a field element
 - For L1 to L2 messages:
 - The `secretHash` MUST fit in a field element
- **Moving tree roots:**
 - Moves MUST be atomic:
 - Any message that is inserted into an outbox MUST be consumed from the matching inbox
 - Any message that is consumed from an inbox MUST be inserted into the matching outbox
- **Consuming messages:** for messages that are consumed from the outboxes:
 - L2 to L1 messages (on L1):
 - The consumer (caller) MUST match the `recipient.actor`
 - The consumer chainid MUST match the `recipient.chainId`
 - The consumer SHOULD check the `sender`
 - L1 to L2 messages (on L2):
 - The consumer contract SHOULD check the `sender` details against the `portal` contract

- The consumer contract SHOULD check that the `secret` is known to the caller
- The consumer contract SHOULD check the `recipient` details against its own details
- The consumer contract SHOULD emit a nullifier to preventing double-spending
- The consumer contract SHOULD check that the message exists in the state

! INFO

- For cost purposes, it can be useful to commit to the public inputs to just pass a single value into the circuit.
- Time constraints might change depending on the exact sequencer selection mechanism.

Logical Execution

Below, we will outline the LOGICAL execution of a L2 block and how the contracts interact with the circuits. We will be executing cross-chain communication before and after the block itself. Note that the L2 inbox only exists conceptually and its functionality is handled by the kernel and the rollup circuits.

We will walk briefly through the steps of the diagram above. The numbering matches the numbering of nodes in the diagram, the start of the action.

1. A portal contract on L1 wants to send a message for L2
2. The L1 inbox populates the message with information of the `sender` (using `msg.sender` and `block.chainid`)
3. The L1 inbox contract inserts the message into its tree

4. On the L2, as part of a L2 block, a transaction consumes a message from the L2 outbox
5. The L2 outbox ensures that the message is included, and that the caller is the recipient and knows the secret to spend. (This is done by the application circuit)
6. The nullifier of the message is emitted to privately spend the message (This is done by the application circuit)
7. The L2 contract sends a message to L1 (specifying a recipient)
8. The L2 inbox populates the message with `sender` information
9. The L2 inbox inserts the message into its storage
10. The rollup circuit starts consuming the messages from the inbox
11. The L2 inbox deletes the messages from its storage
12. The L2 block includes messages from the L1 inbox that are to be inserted into the L2 outbox
13. The L2 outbox state is updated to include the messages
14. The L2 block is submitted to L1
15. The state transitioner receives the block and verifies the proof + validates constraints on block
16. The state transitioner updates it's state to the ending state of the block
17. The state transitioner consumes the messages from the L1 inbox that was specified in the block. They have been inserted into the L2 outbox, ensuring atomicity.
18. The L1 inbox updates it local state by marking the message tree messages as consumed
19. The state transitioner inserts the messages tree root into the L1 Outbox. They have been consumed from the L2 inbox, ensuring atomicity.
20. The L1 outbox updates it local state by inserting the message root and height
21. The portal later consumes a message from the L1 outbox
22. The L1 outbox validates that the message exists and that the caller is indeed the recipient

23. The L1 outbox updates its local state by nullifying the message

⚠️ L2 INBOX IS NOT REAL

The L2 inbox doesn't need to exist independently because it keeps no state between blocks. Every message created on L2 in a block will be consumed and added to the L1 outbox in the same block.

Future work

- Sequencer selection contract(s)
 - Relies on the sequencer selection scheme being more explicitly defined
 - Relies on being able to validate the sequencer selection scheme
- Governance/upgrade contract(s)
 - Relies on the governance/upgrade scheme being more explicitly defined
- Forced transaction inclusion
 - While we don't have an exact scheme, an outline was made in [hackmd](#) and the [forum](#)

Frontier Merkle Tree

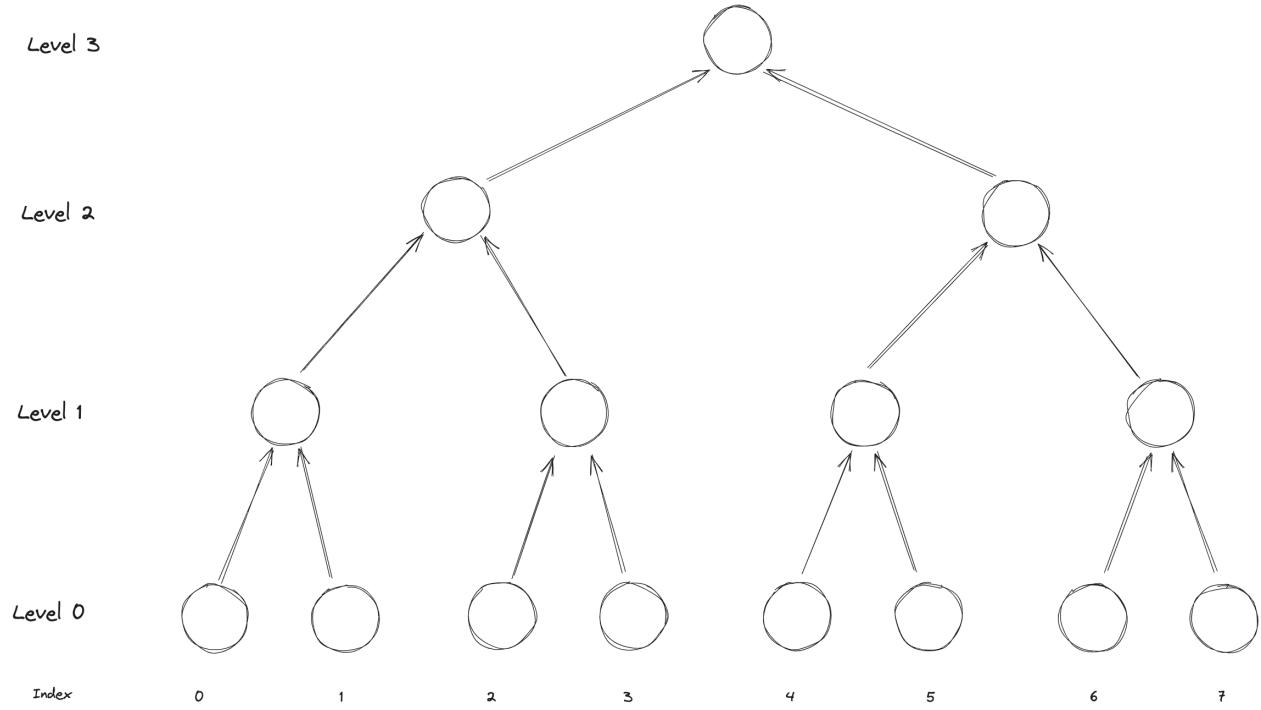
The Frontier Merkle Tree is an append only Merkle tree that is optimized for minimal storage on chain. By storing only the right-most non-empty node at each level of the tree we can always extend the tree with a new leaf or compute the root without needing to store the entire tree. We call these values the frontier of the tree. If we have the next index to insert at and the current frontier, we have everything needed to extend the tree or compute the root, with much less storage than a full merkle tree. Note that we're not actually keeping track of the data in the tree: we only store what's minimally required in order to be able to compute the root after inserting a new element.

We will go through a few diagrams and explanations to understand how this works. And then a pseudo implementation is provided.

Insertion

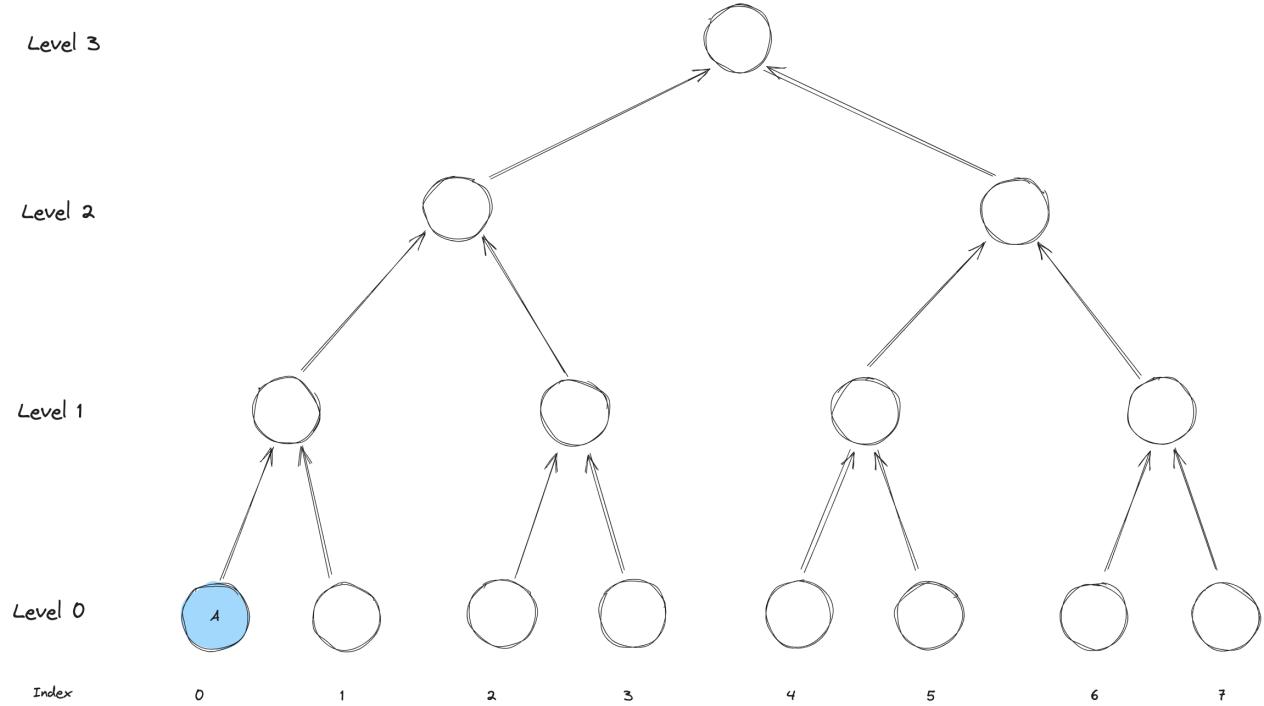
Whenever we are inserting, we need to update the "root" of the largest subtree possible. This is done by updating the node at the level of the tree, where we have just inserted its right-most descendant. This can sound a bit confusing, so we will go through a few examples.

At first, say that we have the following tree, and that it is currently entirely empty.



The first leaf

When we are inserting the first leaf (lets call it A), the largest subtree is that leaf value itself (level 0). In this case, we simply need to store the leaf value in `frontier[0]` and then we are done. For the sake of visualization, we will be drawing the elements in the `frontier` in blue.

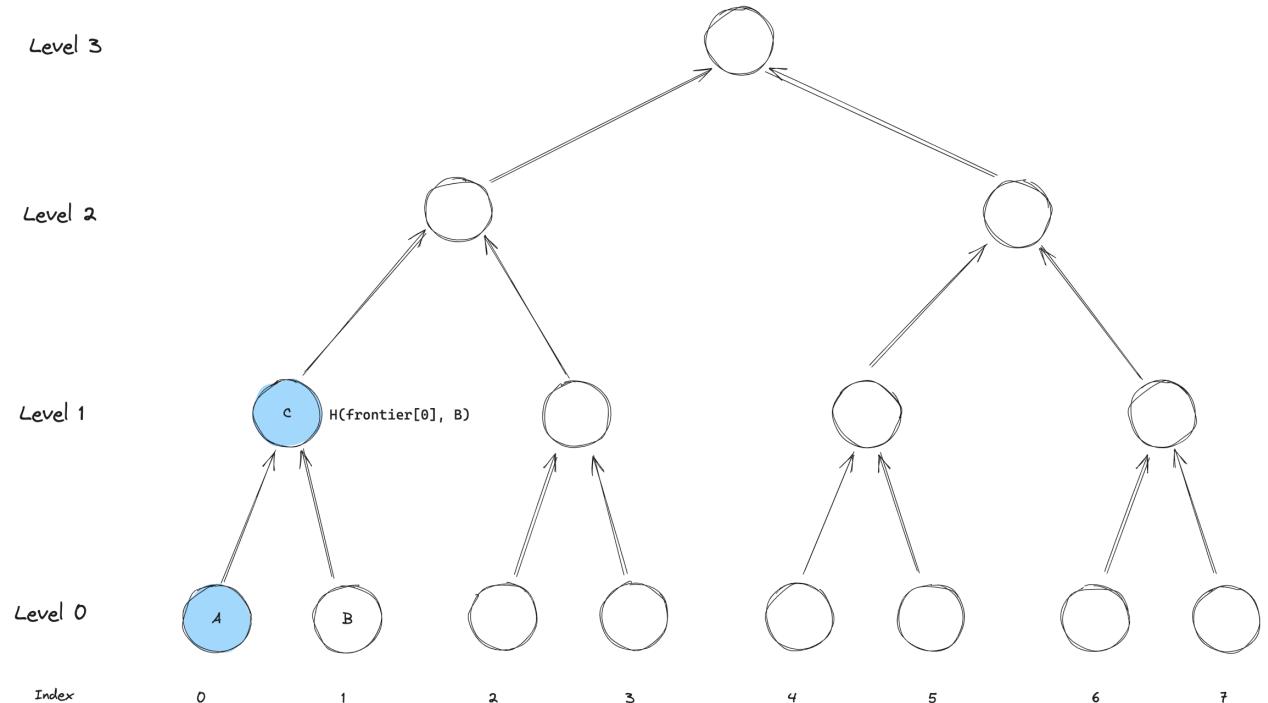


Notice that this will be the case whenever we are inserting a leaf at an even index.

The second leaf

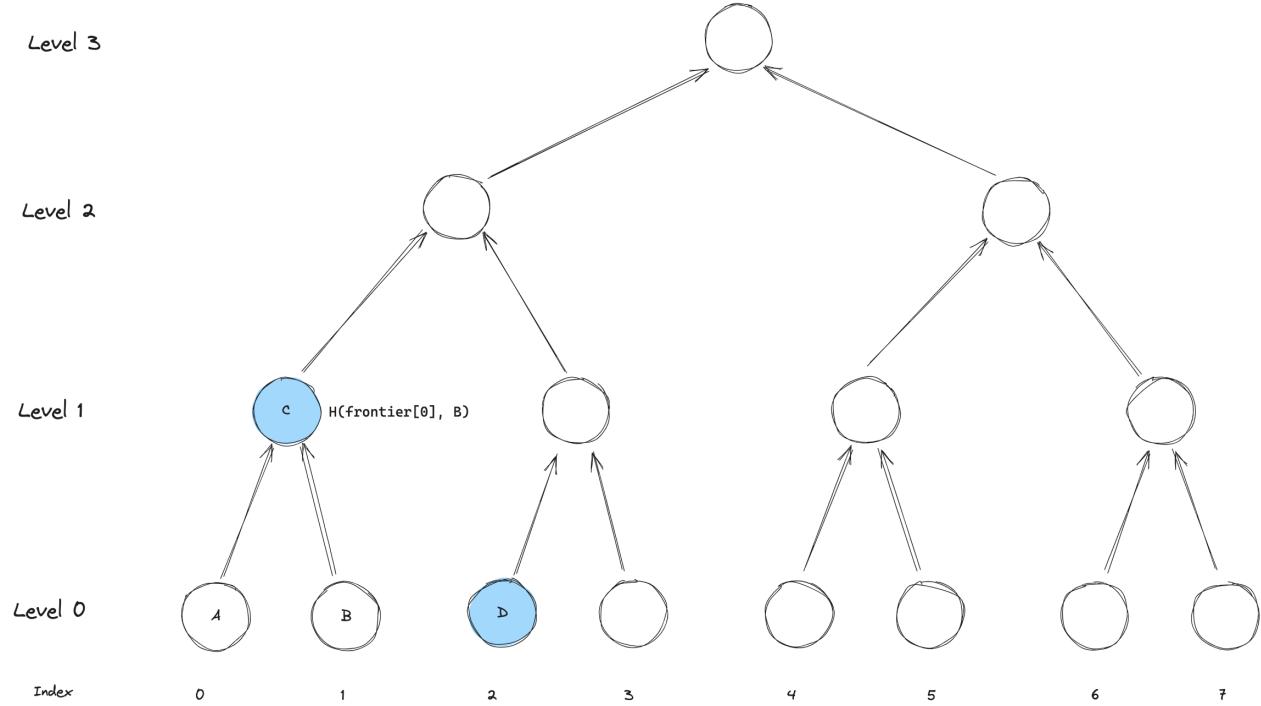
When we are inserting the second leaf (lets call it B), the largest subtree will no longer be at level 0. Instead it will be level 1, since the entire tree below it is now filled! Therefore, we will compute the root of this subtree, `H(frontier[0], B)` and store it in `frontier[1]`.

Notice, that we don't need to store the leaf B itself, since we won't be needing it for any future computations. This is what makes the frontier tree efficient - we get away with storing very little data.



Third leaf

When inserting the third leaf, we are again back to the largest subtree being filled by the insertion being itself at level 0. The update will look similar to the first, where we only update `frontier[0]` with the new leaf.

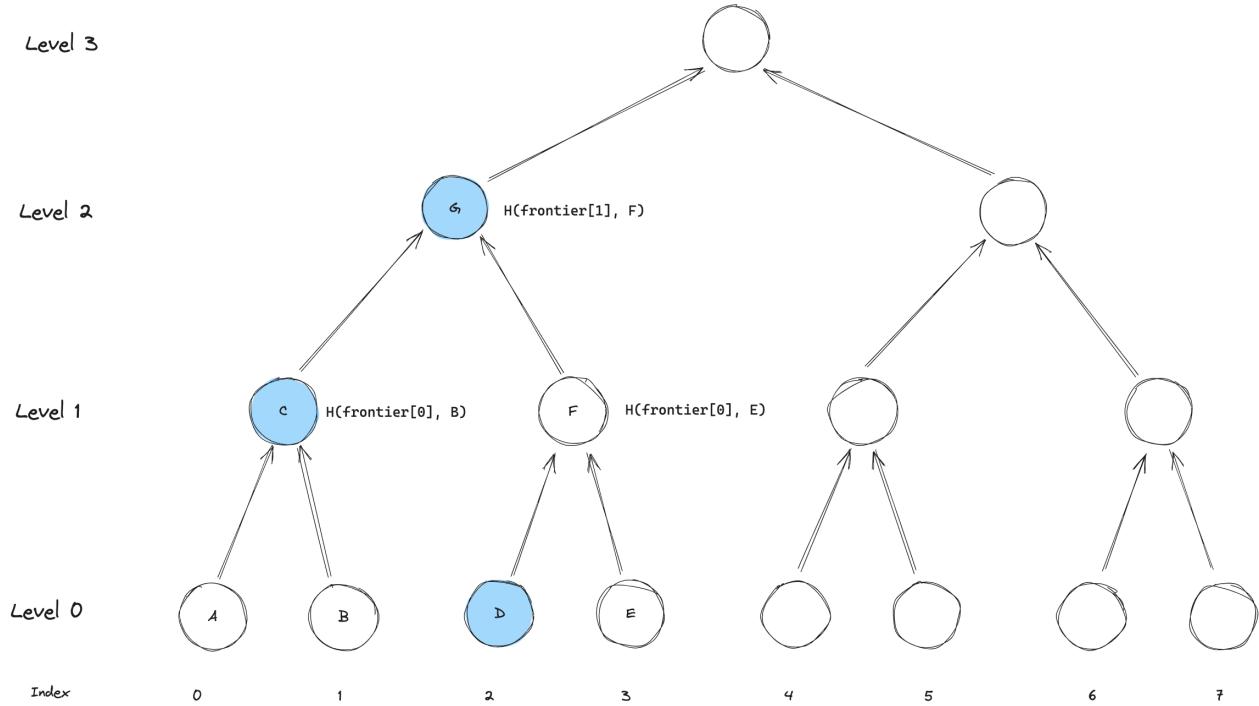


Fourth leaf

When inserting the fourth leaf, things get a bit more interesting. Now the largest subtree getting filled by the insertion is at level 2.

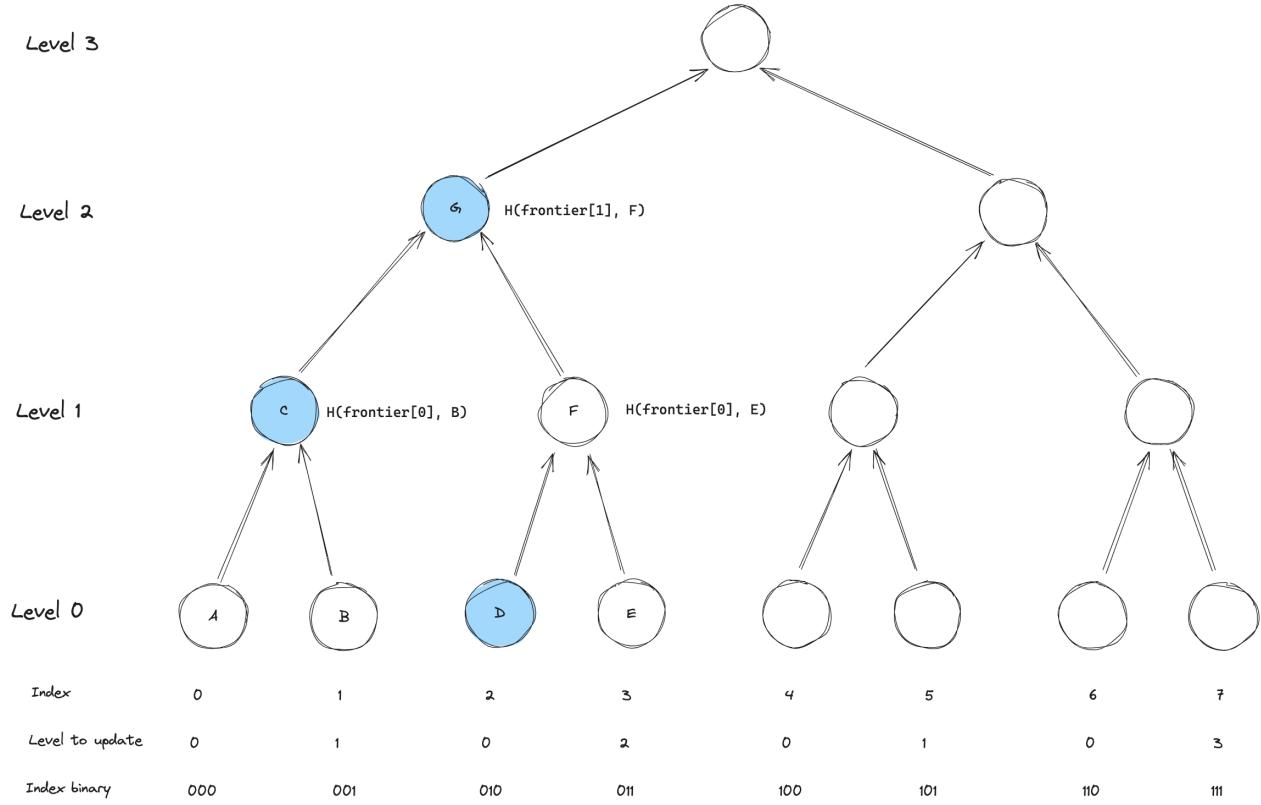
To compute the new subtree root, we have to compute `F = H(frontier[0], E)` and then `G = H(frontier[1], F)`.
`G` is then stored in `frontier[2]`.

As before, notice that we are only updating one value in the frontier.



Figuring out what to update

To figure out which level to update in the frontier, we simply need to figure out what the height is of the largest subtree that is filled by the insertion. While this might sound complex, it is actually quite simple. Consider the following extension of the diagram. We have added the level to update, along with the index of the leaf in binary. Seeing any pattern?



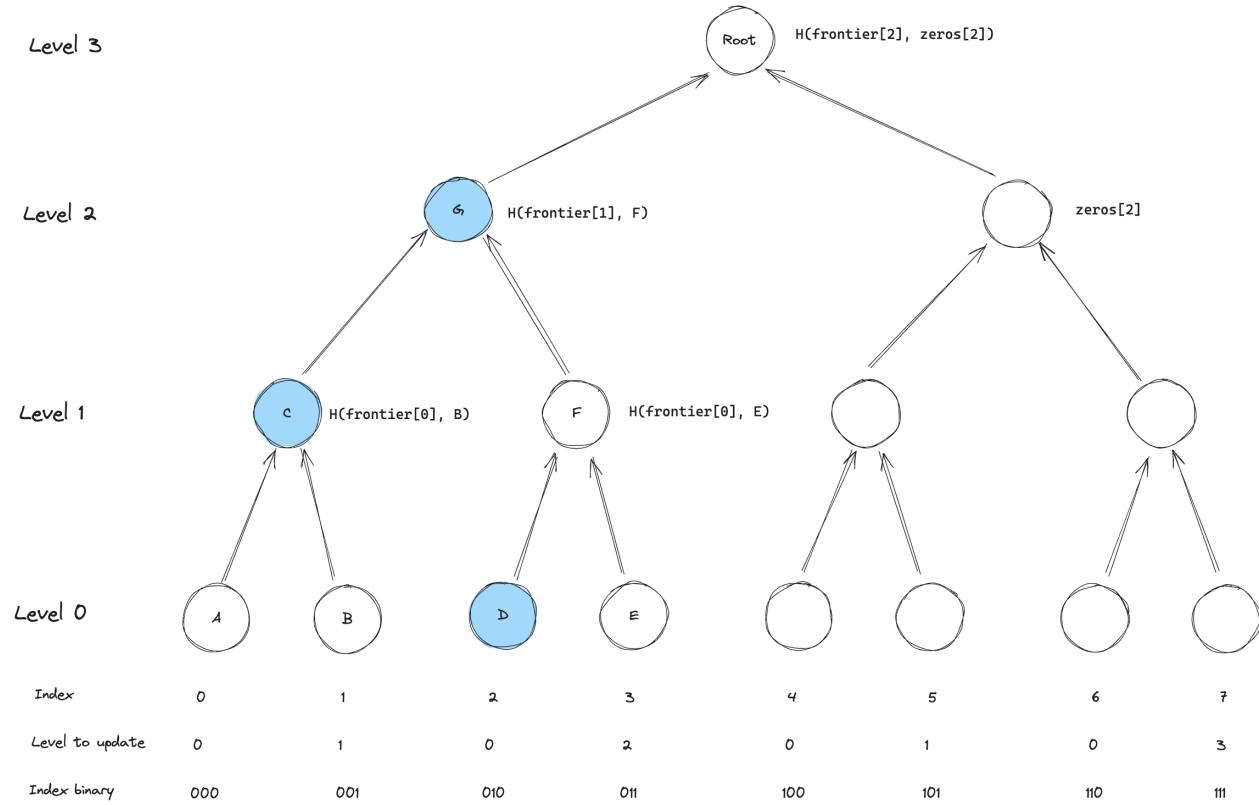
The level to update is simply the number of trailing ones in the binary representation of the index. For a binary tree, we have that every **1** in the binary index represents a "right turn" down the tree. Walking up the tree from the leaf, we can simply count the number of right turns until we hit a left-turn.

How to compute the root

Computing the root based on the frontier is also quite simple. We can use the last index inserted a leaf at to figure out how high up the frontier we should start. Then we know that anything that is at the right of the frontier has not yet been inserted, so all of these values are simply "zeros" values. Zeros here are understood as the root for a subtree only containing zeros.

For example, if we take the tree from above and compute the root for it, we would see

that level 2 was updated last. Meaning that we can simply compute the root as $H(\text{frontier}[2], \text{zeros}[2])$.



For cases where we have built further, we simply "walk" up the tree and use either the frontier value or the zero value for the level.

Pseudo implementation

```
class FrontierTree:
    HEIGHT: immutable(uint256)
    SIZE: immutable(uint256)

    frontier: HashMap[uint256, bytes32] # level => node
    zeros: HashMap[uint256, uint256] # level => root of empty
```

Optimizations

- The `zeros` can be pre-computed and stored in the `Inbox` directly, this way they can be shared across all of the trees.

Data Publication and Availability

DA (and Publication)

This page is heavily based on the Rollup and Data Ramblings documents.

Published Data Format

The "Effects" of a transaction are the collection of state changes and metadata that resulted from executing a transaction. These include:

DA (and Publication)

!(INFO)

This page is heavily based on the Rollup and Data Ramblings documents. As for that, we highly recommend reading [this very nice post](#) written by Jon Charbonneau.

- **Data Availability:** The data is available to anyone right now
- **Data Publication:** The data was available for a period when it was published.

Essentially Data Publication \subset Data Availability, since if it is available, it must also have been published. This difference might be small but becomes important in a few moments.

Progressing the state of the validating light node requires that we can convince it that the data was published - as it needs to compute the public inputs for the proof. The exact method of computing these public inputs can vary depending on the data layer, but generally, it could be by providing the data directly or by using data availability sampling or a data availability committee.

The exact mechanism greatly impacts the security and cost of the system, and will be discussed in the following sections. Before that we need to get some definitions in place.

Definitions

⚠ SECURITY

Security is often used quite in an unspecific manner, "good" security etc, without

specifying what security is. From distributed systems, the *security* of a protocol or system is defined by:

- **Liveness:** Eventually something good will happen.
- **Safety:** Nothing bad will happen.

In the context of blockchain, this *security* is defined by the confirmation rule, while this can be chosen individually by the user, our validating light node (L1 bridge) can be seen as a user, after all, it's "just" another node. For the case of a validity proof based blockchain, a good confirmation rule should satisfy the following sub-properties (inspired by [Sreeram's framing](#)):

- **Liveness:**
 - Data Availability - The chain data must be available for anyone to reconstruct the state and build blocks
 - Ledger Growth - New blocks will be appended to the ledger
 - Censorship Resistance - Honest transactions that are willing to pay will be included if the chain progresses.
- **Safety:**
 - Re-org Resistance - Confirmed transactions won't be reverted
 - Data Publication - The state changes of the block is published for validation check
 - State Validity - State changes along with validity proof allow anyone to check that new state *ROOTS* are correct.

Notice, that safety relies on data publication rather than availability. This might sound strange, but since the validity proof can prove that the state transition function was followed and what changes were made, we strictly don't need the entire state to be available for safety.

With this out the way, we will later be able to reason about the choice of data storage/publication solutions. But before we dive into that, let us take a higher level look at Aztec

to get a understanding of our requirements.

In particular, we will be looking at what is required to give observers (nodes) different guarantees similar to what Jon did in [his post](#). This can be useful to get an idea around what we can do for data publication and availability later.

Rollup 101

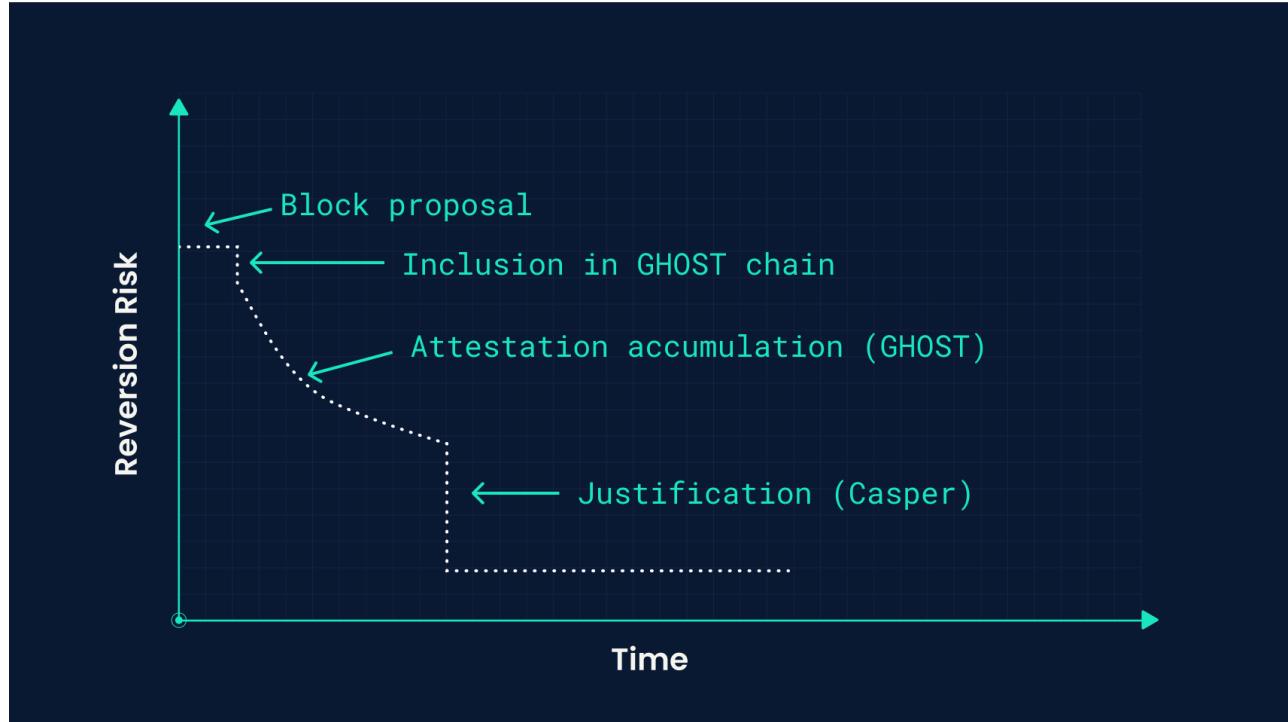
A rollup is broadly speaking a blockchain that put its blocks on some other chain (the host) to make them available to its nodes. Most rollups have a contract on this host blockchain which validates its state transitions (through fault proofs or validity proofs) taking the role of a full-validating light-node, increasing the accessibility of running a node on the rollup chain, making any host chain node indirectly validate its state.

With its state being validated by the host chain, the security properties can eventually be enforced by the host-chain if the rollup chain itself is not progressing. Bluntly, the rollup is renting security from the host. The essential difference between an L1 and a rollup then comes down to who are required for block production (liveness) and to convince the validating light-node (security). For the L1 it is the nodes of the L1, and for the Rollup the nodes of its host (eventually). This in practice means that we can get some better properties for how easy it is to get sufficient assurance that no trickery is happening.

| | Security | Accessibility |
|--|----------|---------------|
| Full node | 😊 | 🙁 |
| Full-verifier light node (L1 state transitioner) | 😊 | 😊 |

With that out the way, we can draw out a model of the rollup as a two-chain system, what Jon calls the *dynamically available ledger* and the *finalized prefix ledger*. The point where we jump from one to the other depends on the confirmation rules applied. In

Ethereum the *dynamically available* chain follows the [LMD-ghost](#) fork choice rule and is the one block builders are building on top of. Eventually consensus forms and blocks from the *dynamic* chain gets included in the *finalized* chain ([Gasper](#)). Below image is from [Bridging and Finality: Ethereum](#).



In rollup land, the *available* chain will often live outside the host where it is built upon before blocks make their way onto the host DA and later get *finalized* by the validating light node that lives on the host as a smart contract.

Depending on the rollup mechanism, rollup full nodes will be able to finalize their own view of the chain as soon as data is available on the host.

Since the rollup cannot add invalid state transitions to the finalized chain due to the validating light node on the host, rollups can be built with or without a separate consensus mechanism for security.

One of the places where the existence of consensus make a difference for the rollup chain is how far you can build ahead, and who can do it.

Consensus

For a consensus based rollup you can run LMD-Ghost similarly to Ethereum, new blocks are built like Ethereum, and then eventually reach the host chain where the light client should also validate the consensus rules before progressing state. In this world, you have a probability of re-orgs trending down as blocks are built upon while getting closer to the finalization. Users can then rely on their own confirmation rules to decide when they deem their transaction confirmed. You could say that the transactions are pre-confirmed until they convince the validating light-client on the host.

No-consensus

If there is no explicit consensus for the Rollup, staking can still be utilized for leader selection, picking a distinct sequencer which will have a period to propose a block and convince the validating light-client. The user can as earlier define his own confirmation rules and could decide that if the sequencer acknowledges his transaction, then he sees it as confirmed. This has weaker guarantees than the consensus-based approach as the sequencer could be malicious and not uphold his part of the deal. Nevertheless, the user could always do an out-of-protocol agreement with the sequencer, where the sequencer guarantees that he will include the transaction or the user will be able to slash him and get compensated.

FERNET

Fernet lives in this category if you have a single sequencer active from the proposal to proof inclusion stage.

Common for both consensus and no-consensus rollups is that the user can decide when he deems his transaction confirmed. If the user is not satisfied with the guarantee provided by the sequencer, he can always wait for the block to be included in the host chain and get the guarantee from the host chain consensus rules.

Data Availability and Publication

As alluded to earlier, we belong to the school of thought that Data Availability and Publication are different things. Generally, what is often referred to as Data Availability is merely Data Publication, e.g., whether or not the data have been published somewhere. For data published on Ethereum you will currently have no issues getting a hold of the data because there are many full nodes and they behave nicely, but they are not guaranteed to continue doing so. New nodes are essentially bootstrapped by other friendly nodes.

With that out the way, it would be prudent to elaborate on our definition from earlier:

- **Data Availability:** The data is available to anyone right now
- **Data Publication:** The data was available for a period when it was published.

With this split, we can map the methods of which we can include data for our rollup. Below we have included only systems that are live or close to live where we have good ideas around the throughput and latency of the data. The latency is based on using Ethereum L1 as the home of the validating light node, and will therefore be the latency between point in time when data is included on the data layer until a point when statements about the data can be included in the host chain.

| Method | Publication | Availability | Quantity | Latency | Description |
|----------|-------------|--------------|---------------------------------------|---------|---|
| calldata | Eth L1 | Eth L1 | 78,125 $\frac{\text{byte}}{\text{s}}$ | None | Part of the transaction payload required to execute history, if you |

| Method | Publication | Availability | Quantity | Latency | Description |
|--------|-------------|------------------------------------|----------|---------|---|
| | | | | | <p>can sync an Ethereum node from zero, this is available.</p> <p>Essentially, if Ethereum lives this is available.</p> <p>Have to compete against everything on Ethereum for blockspace.</p> |
| blobs | Eth L1 | benevolent Eth L1 super full-nodes | x | None | <p>New blob data, will be published but only commitments available from the execution environment.</p> <p>Content can be discarded later and doesn't have to be stored</p> |

| Method | Publication | Availability | Quantity | Latency | Description |
|----------|------------------------------|-----------------------|--|-----------|--|
| | | | | | forever.
Practically a "committee" of whoever wants can keep it, and you rely on someone from this set providing the data to you. |
| ^^ | | | $31,744 \frac{\text{byte}}{\text{s}}$ | None | target of 3 blobs of size 4096 fields (380, 928 bytes per block) |
| ^^ | | | $677,205 \frac{\text{byte}}{\text{s}}$ | None | target of 64 blobs of size 4096 fields (8, 126, 464 bytes per block) |
| Celestia | Celestia + Blobstream bridge | Celestia Full Storage | $161,319 \frac{\text{byte}}{\text{s}}$ | ~100 mins | 2MB blocks.
Can be used in proof after |

| Method | Publication | Availability | Quantity | Latency | Description |
|--------|-------------|--------------|----------|---------|--|
| | | Nodes | | | relay happens, with latency improvements expected. |

Data Layer outside host

When using a data layer that is not the host chain, cost (and safety guarantees) are reduced, and we rely on some "bridge" to tell the host chain about the data. This must happen before our validating light node can progress the block. Therefore the block must be published, and the host must know about it before the host can use it as input to block validation.

This influences how blocks can practically be built, since short "cycles" of publishing and then including blocks might not be possible for bridges with significant delay. This means that a suitable data layer has both sufficient data throughput but also low (enough) latency at the bridge level.

Briefly the concerns we must have for any supported data layer that is outside the host chain is:

- What are the security assumptions of the data layer itself
- What are the security assumptions of the bridge
- What is the expected data throughput (kb/s)
- What is the expected delay (mins) of the bridge

Celestia

Celestia mainnet is starting with a limit of 2 mb/block with 12 second blocks supporting

~166 KB/s.

ⓘ NOTE

They are working on increasing this to 8 mb/block.

As Celestia has just recently launched, it is unclear how much competition there will be for the data throughput, and thereby how much we could expect to get a hold of. Since the security assumptions differ greatly from the host chain (Ethereum) few L2s have been built on top of it yet, and the demand is to be gauged in the future.

Beyond the pure data throughput, we also need Ethereum L1 to know that the data was made available on Celestia. This will require the [blobstream](#) (formerly the quantum gravity bridge) to relay data roots that the rollup contract can process. This is currently done approximately every 100 minutes. Note however, that a separate blobstream is being build by Succinct labs (live on goerli) which should make relays cheaper and more frequent.

Neat structure of what the availability oracles will look like created by the Celestia team:

Espresso

Espresso is not yet live, so the following section is very much in the air, it might be that the practical numbers will change when it is live.

Our knowledge of hotshot is limited here - keeping commentary limited until more educated in this matter.

From their [benchmarks](#), it seems like the system can support 25-30MB/s of throughput by using small committees of 10 nodes. The throughput further is impacted by the size of the node-set from where the committee is picked.

While the committee is small, it seems like they can ensure honesty through the other nodes. But the nodes active here might need a lot of bandwidth to handle both DA

Proposals and VID chunks.

It is not fully clear how often blocks would be relayed to the hotshot contract for consumption by our rollup, but the team says it should be frequent. Cost is estimated to be ~400K gas.

Aztec-specific Data

As part of figuring out the data throughput requirements, we need to know what data we need to publish. In Aztec we have a bunch of data with varying importance; some being important to **everyone** and some being important to **someone**.

The things that are important to **everyone** are the things that we have directly in state, meaning the:

- leaves of the note hash tree
- nullifiers
- public state leafs
- contracts
- L1 → L2
- L2 → L1

Some of these can be moved around between layers, and others are hard-linked to live on the host. For one, moving the cross-chain message L1 → L2 and L2 → L1 anywhere else than the host is fighting an up-hill battle. Also, beware that the state for L2 → L1 messages is split between the data layers, as the messages don't strictly need to be available from the L2 itself, but must be for consumption on L1.

We need to know what these things are to be able to progress the state. Without having the state, we don't know how the output of a state transition should look and cannot prove it.

Beyond the above data that is important to everyone, we also have data that is important to *someone*. These are encrypted and unencrypted logs. Knowing the historic logs is not required to progress the chain, but they are important for the users to ensure that they learn about their notes etc.

A few transaction examples based on our E2E tests have the following data footprints. We will need a few more bytes to specify the sizes of these lists but it will land us in the right ball park.

These were made back in August 2023 and are a bit outdated. They should be updated to also include more complex transactions.

```
Tx ((Everyone, Someone) bytes).
Tx ((192, 1005) bytes): comms=4, nulls=2, pubs=0, l2_to_l1=0,
e_logs=988, u_logs=17
Tx ((672, 3980) bytes): comms=16, nulls=5, pubs=0, l2_to_l1=0,
e_logs=3932, u_logs=48
Tx ((480, 3980) bytes): comms=13, nulls=2, pubs=0, l2_to_l1=0,
e_logs=3932, u_logs=48
Tx ((640, 528) bytes): comms=4, nulls=16, pubs=0, l2_to_l1=0,
e_logs=508, u_logs=20
Tx ((64, 268) bytes): comms=1, nulls=1, pubs=0, l2_to_l1=0,
e_logs=256, u_logs=12
Tx ((128, 512) bytes): comms=2, nulls=2, pubs=0, l2_to_l1=0,
e_logs=500, u_logs=12
Tx ((96, 36) bytes): comms=0, nulls=1, pubs=1, l2_to_l1=0,
e_logs=8, u_logs=28
Tx ((128, 20) bytes): comms=0, nulls=2, pubs=1, l2_to_l1=0,
e_logs=8, u_logs=12
Tx ((128, 20) bytes): comms=1, nulls=1, pubs=1, l2_to_l1=0,
e_logs=8, u_logs=12
Tx ((96, 268) bytes): comms=1, nulls=2, pubs=0, l2_to_l1=0,
e_logs=256, u_logs=12
Tx ((224, 28) bytes): comms=1, nulls=2, pubs=2, l2_to_l1=0,
e_logs=12, u_logs=16
Tx ((480, 288) bytes): comms=1, nulls=2, pubs=6, l2_to_l1=0,
```

For a more liberal estimation, lets suppose we emit 4 nullifiers, 4 new note hashes, and 4 public data writes instead per transaction.

```
Tx ((512, 1036) bytes): comms=4, nulls=4, pubs=4, l2_to_l1=0,
e_logs=988, u_logs=48
```

Assuming that this is a decent guess, and we can estimate the data requirements at different transaction throughput.

Throughput Requirements

Using the values from just above for transaction data requirements, we can get a ball park estimate of what we can expect to require at different throughput levels.

| Throughput | Everyone | Someone | Total |
|------------|--------------------------------------|---------------------------------------|---------------------------------------|
| 1 TPS | $512 \frac{\text{byte}}{\text{s}}$ | $1036 \frac{\text{byte}}{\text{s}}$ | $1548 \frac{\text{byte}}{\text{s}}$ |
| 10 TPS | $5120 \frac{\text{byte}}{\text{s}}$ | $10360 \frac{\text{byte}}{\text{s}}$ | $15480 \frac{\text{byte}}{\text{s}}$ |
| 50 TPS | $25600 \frac{\text{byte}}{\text{s}}$ | $51800 \frac{\text{byte}}{\text{s}}$ | $77400 \frac{\text{byte}}{\text{s}}$ |
| 100 TPS | $51200 \frac{\text{byte}}{\text{s}}$ | $103600 \frac{\text{byte}}{\text{s}}$ | $154800 \frac{\text{byte}}{\text{s}}$ |

Assuming that we are getting $\frac{1}{9}$ of the blob-space or $\frac{1}{20}$ of the calldata and amortize to the Aztec available space.

For every throughput column, we insert 3 marks, for everyone, someone and the total;    meaning that the throughput can be supported when publishing data for

everyone, someone and the total.  meaning that none of it can be supported.

| Space | Aztec Available | 1 TPS | 10 TPS | 50 TPS | 100 Tps |
|---------------------------------|---|-------|--------|--------|---------|
| Calldata | $3,906 \frac{\text{byte}}{\text{s}}$ | ✓✓✓ | 💀💀💀 | 💀💀💀 | 💀💀💀 |
| Eip-4844 | $3,527 \frac{\text{byte}}{\text{s}}$ | ✓✓✓ | 💀💀💀 | 💀💀💀 | 💀💀💀 |
| 64 blob
danksharding | $75,245 \frac{\text{byte}}{\text{s}}$ | ✓✓✓ | ✓✓✓ | ✓✓✓ | ✓✓💀 |
| Celestia
(2mb/12s
blocks) | $17,924 \frac{\text{byte}}{\text{s}}$ | ✓✓✓ | ✓✓✓ | 💀💀💀 | 💀💀💀 |
| Celestia
(8mb/13s
blocks) | $68,376 \frac{\text{byte}}{\text{s}}$ | ✓✓✓ | ✓✓✓ | ✓✓💀 | ✓💀💀 |
| Espresso | Unclear but at least
1 mb per second | ✓✓✓ | ✓✓✓ | ✓✓✓ | ✓✓✓ |

Disclaimer: Remember that these fractions for available space are pulled out of thin air.

With these numbers at hand, we can get an estimate of our throughput in transactions based on our storage medium.

One or multiple data layers?

From the above estimations, it is unlikely that our data requirements can be met by

using only data from the host chain. It is therefore to be considered whether data can be split across more than one data layer.

The main concerns when investigating if multiple layers should be supported simultaneously are:

- **Composability:** Applications should be able to integrate with one another seamlessly and synchronously. If this is not supported, they might as well be entirely separate deployments.
- **Ossification:** By ossification we mean changing the assumptions of the deployments, for example, if an application was deployed at a specific data layer, changing the layer underneath it would change the security assumptions. This is addressed through the [Upgrade mechanism](#).
- **Security:** Applications that depend on multiple different data layers might rely on all its layers to work to progress its state. Mainly the different parts of the application might end up with different confirmation rules (as mentioned earlier) degrading it to the least secure possibly breaking the liveness of the application if one of the layers is not progressing.

The security aspect in particular can become a problem if users deploy accounts to a bad data layer for cost savings, and then cannot access their funds (or other assets) because that data layer is not available. This can be a problem, even though all the assets of the user lives on a still functional data layer.

Since the individual user burden is high with multi-layer approach, we discard it as a viable option, as the probability of user failure is too high.

Instead, the likely design, will be that an instance has a specific data layer, and that "upgrading" to a new instance allows for a new data layer by deploying an entire instance. This ensures that composability is ensured as everything lives on the same data layer. Ossification is possible hence the [upgrade mechanism](#) doesn't "destroy" the old instance. This means that applications can be built to reject upgrades if they believe the new data layer is not secure enough and simple continue using the old.

Privacy is Data Hungry - What choices do we really have?

With the target of 10 transactions per second at launch, in which the transactions are likely to be more complex than the simple ones estimated here, some of the options simply cannot satisfy our requirements.

For one, EIP-4844 is out of the picture, as it cannot support the data requirements for 10 TPS, neither for everyone or someone data.

At Danksharding with 64 blobs, we could theoretically support 50 tps, but will not be able to address both the data for everyone and someone. Additionally this is likely years in the making, and might not be something we can meaningfully count on to address our data needs.

With the current target, data cannot fit on the host, and we must work to integrate with external data layers. Of these, Celestia has the current best "out-the-box" solution, but Eigen-da and other alternatives are expected to come online in the future.

References

- <https://dba.xyz/do-rollups-inherit-security/>
- <https://ethereum.org/en/roadmap/danksharding/>
- <https://eips.ethereum.org/EIPS/eip-4844>
- <https://github.com/ethereum/consensus-specs/blob/dev/specs/deneb/polynomial-commitments.md>
- https://eth2book.info/capella/part2/consensus/lmd_ghost/
- https://eth2book.info/capella/part2/consensus/casper_ffg/
- <https://notes.ethereum.org/cG-j3r7kRD6ChQyxjUdKkw>
- <https://forum.celestia.org/t/security-levels-for-data-availability-for-light-nodes/919>
- <https://ethresear.ch/t/peerdas-a-simpler-das-approach-using-battle-tested-p2p->

[components/16541](#)

- <https://jumpcrypto.com/writing/bridging-and-finality-ethereum/>
- <https://x.com/sreeramkannan/status/1683735050897207296>
- <https://blog.celestia.org/introducing-blobstream/>

Published Data Format

The "Effects" of a transaction are the collection of state changes and metadata that resulted from executing a transaction. These include:

| Field | Type | Description |
|---------------|---|--|
| revertCode | RevertCode | Indicates the reason for reverting in public application logic. 0 indicates success. |
| note_hashes | Tuple<Fr, typeof
MAX_NOTE_HASHES_PER_TX> | The note hashes to be inserted into the note hash tree. |
| nullifiers | Tuple<Fr, typeof
MAX_NULLIFIERS_PER_TX> | The nullifiers to be inserted into the nullifier tree. |
| l2_to_l1_msgs | Tuple<Fr, typeof
MAX_L2_TO_L1_MSGS_PER_TX> | The L2 to L1 messages to be inserted into the |

| Field | Type | Description |
|--------------------|--|---|
| | | messagebox
on L1. |
| public_data_writes | Tuple<PublicDataWrite, typeof
MAX_PUBLIC_DATA_UPDATE_REQUESTS_PER_TX> | Public data
writes to be
inserted into
the public
data tree |
| encrypted_logs | TxL2Logs | Buffers
containing
the emitted
encrypted
logs. |
| unencrypted_logs | TxL2Logs | Buffers
containing
the emitted
unencrypted
logs. |

Each can have several transactions. Thus, an block is presently encoded as:

| byte start | num
bytes | name |
|------------|--------------|-----------------------------------|
| 0x0 | 0x4 | len(newL1ToL2Msgs) (denoted
a) |
| 0x4 | a *
0x20 | newL1ToL2Msgs |

| byte start | num bytes | name |
|--|------------------|---|
| $0\times 4 + a * 0\times 20 = \text{tx0Start}$ | 0×4 | $\text{len}(\text{numTxs})$ (denoted t) |
| | | TxEffec 0 |
| tx0Start | 0×20 | revertCode |
| tx0Start + 0×20 | 0×1 | $\text{len}(\text{noteHashes})$ (denoted b) |
| tx0Start + $0\times 20 + 0\times 1$ | $b * 0\times 20$ | noteHashes |
| tx0Start + $0\times 20 + 0\times 1 + b * 0\times 20$ | 0×1 | $\text{len}(\text{nullifiers})$ (denoted c) |
| tx0Start + $0\times 20 + 0\times 1 + b * 0\times 20 + 0\times 1$ | $c * 0\times 20$ | nullifiers |
| tx0Start + $0\times 20 + 0\times 1 + b * 0\times 20 + 0\times 1 + c * 0\times 20$ | 0×1 | $\text{len}(\text{I2ToL1Msgs})$ (denoted d) |
| tx0Start + $0\times 20 + 0\times 1 + b * 0\times 20 + 0\times 1 + c * 0\times 20 + 0\times 1$ | $d * 0\times 20$ | I2ToL1Msgs |
| tx0Start + $0\times 20 + 0\times 1 + b * 0\times 20 + 0\times 1 + c * 0\times 20 + 0\times 1 + d * 0\times 20$ | 0×1 | $\text{len}(\text{newPublicDataWrites})$ (denoted e) |
| tx0Start + $0\times 20 + 0\times 1 + b * 0\times 20 + 0\times 1 + c * 0\times 20 + 0\times 1 + d * 0\times 20 + 0\times 01$ | $e * 0\times 40$ | newPublicDataWrites |
| tx0Start + $0\times 20 + 0\times 1 + b * 0\times 20 + 0\times 1 + c * 0\times 20 + 0\times 1 + d * 0\times 20 + 0\times 01 + e * 0\times 40$ | 0×04 | $\text{byteLen}(\text{newEncryptedLogs})$ (denoted f) |
| tx0Start + $0\times 20 + 0\times 1 + b * 0\times 20 + 0\times 1 + c * 0\times 20 + 0\times 1 + d * 0\times 20 + 0\times 01 + e * 0\times 40$ | f | newEncryptedLogs |

| byte start | num bytes | name |
|--|-----------|--|
| 0×20 + 0×1 + d * 0×20 + 0×01 + e * 0×40 + 0×4 | | |
| tx0Start + 0×20 + 0×1 + b * 0×20 + 0×1 + c *
0×20 + 0×1 + d * 0×20 + 0×01 + e * 0×40 + 0×4
+ f | 0×04 | byteLen(newUnencryptedLogs)
(denoted g) |
| tx0Start + 0×20 + 0×1 + b * 0×20 + 0×1 + c *
0×20 + 0×1 + d * 0×20 + 0×01 + e * 0×40 + 0×4
+ f + 0×4 | g | newUnencryptedLogs |
| | | , |
| | | TxEffec 1 |
| | | ... |
| | | , |
| | | ... |
| | | TxEffec (t - 1) |
| | | ... |
| | | , |

Logs

Logs on Aztec are similar to logs on Ethereum, enabling smart contracts to convey arbitrary data to external entities. Offchain applications can use logs to interpret events that have occurred on-chain. There are three types of log:

- [Unencrypted log](#).
- [Encrypted log](#).
- [Encrypted note preimage](#).

Requirements

1. Availability: The logs get published.

A rollup proof won't be accepted by the rollup contract if the log preimages are not available. Similarly, a sequencer cannot accept a transaction unless log preimages accompany the transaction data.

2. Immutability: A log cannot be modified once emitted.

The protocol ensures that once a proof is generated at any stage (for a function, transaction, or block), the emitted logs are tamper-proof. In other words, only the original log preimages can generate the committed hashes in the proof.

3. Integrity: A contract cannot impersonate another contract.

Every log is emitted by a specific contract, and users need assurances that a particular log was indeed generated by a particular contract (and not some malicious impersonator contract). The protocol ensures that the source contract's address for a log can be verified, while also preventing the forging of the address.

Log Hash

Hash Function

The protocol uses **SHA256** as the hash function for logs, and then reduces the 256-bit result to 248 bits for representation as a field element.

Throughout this page, `hash(value)` is an abbreviated form of:

```
truncate_to_field(SHA256(value))
```

Hashing

Regardless of the log type, the log hash is derived from an array of fields, calculated as:

```
hash(log_preimage[0], log_preimage[1], ..., log_preimage[N - 1])
```

Here, *log_preimage* is an array of field elements of length *N*, representing the data to be broadcast.

Emitting Logs from Function Circuits

A function can emit an arbitrary number of logs, provided they don't exceed the specified `[limit]`. The function circuits must compute a hash for each log, and push all the hashes into the public inputs for further processing by the protocol circuits.

Aggregation in Protocol Circuits

To minimize the on-chain verification data size, protocol circuits aggregate log hashes. The end result is a single hash within the base rollup proof, encompassing all logs of the same type.

Each protocol circuit outputs two values for each log type:

- `accumulated_logs_hash`: A hash representing all logs.
- `accumulated_logs_length`: The total length of all log preimages.

Both the `accumulated_logs_hash` and `accumulated_logs_length` for each type are included in the base rollup's `txs_effect_hash`. When rolling up to merge and root circuits, the two input proof's `txs_effect_hashes` are hashed together to form the new value of `txs_effect_hash`.

When publishing a block on L1, the raw logs of each type and their lengths are provided (**Availability**), hashed and accumulated into each respective `accumulated_logs_hash` and `accumulated_logs_length`, then included in the on-chain recalculation of `txs_effect_hash`. If this value doesn't match the one from the rollup circuits, the block will not be valid (**Immutability**).

For private and public kernel circuits, beyond aggregating logs from a function call, they ensure that the contract's address emitting the logs is linked to the `logs_hash`. For more details, refer to the "Hashing" sections in [Unencrypted Log](#), [Encrypted Log](#), and [Encrypted Note Preimage](#).

Unencrypted Log

Unencrypted logs are used to communicate public information out of smart contracts. They can be emitted from both public and private functions.

 **INFO**

Emitting unencrypted logs from private functions may pose a privacy leak. However, in-protocol restrictions are intentionally omitted to allow for potentially valuable use cases, such as custom encryption schemes utilizing Fully

Homomorphic Encryption (FHE), and similar scenarios.

Hashing

Following the iterations for all private or public calls, the tail kernel circuits hash each log hash with the contract contract before computing the *accumulated_logs_hash*.

1. Hash the *contract_address* to each *log_hash*:

- o *log_hash_a* = *hash(contract_address_a, log_hash_a)*
- o Repeat the process for all *log_hashes* in the transaction.

2. Accumulate all the hashes and output the final hash to the public inputs:

- o *accumulated_logs_hash* = *hash(log_hash[0], log_hash[1], ..., log_hash[N - 1])* for N logs.

Encoding

The following represents the encoded data for an unencrypted log:

```
log_data = [log_preimage_length, contract_address, ...log_preimage]
```

Verification

```
function hash_log_data(logs_data) {
    const log_preimage_length = logs_data.read_u32();
    logs_data.accumulated_logs_length += log_preimage_length;
    const contract_address = logs_data.read_field();
    const log_preimage = logs_data.read_fields(log_preimage_length);
    const log_hash = hash(...log_preimage);
```

Encrypted Log

Encrypted logs contain information encrypted using the recipient's key. They can only be emitted from private functions. This restriction is due to the necessity of obtaining a secret for log encryption, which is challenging to manage privately in a public domain.

Hashing

Private kernel circuits ensure the association of the contract address with each encrypted *log_hash*. However, unlike unencrypted logs, submitting encrypted log preimages with their contract address poses a significant privacy risk. Therefore, instead of using the *contract_address*, a *masked_contract_address* is generated for each encrypted *log_hash*.

The *masked_contract_address* is a hash of the *contract_address* and a random value *randomness*, computed as:

```
masked_contract_address = hash(contract_address, randomness).
```

Here, *randomness* is generated in the private function circuit and supplied to the private kernel circuit. The value must be included in the preimage for encrypted log generation. The private function circuit is responsible for ensuring that the *randomness* differs for every encrypted log to avoid potential information linkage based on identical *masked_contract_address*.

After successfully decrypting an encrypted log, one can use the *randomness* in the log preimage, hash it with the *contract_address*, and verify it against the *masked_contract_address* to ascertain that the log originated from the specified contract.

1. Hash the *contract_address_tag* to each *log_hash*:

- *masked_contract_address_a* = *hash(contract_address_a, randomness)*
- *log_hash_a* = *hash(contract_address_tag_a, log_hash_a)*
- Repeat the process for all *log_hashes* in the transaction.

2. Accumulate all the hashes in the tail and outputs the final hash to the public inputs:

- *accumulated_logs_hash* = *hash(log_hash[0], log_hash[1], ..., log_hash[N - 1])* for N logs, with hashes defined above.

Note that, in some cases, the user may want to reveal which contract address the encrypted log came from. Providing a *randomness* value of 0 signals that we should not mask the address, so in this case the log hash is simply:

```
log_hash_a = hash(contract_address_a, log_hash_a)
```

Encoding

The following represents the encoded data for an encrypted log:

```
log_data = [log_preimage_length, masked_contract_address,  
...log_preimage]
```

Verification

```
function hash_log_data(logs_data) {  
    const log_preimage_length = logs_data.read_u32();  
    logs_data.accumulated_logs_length += log_preimage_length;
```

Encrypted Note Preimage

Similar to [encrypted logs](#), encrypted note preimages are data that only entities possessing the keys can decrypt to view the plaintext. Unlike encrypted logs, each encrypted note preimage can be linked to a note, whose note hash can be found in the block data.

Note that a note can be "shared" to one or more recipients by emitting one or more encrypted note preimages. However, this is not mandatory, and there may be no encrypted preimages emitted for a note if the information can be obtained through alternative means.

Hashing

As each encrypted note preimage can be associated with a note in the same transaction, enforcing a *contract_address_tag* is unnecessary. Instead, by calculating the *note_hash* using the decrypted note preimage, hashed with the *contract_address*, and verify it against the block data, the recipient can confirm that the note was emitted from the specified contract.

The kernel circuit simply accumulates all the hashes:

- `accumulated_logs_hash = hash(log_hash[0], log_hash[1], ..., log_hash[N - 1])` for N logs.

Encoding

The following represents the encoded data for an unencrypted note preimage:

```
log_data = [log_preimage_length, ...log_preimage]
```

Verification

```
function hash_log_data(logs_data) {
    const log_preimage_length = logs_data.read_u32();
    logs_data.accumulated_logs_length += log_preimage_length;
    const log_preimage = logs_data.read_fields(log_preimage_length);
    return hash(...log_preimage);
}
```

Log Encryption

Refer to [Private Message Delivery](#) for detailed information on generating encrypted data.

Pre-Compiled Contracts

Registry

<!-- Review notes:

Registry

The protocol should allow users to express their preferences in terms of encryption & tagging mechanisms, and also provably advertise their encryption & tagging public keys. A canonical registry contract provides an application-level solution to both problems.

Overview and Usage

At the application level, a canonical singleton contract allows accounts to register their public keys and their preference for encryption & tagging methods. This data is kept in public storage for anyone to check when they need to send a note to an account.

An account can directly call the registry via a public function to set or update their public keys and their encryption & tagging preferences. New accounts should register themselves on deployment. Alternatively, anyone can create an entry for a new account (but not update) if they demonstrate that the public key and encryption & tagging method can be hashed to the new account's address. This allows third party services to register addresses to improve usability.

An app contract can provably read the registry during private execution via a merkle membership proof against a recent public state root, using the [archive tree](#). The rationale for not making a call to the registry to read is to reduce the number of function calls. When reading public state from private-land, apps must set a `max_block_number` for the current transaction to ensure the public state root is not more than `N = max_block_number - current_block_number` blocks old. This means that, if a user rotates their public key, for at most `N` blocks afterwards they may still receive notes encrypted using their old public key, which we consider to be acceptable.

An app contract can also prove that an address is not registered in the registry via a non-inclusion proof, since the public state tree is implemented as an indexed merkle tree. To prevent an app from proving that an address is not registered when in fact it was registered less than N blocks ago, we implement this check as a public function. This means that the transaction may leak that an undisclosed application attempted to interact with a non-registered address but failed.

Note that, if an account is not registered in the registry, a sender could choose to supply the public key along with the preimage of an address on-the-fly , if this preimage was shared with them off-chain. This allows a user to send notes to a recipient before the recipient has deployed their account contract.

Pseudocode

The registry contract exposes functions for setting public keys and encryption methods, plus a public function for proving non-membership of some address. Reads are meant to be done directly via storage proofs and not via calls to save on proving times. Encryption and tagging preferences are expressed via their associated precompile address.

```
contract Registry

    public mapping(address => { keys, precompile_address })
registry

    public fn set(keys, precompile_address)
        this.do_set(msg_sender, keys, precompile_address)

    public fn set_from_preimage(address, keys,
precompile_address, ...address_preimage)
        assert address not in registry
        assert hash(keys, precompile_address,
```

Storage Optimizations

The registry stores a struct for each user, which means that each entry requires multiple storage slots. Reading multiple storage slots requires multiple merkle membership proofs, which increase the total proving cost of any execution that needs access to the registry.

To reduce the number of merkle membership proofs, the registry keeps in storage only the hash of the data stored, and emits the preimage as an unencrypted event. Nodes are expected to store these preimages, so they can be returned when clients query for the public keys for an address. Clients then prove that the preimage hashes to the commitment stored in the public data tree via a single merkle membership proof.

Note that this optimization may also be included natively into the protocol, [pending this discussion](#).

Multiple Recipients per Address

While account contracts that belong to individual users have a clear set of public keys to announce, some private contracts may be shared by a group of users, like in a multisig or an escrow contract. In these scenarios, we want all messages intended for the shared contract to actually be delivered to all participants, using the encryption method selected by each.

This can be achieved by having the registry support multiple sets of keys and precompiles for each entry. Applications can then query the registry and obtain a list of recipients, rather than a single one.

The registry limits multi-recipient registrations to no more than

`MAX_ENTRIES_PER_ADDRESS` to prevent abuse, since this puts an additional burden on the sender, who needs to emit the same note multiple times, increasing the cost of their transaction.

Contracts that intend to register multiple recipients should account for those recipients eventually rotating their keys. To support this, contracts should include a method to refresh the registered addresses:

```
contract Sample

    private address[] owners

    private fn register()
        let to_register = owners.map(owner =>
read_registry(owner))
        registry.set(this, to_register)
```

Discussion

See [*Addresses, keys, and sending notes \(Dec 2023 edition\)*](#) for relevant discussions on this topic.

Private Message Delivery

Private message delivery encompasses the encryption, tagging, and broadcasting of private messages on the Aztec Network.

Private Message Delivery

In Aztec, users need to pass private information between each other. Whilst Aztec enables users to share arbitrary private messages, we'll often frame the discussion toward...

Guidelines

Application contracts are in control of creating, encrypting, tagging, and broadcasting private notes to users. As such, each application is free to follow whatever scheme it p...

Private Message Delivery

In Aztec, users need to pass private information between each other. Whilst Aztec enables users to share arbitrary private messages, we'll often frame the discussion towards a sender sharing the preimage of a private note with some recipient.

If Alice executes a function that generates a note for Bob:

1. Alice will need to **encrypt** that note such that Bob, and only Bob is able to decrypt it.
2. Alice will need to **broadcast** the encrypted note ciphertext so as to make it available for Bob to retrieve.
3. Alice will need to **broadcast** a 'tag' alongside the encrypted note ciphertext. This tag must be identifiable by Bob's chosen [note discovery protocol](#) but not identifiable by any third party as "intended for Bob".

Requirements

- **Users must be able to choose their note tagging mechanism.** We expect improved note discovery schemes to be designed over time. The protocol should be flexible enough to accommodate them and for users to opt in to using them as they become available. This flexibility should be extensible to encryption mechanisms as well as a soft requirement.
- **Users must be able to receive notes before interacting with the network.** A user should be able to receive a note just by generating an address. It should not be necessary for them to deploy their account contract in order to receive a note.
- **Applications must be able to safely send notes to any address.** Sending a note to an account could potentially transfer control of the call to that account, allowing the account to control whether they want to accept the note or not, and

potentially bricking an application, since there are no catching exceptions in private function execution.

- **Addresses must be as small as possible.** Addresses will be stored and broadcasted constantly in applications. Larger addresses means more data usage, which is the main driver for cost. Addresses must fit in at most 256 bits, or ideally a single field element.
- **Total number of function calls should be minimized.** Every function call requires an additional iteration of the private kernel circuit, which adds several seconds of proving time.
- **Encryption keys should be rotatable.** Users should be able to rotate their encryption keys in the event their private keys are compromised, so that any further interactions with apps can be private again, without having to migrate to a new account.

Constraining Message Delivery

The protocol will enable app developers to constrain the correctness of the following:

1. The encryption of a user's note.
2. The generation of the tag for that note.
3. The publication of that note and tag to the correct data availability layer.

Each app will define whether to constrain each such step. Encryption and tagging will be done through a set of [precompiled contracts](#), each contract offering a different mechanism, and users will advertise their preferred mechanisms in a canonical [registry](#).

The advantages of this approach are:

1. It enables a user to select their preferred [note discovery protocol](#) and [encryption scheme](#).

2. It ensures that notes are correctly encrypted with a user's public encryption key.
3. It ensures that notes are correctly tagged for a user's chosen note discovery protocol.
4. It provides scope for upgrading these functions or introducing new schemes as the field progresses.
5. It protects applications from malicious unprovable functions.

Note Discovery Protocol Selection

In order for a user to consume notes that belong to them, they need to identify, retrieve and decrypt them. A simple, privacy-preserving approach to this would be to download all of the notes and attempt decryption. However, the total number of encrypted notes published by the network will be substantial, making it infeasible for some users to do this. Those users will want to utilize a note discovery protocol to privately identify their notes.

Selection of the encryption and tagging mechanisms to use for a particular note are the responsibility of a user's wallet rather than the application generating the note. This is to ensure the note is produced in a way compatible with the user's chosen note discovery scheme. Leaving this decision to applications could result in user's having to utilise multiple note discovery schemes, a situation we want to avoid.

User Handshaking

Even if Alice correctly encrypts the note she creates for Bob and generates the correct tag to go with it, how does Bob know that Alice has sent him a note? Bob's note discovery protocol may require him to speculatively 'look' for notes with the tags that Alice (and his other counterparties) have generated. If Alice and Bob know each other then they can communicate out-of-protocol. But if they have no way of interacting then the network needs to provide a mechanism by which Bob can be

alerted to the need to start searching for a specific sequence of tags.

To facilitate this we will deploy a canonical 'handshake' contract that can be used to create a private note for a recipient containing the sender's information (e.g. public key). It should only be necessary for a single handshake to take place between two users. The notes generated by this contract will be easy to identify, enabling users to retrieve these notes, decrypt them and use the contents in any deterministic tag generation used by their chosen note discovery protocol.

Encryption and Decryption

Applications should be able to provably encrypt data for a target user, as part of private message delivery. As stated in the Keys section, we define three types of encrypted data, based on the sender and the recipient, from the perspective of a user:

Incoming data: data created by someone else, encrypted for and sent to the user.
Outgoing data: data created by the user to be sent to someone else, encrypted for the user.
Internal incoming data: data created by the user, encrypted for and sent to the user. Encryption mechanisms support these three types of encryption, which may rely on different keys advertised by the user.

Key Abstraction

To support different kinds of encryption mechanisms, the protocol does not make any assumptions on the type of public keys advertised by each user. Validation of their public keys is handled by the precompile contract selected by the user.

Provable Decryption

While provable encryption is required to guarantee correct private message delivery,

provable decryption is required for disclosing activity within an application. This allows auditability and compliance use cases, as well as being able to prove that a user did not execute certain actions. To support this, encryption precompiles also allow for provable decryption.

Note Tagging

Note discovery schemes typically require notes to be accompanied by a stream of bytes generated specifically for the note discovery protocol. This 'tag' is then used in the process of note identification. Whilst the tag itself is not sufficient to enable efficient note retrieval, the addition of it alongside the note enables the note discovery protocol to privately select a subset of the global set of notes to be returned to the user. This subset may still require some degree of trial-decryption but this is much more feasible given the reduced dataset.

When applications produce notes, they will need to call a protocol defined contract chosen by the recipient and request that a tag be generated. From the protocol's perspective, this tag will simply be a stream of bytes relevant only to the recipient's note discovery protocol. It will be up to the precompile to constrain that the correct tag has been generated and from there the protocol circuits along with the rollup contract will ensure that the tag is correctly published along with the note.

Constraining tag generation is not solely about ensuring that the generated tag is of the correct format. It is also necessary to constrain that tags are generated in the correct sequence. A tag sequence with duplicate or missing tags makes it much more difficult for the recipient to retrieve their notes. This will likely require tags to be nullified once used.

Guidelines

Application contracts are in control of creating, encrypting, tagging, and broadcasting private notes to users. As such, each application is free to follow whatever scheme it prefers, choosing to override user preferences or use custom encryption and note tagging mechanisms. However, this may hinder composability, or not be compatible with existing wallet software.

In order to satisfy the requirements established for private message delivery, we suggest the following guidelines when building applications, which leverage the canonical [registry](#) contract.

Provably Sending a Note

To provably encrypt, tag, and send a note to a recipient, applications should first check the registry. This ensures that the latest preferences for the recipient are honored, in case they rotated their keys or updated their precompile preference. The registry should be queried via a direct storage read and not a function call, in order to save an additional recursion which incurs in extra proving time.

If the recipient is not in the registry, then the app should allow the sender to provide the recipient's public key from the recipient's address preimage. This allows users who have never interacted with the chain to receive encrypted notes, though it requires a collaborative sender.

If the user is not in the registry and the sender cannot provide the address preimage, then the application must prove that the user was not in the registry, or a malicious sender could simply not submit a correct merkle membership proof for the read and grief the recipient. In this scenario, it is strongly recommended that the application skips the note for the recipient as opposed to failing. This prevents an unregistered address from accidentally or maliciously bricking an application, if there is a note delivery to them in a critical code path in the application.

Execution of the precompile that implements the recipient's choice for encryption and tagging should be done using a batched delegated call, to amortize the cost of sending multiple notes using the same method, and to ensure the notes are broadcasted from the application contract's address.

Pseudocode

The following pseudocode covers how to provably send a note to a recipient, given an `encryption_type` (incoming, outgoing, or internal incoming). Should the registry support [multiple entries for a given recipient](#), this method must execute a batched call per each entry recovered from the registry.

```
fn provably_send_note(recipient, note, encryption_type)

    let block_number = context.latest_block_number
    let public_state_root = context.roots[block_number].public_state
    let storage_slot = calculate_slot(registry_address,
        registry_base_slot, recipient)

    let public_keys, precompile_address
    if storage_slot in public_state_root
        context.update_tx_max_valid_block_number(block_number + N)
        public_keys, precompile_address =
    indexed_merkle_read(public_state_root, storage_slot)
    else if recipient in pxe_oracle
        address_preimage = pxe_oracle.get_preimage(recipient)
        assert hash(address_preimage) == recipient
        public_keys, precompile_address = address_preimage
    else
        registry_address.assert_non_membership(recipient)
        return

batch_private_delegate_call(precompile_address.encrypt_and_broadcast,
{ public_keys, encryption_type, recipient, note })
```

Unconstrained Message Delivery

Applications may choose not to constrain proper message delivery, based on their requirements. In this case, the guidelines are the same as above, but without constraining correct execution, and without the need to assert non-membership when the recipient is not in the registry. Apps can achieve this by issuing a synchronous [unconstrained call](#) to the encryption precompile `encrypt_and_tag` function, and emitting the resulting encrypted note.

This flexibility is useful in scenarios where the sender can be trusted to make its best effort so the recipient receives their private messages, since it reduces total proving time. An example is a standalone direct value transfer, where the sender wants the recipient to access the funds sent to them.

Delivering Messages for Self

Applications may encrypt, tag, and broadcast messages for the same user who's initiating a transaction, using the outgoing or the incoming internal encryption key. This allows a user to have an on-chain backup of their private transaction history, which they can use to recover state in case they lose their private database. In this scenario, unconstrained message delivery is recommended, since the sender is incentivized to correctly encrypt message for themselves.

Applications may also choose to query the user wallet software via an oracle call, so the wallet can decide whether to broadcast the note to self on chain based on user preferences. This allows users to save on gas costs by avoiding unnecessary note broadcasts if they rely on other backup strategies.

Last, applications with strong compliance and auditability requirements may choose to enforce provable encryption, tagging, and delivery to the sender user. This ensures that all user activity within the application is stored on-chain, so the user can later provably

disclose their activity or repudiate actions they did not take.

Delivering Messages to Multiple Recipients via Shared Secrets

As an alternative to registering [multiple recipients for a given address](#), multisig participants may deploy a contract using a shared secret derived among them. This makes it cheaper to broadcast messages to the group, since every note does not need to be individually encrypted for each of them. However, it forces all recipients in the group to use the same encryption and tagging method, and adds an extra address they need to monitor for note discovery.

Discussions

See [Addresses, keys, and sending notes \(Dec 2023 edition\)](#) and [Broadcasting notes in token contracts](#) for relevant discussions on this topic.

Gas & Fees

The Aztec network uses a fee system to incentivize sequencers to process transactions and publish blocks.

This section breaks down:

- fee juice
- how users specify gas/fee parameters in their transactions
- fee abstraction
- tracking gas/fee information in the kernel circuits
- how gas/fees cover the costs of transaction execution
- published data pertaining to gas/fees

Fee Juice

Fee Juice is an enshrined asset in the Aztec network that is used to pay fees.

Specifying Gas & Fee Info

When users submit a TxExecutionRequest on the Aztec Network, they provide a TxContext, which holds GasSettings for the transaction.

Transaction Setup and Teardown

All transactions on the Aztec network have a private component, which is processed locally, and optionally have a public component, which is processed by sequencers usi...

Kernel Tracking

Gas and fees are tracked throughout the kernel circuits to ensure that users are charged correctly for their transactions.

Fee Schedule

The transaction fee is comprised of a DA component, an L2 component, and an inclusion fee. The DA and L2 components are calculated by multiplying the gas consumed in...

Published Gas & Fee Data

When a block is published to L1, it includes information about the gas and fees at a block-level, and at a transaction-level.

Fee Juice

Fee Juice is an enshrined asset in the Aztec network that is used to pay fees.

It has several important properties:

- It is fungible
- It cannot be transferred between accounts on the Aztec network
- It is obtained on Aztec via a bridge from Ethereum
- It only has public balances

All transactions on the Aztec network have a [non-zero transaction_fee](#), denominated in FPA, which must be paid for the transaction to be included in the block.

When a block is successfully published on L1, the sequencer is paid on L1 the sum of all transaction fees in the block, denominated in FPA.



DANGER
We need a definition of the L1 fee juice.

Specifying Gas & Fee Info

When users submit a `TxExecutionRequest` on the Aztec Network, they provide a `TxContext`, which holds `GasSettings` for the transaction.

An abridged version of the class diagram is shown below:

 NOTE

All fees are denominated in [Fee Juice](#).

Gas Dimensions and Max Inclusion Fee

Transactions are metered for their gas consumption across two dimensions:

1. **Data Availability (DA) Gas:** This dimension measures data usage by the transaction, e.g. creating/spending notes, emitting logs, etc.
2. **Layer 2 (L2) Gas:** This dimension measures computation usage of the public VM.

This is similar to the gas model in Ethereum, where transactions consume gas to perform operations, and may also consume blob gas for storing data.

Separately, every transaction has overhead costs associated with it, e.g. verifying its encompassing rollup proof on L1, which are captured in the `maxInclusionFee`, which is not tied to gas consumption on the transaction, but is specified in FPA.

See the [Fee Schedule](#) for a detailed breakdown of costs associated with different actions.

gasLimits and teardownGasLimits

Transactions can optionally have a "teardown" phase as part of their public execution, during which the "transaction fee" is available to public functions. This is useful to transactions/contracts that need to compute a "refund", e.g. contracts that facilitate [fee abstraction](#).

Because the transaction fee must be known at the time teardown is executed, transactions must effectively "prepay" for the teardown phase. Thus, the `teardownGasLimits` are portions of the `gasLimits` that are reserved for the teardown phase.

For example, if a transaction has `gasLimits` of 1000 DA gas and 2000 L2 gas, and `teardownGasLimits` of 100 DA gas and 200 L2 gas, then the transaction will be able to consume 900 DA gas and 1800 L2 gas during the main execution phase, but 100 DA gas and 200 L2 gas will be consumed to cover the teardown phase: even if teardown does not consume that much gas, the transaction will still be charged for it; even if the transaction does not have a teardown phase, the gas will still be consumed.

maxFeesPerGas and feePerGas

The `maxFeesPerGas` field specifies the maximum fees that the user is willing to pay per gas unit consumed in each dimension.

Separately, the protocol specifies the current `feePerGas` for each dimension, which is used to calculate the transaction fee.

These are held in the L2 blocks `Header`

A transaction cannot be executed if the `maxFeesPerGas` is less than the `feePerGas` for any dimension.

The `feePerGas` is presently held constant at `1` for both dimensions, but may be updated in future protocol versions.

`totalFees` is the total fees collected in the block in FPA.

`coinbase` is the L1 address that receives the fees.

Transaction Fee

The transaction fee is calculated as:

```
transactionFee = maxInclusionFee + (DA gas consumed *  
feePerDaGas) + (L2 gas consumed * feePerL2Gas)
```

 NOTE

Why is the "max" inclusion fee charged? We're working on a mechanism that will allow users to specify a maximum fee they are willing to pay, and the network will only charge them the actual fee. This is not yet implemented, so the "max" fee is always charged.

See more on how the "gas consumed" values are calculated in the [Fee Schedule](#).

Maximum Transaction Fee

The final transaction fee cannot be calculated until all public function execution is complete. However, a maximum theoretical fee can be calculated as:

```
maxTransactionFee = maxInclusionFee + (gasLimits.daGas *  
maxFeesPerDaGas) + (gasLimits.l2Gas * maxFeesPerL2Gas)
```

This is useful for imposing [validity conditions](#).

fee_payer

The `fee_payer` is the entity that pays the transaction fee.

It is effectively set in private by the contract that calls
`context.set_as_fee_payer()`.

This manifests as a boolean flag `is_fee_payer` in the
`PrivateCircuitPublicInputs`. The private kernel circuits will check this flag for
every call stack item.

When a call stack item is found with `is_fee_payer` set, the kernel circuit will set
`fee_payer` in its `PrivateKernelCircuitPublicInputs` to be the
`callContext.contractAddress`.

This is subsequently passed through the `PublicKernelCircuitPublicInputs` to
the `KernelCircuitPublicInputs`.

If the `fee_payer` is not set, the transaction will be considered invalid.

If a transaction attempts to set `fee_payer` multiple times, the transaction will be
considered invalid.

Transaction Setup and Teardown

All transactions on the Aztec network have a private component, which is processed locally, and optionally have a public component, which is processed by sequencers using the [Public VM \(AVM\)](#).

Transactions are broken into distinct phases:

1. Private setup
2. Private app logic
3. Public setup
4. Public app logic
5. Public teardown
6. Base rollup

The private setup phase is used to specify what public function will be called for public teardown, and what entity will pay the transaction fee (i.e. the `fee_payer`).

The "setup" phases are "non-revertible", meaning that if execution fails, the transaction is considered invalid and cannot be included in a block.

If execution fails in the private app logic phase, the user will not be able to generate a valid proof of their private computation, so the transaction will not be included in a block.

If the execution fails in the public app logic the *side effects* from private app logic and public app logic will be reverted, but the transaction can still be included in a block. Execution then proceeds to the public teardown phase.

If the execution fails in the public teardown phase, the *side effects* from private app logic, public app logic, and public teardown will be reverted, but the transaction can still be included in a block. Execution then proceeds to the base rollup phase.

In the event of a failure in public app logic or teardown, the user is charged their full [gas limit](#) for the transaction across all dimensions.

The public teardown phase is the only phase where the final transaction fee is available to public functions. [See more](#).

In the base rollup, the kernel circuit injects a public data write that levies the transaction fee on the `fee_payer`.

An example: Fee Abstraction

Consider a user, Alice, who does not have FPA but wishes to interact with the network. Suppose she has a private balance of a fictitious asset "BananaCoin" that supports public and private balances.

Suppose there is a Fee Payment Contract (FPC) that has been deployed by another user to the network. Alice can structure her transaction as follows:

0. Before the transaction, Alice creates a private authwit in her wallet, allowing the FPC to transfer to public a specified amount of BananaCoin from Alice's private balance to the FPC's public balance.
1. Private setup:
 - Alice calls a private function on the FPC which is exposed for public fee payment in BananaCoin.
 - The FPC checks that the amount of teardown gas Alice has allocated is sufficient to cover the gas associated with the teardown function it will use to provide a refund to Alice.
 - The FPC specifies its teardown function as the one the transaction will use.

- The FPC enqueues a public call to itself for the public setup phase.
- The FPC designates itself as the `fee_payer`.

2. Private app logic:

- Alice performs an arbitrary computation in private, potentially consuming DA gas.

3. Public setup:

- The FPC transfers the specified amount of BananaCoin from Alice to itself.

4. Public app logic:

- Alice performs an arbitrary computation in public, potentially consuming DA and L2 gas.

5. Public teardown:

- The FPC looks at `transaction_fee` to compute Alice's corresponding refund of BananaCoin.
- The FPC transfers the refund to Alice via a partial note.

6. Base rollup:

- The Base rollup kernel circuit injects a public data write that levies the transaction fee on the `fee_payer`.

This illustrates the utility of the various phases. In particular, we see why the setup phase must not be revertible: if Alice's public app logic fails, the FPC is still going to pay the fee in the base rollup; if public setup were revertible, the transfer of Alice's BananaCoin would revert so the FPC would be losing money.

Sequencer Whitelisting

Because a transaction is invalid if it fails in the public setup phase, sequencers are taking a risk by processing them. To mitigate this risk, it is expected that sequencers will only process transactions that use public functions that they have whitelisted.

Defining Setup

The private function that is executed first is referred to as the "entrypoint".

Tracking which side effects belong to setup versus app logic is done by keeping track of **side effect counters**, and storing the value of the counter at which the setup phase ends within the private context.

This value is stored in the `PrivateContext` as the `min_revertible_side_effect_counter`, and is set by calling `context.end_setup()`.

This is converted into the `PrivateCircuitPublicInputs` as `min_revertible_side_effect_counter`.

Execution of the entrypoint is always verified/processed by the `PrivateKernelInit` circuit.

It is only the `PrivateKernelInit` circuit that looks at the `min_revertible_side_effect_counter` as reported by `PrivateCirclePublicInputs`, and thus it is only the entrypoint that can effectively call `context.end_setup()`.

Defining Teardown

At any point during private execution, a contract may call `context.set_public_teardown_function` to specify a public function that will be called during the public teardown phase. This function takes the same arguments as `context.call_public_function`, but does not have a side effect counter

associated with it.

Similar to `call_public_function`, this results in the hash of a `PublicCallStackItem` being set on `PrivateCircuitPublicInputs` as `public_teardown_function_hash`.

The private kernel circuits will verify that this hash is set at most once.

Interpreting the `min_revertible_side_effect_counter`

Notes, nullifiers, and logs are examples of side effects that are partitioned into setup and app logic.

[Enqueueing a public function](#) from private is also a side effect: if the counter associated with an enqueued public function is less than the `min_revertible_side_effect_counter`, the public function will be executed during the public setup phase, otherwise it will be executed during the public app logic phase.

As mentioned above, setting the public teardown function is not a side effect.

If a transaction has enqueued public functions, or has a public teardown function, then during the `PrivateKernelTailToPublic` the `min_revertible_side_effect_counter` is used to partition the side effects produced during private execution into revertible and non-revertible sets on the `PublicKernelCircuitPublicInputs`, i.e. `end` and `end_non_revertible`.

The public teardown function is set on the `PublicKernelCircuitPublicInputs` as `public_teardown_function_hash`.

If a transaction does not have any enqueued public functions, and does not have a

public teardown function, then the `PrivateKernelTail` is used instead of the `PrivateKernelTailToPublic`, and no partitioning is done.

Kernel Tracking

Gas and fees are tracked throughout the kernel circuits to ensure that users are charged correctly for their transactions.

Private Kernel Circuits Overview

On the private side, the ordering of the circuits is:

1. PrivateKernelInit
2. PrivateKernelInner
3. PrivateKernelTail or PrivateKernelTailToPublic

The structs are (irrelevant fields omitted):

Private Context Initialization

Whenever a private function is run, it has a `PrivateContext` associated with it, which is initialized in part from a `PrivateContextInputs` object.

The [gas settings that users specify](#) become part of the values in the `TxContext` within the `PrivateContextInputs` of the [entrypoint](#). These values are copied to the `PrivateCircuitPublicInputs`.

The same `TxContext` is provided as part of the `TxRequest` in the `PrivateKernelInitCircuitPrivateInputs`. This is done to ensure that the `TxContext` in the `PrivateCallData` (what was executed) matches the `TxContext` in the `TxRequest` (users' intent).

Private Kernel Init

The PrivateKernelInit circuit takes in a `PrivateCallData` and a `TxRequest` and outputs a `PrivateKernelCircuitPublicInputs`.

It must:

- check that the `TxContext` provided as in the `TxRequest` input matches the `TxContext` in the `PrivateCallData`
- copy the `TxContext` from the `TxRequest` to the `PrivateKernelCircuitPublicInputs.constants.tx_context`
- copy the `Header` from the `PrivateCircuitPublicInputs` to the `PrivateKernelCircuitPublicInputs.constants.historical_header`
- set the `min_revertible_side_effect_counter` if it is present in the `PrivateCallData`
- set the `fee_payer` if the `is_fee_payer` flag is set in the `PrivateCircuitPublicInputs`
- set the `public_teardown_function_hash` if it is present in the `PrivateCircuitPublicInputs`
- set the `combined_constant_data.global_variables` to zero, since these are not yet known during private execution

Private Kernel Inner

The PrivateKernelInner circuit takes in a `PrivateKernelData` and a `PrivateCallData` and ultimately outputs a `PrivateKernelCircuitPublicInputs`.

It must:

- set the `fee_payer` if the `is_fee_payer` flag is set in the

- set the `public_teardown_function_hash` if it is present in the `PrivateCircuitPublicInputs` (and is not set in the input `PrivateKernelData`)
- copy the constants from the `PrivateKernelData` to the `PrivateKernelCircuitPublicInputs.constants`

Private Kernel Tail

The `PrivateKernelTail` circuit takes in a `PrivateKernelData` and outputs a `KernelCircuitPublicInputs` (see diagram below).

This is only used when there are no enqueued public functions or public teardown functions.

It must:

- check that there are no enqueued public functions or public teardown function
- compute the gas used
 - this will only include DA gas *and* any gas specified in the `teardown_gas_allocations`
- ensure the gas used is less than the gas limits
- ensure that `fee_payer` is set, and set it in the `KernelCircuitPublicInputs`
- copy the constants from the `PrivateKernelData` to the `KernelCircuitPublicInputs.constants`

NOTE

Transactions without a public component can safely set their teardown gas allocations to zero. They are included as part of the gas computation in the private kernel tail for consistency (limits always include teardown gas allocations) and future-compatibility if we have a need for private teardown functions.

Private Kernel Tail to Public

The `PrivateKernelTailToPublic` circuit takes in a `PrivateKernelData` and outputs a `PublicKernelCircuitPublicInputs` (see diagram below).

This is only used when there are enqueued public functions or a public teardown function.

It must:

- check that there are enqueued public functions or a public teardown function
- partition the side effects produced during private execution into revertible and non-revertible sets of `PublicAccumulatedData`
- compute gas used for the revertible and non-revertible. Both sets can have a DA component, but the revertible set will also include the teardown gas allocations the user specified (if any). This ensures that the user effectively pre-pays for the gas consumed in teardown.
- ensure the gas used (across revertible and non-revertible) is less than the gas limits
- ensure that `fee_payer` is set, and set it in the `PublicKernelCircuitPublicInputs`
- set the `public_teardown_call_request` in the `PublicKernelCircuitPublicInputs`
- copy the constants from the `PrivateKernelData` to the `PublicKernelCircuitPublicInputs.constants`

Mempool/Node Validation

A `Tx` broadcasted to the network has:

```

Tx {
    /**
     * Output of the private kernel circuit for this tx.
     */
    data: PrivateKernelTailCircuitPublicInputs,
    /**
     * Proof from the private kernel circuit.
     */
    proof: Proof,
    /**
     * Encrypted logs generated by the tx.
     */
    encryptedLogs: EncryptedTxL2Logs,
    /**
     * Unencrypted logs generated by the tx.
     */
    unencryptedLogs: UnencryptedTxL2Logs,
    /**
     * Enqueued public functions from the private circuit to be run
     * by the sequencer.
     * Preimages of the public call stack entries from the private
     * kernel circuit output.
     */
    queuedPublicFunctionCalls: PublicCallRequest[],
    /**
     * Public teardown function from the private circuit to be run
     * by the sequencer.
     * Preimage of the public teardown function hash from the
     * private kernel circuit output.
     */
    publicTeardownFunctionCall: PublicCallRequest,
}

```

Where the `PrivateKernelTailCircuitPublicInputs` may be destined for the base rollup (if there is no public component), or the public kernel circuits (if there is a public component).

Regardless, it has a `fee_payer` set.

When a node receives a transaction, it must check that:

1. the `fee_payer` is set
2. the `fee_payer` has a balance of [Fee Juice](#) greater than the computed [transaction fee](#) if the transaction has no public component
3. the `fee_payer` has a balance of FPA greater than the computed [max transaction fee](#) if the transaction has a public component

See other [validity conditions](#).

Public Kernel Circuits

On the public side, the order of the circuits is:

1. `PublicKernelSetup`
2. `PublicKernelAppLogic`
3. `PublicKernelTeardown`
4. `PublicKernelTail`

The structs are (irrelevant fields omitted):

Public Context Initialization

Whenever a public function is run, it has a `PublicContext` associated with it, which is initialized in part from a `PublicContextInputs` object.

The sequencer must provide information including the current `gas_fees`, the current `gas_left`, and the `transaction_fee`, but we cannot trust these values to be correct: we must compute the correct values in the public kernel circuits, and validate that the

sequencer provided the correct values.

Further, the sequencer is only obligated to provide the `transaction_fee` to the teardown function, as that is the only point at which the transaction fee can be known.

Public Circuit Public Inputs

The "outputs" of the public functions are coming from the public VM.

Therefore, once we verify that the `start_gas_left` which the sequencer provided is correct, we can trust the `end_gas_left` that the public VM reports.

Further, we can trust that the `transaction_fee` the public VM reported is the one which was made available to the public functions during teardown (though we must verify that the sequencer provided the correct value).

The `PublicCircuitPublicInputs` include the `global_variables` as injected via the `PublicContextInputs`. The first public kernel circuit to run, regardless of whether it is a setup, app, or teardown kernel, is responsible for setting its `constant_data.global_variables` equal to these. All subsequent public kernel circuit runs must verify that the `global_variables` from the `PublicCircuitPublicInputs` match the ones from the previously set `constant_data.global_variables`.

Public Kernel Setup

The `PublicKernelSetup` circuit takes in a `PublicKernelData` and a `PublicCallData` and outputs a `PublicKernelCircuitPublicInputs`.

It must assert that the `revert_code` in the `PublicCircuitPublicInputs` is equal to zero.

It must assert that the

`public_call.call_stack_item.public_inputs.global_variables.gas_fees` are valid according to the update rules defined.

It must compute the gas used in the `PublicKernelData` provided, and verify that the `gas_limits` in the `PublicKernelData`'s `TxContext` minus the computed `gas_used` is equal to the `start_gas_left` specified on the `PublicCircuitPublicInputs`.

This ensures that the public VM was provided with the correct starting gas values.

It must update the gas used in `end_non_revertible` as:

```
# assume the previous PublicKernelCircuitPublicInputs was copied to
circuit_outputs
pub fn update_non_revertible_gas_used(public_call: PublicCallData,
circuit_outputs: &mut PublicKernelCircuitPublicInputsBuilder) {
    let tx_gas_limits =
        circuit_outputs.constants.tx_context.gas_settings.gas_limits;
    let call_gas_left =
        public_call.call_stack_item.public_inputs.end_gas_left;
    let accum_end_gas_used = circuit_outputs.end.gas_used;

    circuit_outputs.end_non_revertible.gas_used = tx_gas_limits
        .sub(call_gas_left)
        .sub(accum_end_gas_used);
}
```

ⓘ GLOBAL GAS LIMIT FOR ALL ENQUEUED PUBLIC CALLS

Within the AVM, users may specify gas limits for each public function call. This does not apply to the "top-level" enqueued call: they all pull from the same global gas limit, and there is no way to "catch" an "out of gas" at this top-level apart from reverting.

Public Kernel App Logic

The `PublicKernelAppLogic` circuit takes in a `PublicKernelData` and a `PublicCallData` and outputs a `PublicKernelCircuitPublicInputs`.

It must perform the same computation as the `PublicKernelSetup` regarding verification of the `start_gas_left` and the `gas_fees`.

It must check the `revert_code` in the `PublicCircuitPublicInputs`.

If the `revert_code` is zero

Instead of updating `end_non_revertible`, it must update `end` as:

```
# assume the previous PublicKernelCircuitPublicInputs was copied to
circuit_outputs
pub fn update_revertible_gas_used(public_call: PublicCallData,
circuit_outputs: &mut PublicKernelCircuitPublicInputsBuilder) {
    let tx_gas_limits =
        circuit_outputs.constants.tx_context.gas_settings.gas_limits;
    let call_gas_left =
        public_call.call_stack_item.public_inputs.end_gas_left;
    let accum_end_non_revertible_gas_used =
        circuit_outputs.end_non_revertible.gas_used;

    circuit_outputs.end.gas_used = tx_gas_limits
        .sub(call_gas_left)
        .sub(accum_end_non_revertible_gas_used);
}
```

If the revert_code is non-zero

All side effects from the revertible set are discarded.

It consumes all the gas left:

```
# assume the previous PublicKernelCircuitPublicInputs was copied to
circuit_outputs
pub fn update_revertible_gas_used(public_call: PublicCallData,
circuit_outputs: &mut PublicKernelCircuitPublicInputsBuilder) {
    let tx_gas_limits =
circuit_outputs.constants.tx_context.gas_settings.gas_limits;
    let accum_end_non_revertible_gas_used =
circuit_outputs.end_non_revertible.gas_used;

    circuit_outputs.end.gas_used = tx_gas_limits
        .sub(accum_end_non_revertible_gas_used);
}
```

It sets the revert_code in PublicKernelCircuitPublicInputs to 1.

GAS RESERVED FOR PUBLIC TEARDOWN

Recall in the [Private Kernel Tail to Public](#) circuit, the gas allocated for the public teardown function was included in the end gas used. This ensures that we have gas available for teardown even though app logic consumed all gas.

WARNING

Consuming all gas left in the event of revert creates incentives for the sequencer to arrange transactions such that they revert, which is suboptimal. Future improvements will likely address this by only consuming the gas that was actually used, even in the event of revert.

Public Kernel Teardown

The PublicKernelTeardown circuit takes in a `PublicKernelData` and a `PublicCallData` and outputs a `PublicKernelCircuitPublicInputs`.

It must perform the same computation as the PublicKernelSetup regarding verification of the `gas_fees`.

It must assert that the `start_gas_left` is equal to the `PublicKernelData`'s `public_inputs.constants.tx_context.gas_settings.teardown_gas_allocations`

It must also compute the gas used in the `PublicKernelData` provided, and the `transaction fee` using this computed value, then verify that the `transaction_fee` in the `PublicCircuitPublicInputs` is equal to the computed transaction fee.

This ensures that the public VM was provided with the correct transaction fee, and that teardown did not exceed the gas limits.

Handling reverts

Teardown is attempted even if the app logic failed.

The teardown kernel can see if the app logic failed by checking if `revert_code` in the `PublicKernelCircuitPublicInputs` is set to `1`.

It also has access to the `revert_code` reported by the AVM of the current call within `PublicCircuitPublicInputs`.

The interplay between these two `revert_code`s is as follows:

| Kernel
<code>revert_code</code> | current AVM
<code>revert_code</code> | Resulting Kernel
<code>revert_code</code> |
|------------------------------------|---|--|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 2 |
| 1 | 1 | 3 |
| 2 or 3 | (any) | (unchanged) |

Rollup Kernel Circuits

The base rollup kernel circuit takes in a `KernelData`, which contains a `KernelCircuitPublicInputs`, which it uses to compute the `transaction_fee`.

Additionally, it verifies that the max fees per gas specified by the user are greater than the current block's fees per gas. It also verifies the `constant_data.global_variables.gas_fees` are correct.

After the public data writes specific to this transaction have been processed, and a new tree root is produced, the kernel circuit injects an additional public data write based upon that root which deducts the transaction fee from the `fee_payer`'s balance.

The calculated transaction fee is set as output on the base rollup as `accumulated_fees`. Each subsequent merge rollup circuit sums this value from both of its inputs. The root rollup circuit then uses this value to set the `total_fees` in the `Header`.

Fee Schedule

The [transaction fee](#) is comprised of a DA component, an L2 component, and an inclusion fee. The DA and L2 components are calculated by multiplying the gas consumed in each dimension by the respective `feePerGas` value. The inclusion fee is a fixed cost associated with the transaction, which is used to cover the cost of verifying the encompassing rollup proof on L1.

DA Gas

DA gas is consumed to cover the costs associated with publishing data associated with a transaction.

These data include:

- new note hashes
- new nullifiers
- new $I_2 \rightarrow I_1$ message hashes
- new public data writes
- new logs
- protocol metadata (e.g. the amount of gas consumed, revert code, etc.)

The DA gas used is then calculated as:

```
DA_BYTES_PER_FIELD = 32
DA_GAS_PER_BYTE = 16
FIXED_DA_GAS = 512

# FIXED_DA_GAS covers the protocol metadata,
```

(i) NON-ZERO transaction_fees

A side effect of the above calculation is that all transactions will have a non-zero transaction_fee.

L2 Gas

L2 gas is consumed to cover the costs associated with executing the public VM, proving the public VM circuit, and proving the public kernel circuit.

It is also consumed to perform fixed, mandatory computation that must be performed per transaction by the sequencer, regardless of what the transaction actually does; examples are TX validation and updating state roots in trees.

The public vm has an [instruction set](#) with opcode level gas metering to cover the cost of actions performed within the public VM.

Additionally, there is a fixed cost associated with each iteration of the public VM (i.e. the number of enqueued public function calls, plus 1 if there is a teardown function), which is used to cover the cost of proving the public VM circuit.

The L2 gas used is then calculated as:

```
FIXED_L2_GAS = 512  
FIXED_AVM_STARTUP_L2_GAS = 1024
```

```
num_avm_invocations = (number of enqueued public function calls) +  
                      (is there a teardown function ? 1 : 0)  
  
l2_gas_used = FIXED_L2_GAS  
              + FIXED_AVM_STARTUP_L2_GAS * num_avm_invocations
```

L2 Gas from Private

Private execution also consumes L2 gas, because there is still work that needs to be performed by the sequencer correspondent to the private outputs, which is effectively L2 gas. The following operations performed in private execution will consume L2 gas:

- 32 L2 gas per note hash
- 64 L2 gas per nullifier
- 4 L2 gas per byte of logs (note encrypted, encrypted, and unencrypted)

Max Inclusion Fee

Each transaction, and each block, has inescapable overhead costs associated with it which are not directly related to the amount of data or computation performed.

These costs include:

- verifying the private kernel proof of each transaction
- executing/proving the base/merge/root rollup circuits
 - includes verifying that every new nullifier is unique across the tx/block
 - includes processing I2→I1 messages of each transaction, even if they are empty (and thus have no DA gas cost)
 - includes ingesting I1→I2 messages that were posted during the previous block
 - injecting a public data write to levy the transaction fee on the `fee_payer`
- publishing the block header to the rollup contract on L1
 - includes verification of the rollup proof
 - includes insertion of the new root of the I2→I1 message tree into the L1

Outbox

- consumes the pending messages in the L1 Inbox
- publishing the block header to DA

See [the L1 contracts section](#) for more information on the L1 Inbox and Outbox.

Users cover these costs by [specifying an inclusion fee](#), which is different from other parameters specified in that it is a fixed fee offered to the sequencer, denominated in [Fee Juice](#).

Even though these line items will be the same for every transaction in a block, the cost to the sequencer will vary, particularly based on:

- congestion on L1
- prevailing price of proof generation

A price discovery mechanism is being developed to help users set the inclusion fee appropriately.

Published Gas & Fee Data

When a block is published to L1, it includes information about the gas and fees at a block-level, and at a transaction-level.

Block-level Data

The block header contains a `GlobalVariables`, which contains a `GasFees` object. This object contains the following fields:

- `feePerDaGas`: The fee in [Fee Juice](#) per unit of DA gas consumed for transactions in the block.
- `feePerL2Gas`: The fee in FPA per unit of L2 gas consumed for transactions in the block.

`GlobalVariables` also includes a `coinbase` field, which is the L1 address that receives the fees.

The block header also contains a `totalFees` field, which is the total fees collected in the block in FPA.

Updating the `GasFees` Object

Presently, the `feePerDaGas` and `feePerL2Gas` are fixed at 1 FPA per unit of DA gas and L2 gas consumed, respectively.

In the future, these values may be updated dynamically based on network conditions.

GAS TARGETS

Should we move to a 1559-style fee market with block-level gas targets, there is an interesting point where gas "used" presently includes the entire `teardown_gas_allocation` regardless of how much of that allocation was spent. In the future, if this becomes a concern, we can update our accounting to reflect the true gas used for the purposes of updating the `GasFees` object, though the user will be charged the full `teardown_gas_allocation` regardless.

Transaction-level Data

The transaction data which is published to L1 is a `TxEffects` object, which includes

- `transaction_fee`: the fee paid by the transaction in FPA

Nodes and Actors

To analyze the suitability of different node types for various actor types in the system, let's break down the node types and actor types, considering the characteristics, preferences, abilities, and potential motivations of each actor. This will help determine which actors are likely to run which nodes and how they might interact with the Private eXecution Environment (PXE).

Background

Before diving into Aztec specific data, we take a look at general blockchain nodes. It should help us have a shared base, which will later be useful for comparing the Aztec setups, where multiple nodes are involved due to its construction.

Node Types

The below node types are ordered in resource requirements, and exists for most blockchains. All the nodes participate in the peer-to-peer (p2p) network but with varying capacity.

1. Light Node:

- Download and validate headers from the p2p network.
 - Sometimes an "ultra-light" node is mentioned, this is a node that don't validate the headers it receive but just accept it. These typically are connected to a third party trusted by the user to provide valid headers.
- Stores only the headers.
- Querying any state not in the header is done by requesting the data from a third party, e.g. Infura or other nodes in the p2p network. Responses are validated with the headers as a trust anchor.

- Storage requirements typically measured in MBs (< 1GB).
- Synchronization time is typically measured in minutes.

2. Full Node:

- Receive and validate blocks (header and body) from the p2p network.
- Stores the complete active state of the chain
- Typically stores recent state history (last couple of hours is common)
- Typically stores all blocks since genesis (some pruning might be done)
- Can respond to queries of current and recent state
- Storage requirements typically measured in GBs (< 1TB)
- Synchronization time is typically measured in hours/days.

3. Archive Node:

- Receive and validate blocks (header and body) from the p2p network
- Stores the full state history of the chain
- Stores all blocks since genesis
- Can respond to queries of state at any point in time
- Storage requirements typically measured in TBs
- Synchronization time is typically measured in hours/days.

Beyond these node types, there are also nodes that participate in the block production who rely on full or archive nodes to extend the chain. In Ethereum these are called validators, but really, any of the nodes above do some validation of the blocks or headers. The Ethereum "validator" is really just their block producer.

Block production can generally be split into two parts: 1) building the block and 2) proposing the block.

❗ PROPOSER-BUILDER-SEPARATION (PBS)

When these two parts are split between different parties it is often referred to as Proposer-Builder-Separation.

A proposer generally have to put up some value to be able to propose a block and earn rewards from the blocks proposed. In PoW this value is burnt *in electricity, and in PoS it is staked* which can be "slashed" according to the rules of the chain.

In the Ethereum world you can say the "validator" is a proposer, that either builds his own blocks or outsource it to someone else, such as [Flashbots](#).

BLOBS

Blobs in Ethereum is a special kind of data that is proven to be available at time of publication and for a short period after. After this period (~18 days), the blob is considered shared and can be pruned from the chain. It is not needed to synchronize a new node.

Blobs will likely be stored by certain "super-archive" nodes, but not by all archive nodes. Meaning that the set of blob-serving nodes likely will be small. As the blob-hash is part of the block header, it is easy to validate that some chunk of data was indeed the published blob. Relies on an 1/n honesty assumption for access to the blob.

Actor Types

1. Mainstream Users:

- Currently don't care about technicalities, just want to use the system.
- Will likely not run any type of node, unless it is bundled with their wallet so they don't even know it is there.
- Generally don't care about trusting Infura or other third parties blindly.

2. Enthusiast:

- More knowledgeable and interested in technology than the mainstream user.
- Willing to invest time and resources but not at a professional level.

- Likely to run a light node for the extra security.

3. Power Users:

- Technically proficient and deeply engaged.
- Likely to have the resources and motivation to run more demanding nodes, and potentially participate in block production.

4. Developers:

- Highly knowledgeable with technical skills.
- Interested in detailed state and historical data for development and testing.

5. Idealists:

- Want maximal autonomy, and are willing to invest resources to achieve it.
- Will rely entirely on their own nodes and resources for serving their queries, current or historical by running an Archive Node.
- Likely to run nodes that contribute directly to the blockchain's operation as part of block production.
- Possibly willing to store all blobs.

6. Institutions:

- Have significant resources (and potentially technical) capabilities.
- Interested in robust participation, and potentially participate in block production for financial gain.

Aztec Relevant

In the following section, we are following the assumption that Aztec is running on top of Ethereum and using some DA to publish its data.

Beyond the [state](#) of the chain an Aztec node also maintains a database of encrypted data and their tags - an index of tags. The index is used when a user wishes to retrieve notes that belong to them. The index is built by the node as it receives new blocks from the network.

If the node have the full index it can serve any user that wants to retrieve their notes. This will be elaborated further in [responding to queries](#). A node could be configured to only build and serve the index for a subset of the tags and notes. For example, a personal node could be configured to only build the index for the notes that belong to the owner based on their [tag derivation](#).

If the node is intended only for block production, it can skip the index entirely.

Synchronizing An Aztec Node

To synchronize an Aztec full- or archive-node, we need the blocks to build state and index. We have two main options for doing so:

- Aztec nodes which is running dependency-minimized (i.e. not relying on a third party for data), and agree with the bridge on what is canonical, can retrieve the headers directly from their Ethereum node and the matching bodies from their node on the DA network. If blobs are used for DA, both of these could be the same node. The node will build the state and index as it receives the blocks.
- Aztec nodes that are not running dependency-minimized can rely on third parties for retrieving state and block bodies and use Ethereum as a "trust-anchor". Here meaning that the Aztec node could retrieve the full state from IPFS and then validate it against the headers it receives from the Ethereum node - allowing a quick synchronization process. Since the index is not directly part of state, its validity cannot be validated as simple as the state. The index will have to be built from the blocks (validating individual decrypted messages can be done against the state).

An Aztec light-node is a bit different, because it does not store the actual state *nor* index, only the headers. This means that it will follow the same model as Ethereum light-nodes, where it can retrieve data by asking third parties for it and then validating that it is in the state (anchored by the headers stored).

! AZTEC-SPECIFIC ULTRA LIGHT NODES

For users following the validating bridge as the canonical rollup, the bridge IS the light node, so they can get away with not storing anything, and just download the header data from the bridge. Since the [archive](#) is stored in the bridge, an Aztec ultra-light node can validate membership proofs against it without needing the exact header (given sufficient membership proofs are provided). This essentially enable instant sync for ultra-light nodes → if you either run an Ethereum node or trust a third party.

Responding to Queries

In the following, we will briefly outline how the different type of Aztec node will respond to different kinds of queries. Namely we will consider the following queries:

- Requesting current public state and membership proofs
- Requesting historical public state and membership proofs
- Requesting current membership proof of note hash, nullifier, I1 to I2 message or contract
- Requesting historical membership proof of note hash, nullifier, I1 to I2 message or contract
- Requesting encrypted notes for a given tag

Light-Node

As mentioned, light-nodes will be retrieving data from a third party, and then validating it against the headers at its disposal. Note that if we have the latest archive hash, two extra inclusion proofs can be provided to validate the response against the archive as well without needing any other data.

If we don't care about other people seeing what data we are retrieving, we can simply ask the third parties directly. However, when dealing with private information (e.g.

notes and their inclusion proofs), you need to be careful about how you retrieve it - you need to use some form of private information retrieval (PIR) service.

The exact nature of the service can vary and have not been decided yet, but it could be some form of Oblivious Message Retrieval (OMR) service. This is a service that allows you to retrieve a message from a database without revealing which message you are retrieving. An OMR service could be run by a third party who runs a full or archive node and is essentially just a proxy for other nodes to fetch information from the database. There might exist multiple OMR services, some with access to historical state and some without.

Assuming that OMR is used, the user would occasionally use the light-node to request encrypted notes from the OMR service. These notes can be decrypted by the user, and used to compute the note hash, for which they will need an inclusion proof to spend the note. When the user wish to spend the note, they can request the inclusion proof from the OMR service, which they can then validate against the headers they have stored.

NOTE DISCOVERY ISSUE

The issue of how to discover which notes belong to you is not solved by the Aztec protocol currently, and we have a [Request For Proposals](#) on the matter.

Full-Node

Can satisfy any query on current data, but will have to retrieve historical membership proofs from a third party or recompute the requested state based on snapshots and blocks.

Can utilize the same protocol as the Light-node for retrieving data that it doesn't already have.

Archive-Node

With a fully synched archive node you can respond to any query using your own data - no third party reliance.

Private eXecution Environment (PXE)

The Aztec PXE is required to do anything meaningful for an end-user. It is the interface between the user and the Aztec network and acts as a private enclave that runs on the end-user's machine.

While it is responsible for executing contracts and building proofs for the user's transactions privately, it needs data from the node to do so. It can be connected to any of the above types of nodes, inheriting their assumptions. From the point of view of the PXE, it is connected to a node, and it does not really care which one.

When requesting encrypted notes from the node the PXE will decrypt it and store the decrypted notes in its own database. This database is used to build proofs for the user's transactions.

From most end users points of view, the PXE will be very close to a wallet.

Bundling the PXE with an Aztec ultra-light node

A deployment of the PXE could be bundled together with an aztec ultra-light node which in turn is connected to "some" Ethereum node. The ethereum node could simply be infura, and then the ultra-light node handles its connections to the OMR service separately.

This setup can be made very light-weight and easy to use for the end-user, who might not be aware that they are running a node at all.

The reasoning that the ultra-light node should be bundled with an internal ultra-light node by default instead of simply using an RPC endpoint is to avoid users "missing" the OMR services and leak private information.

If the user is running a node themselves, on a separate device or whatever, they could connect the PXE to that node instead of the bundled one.

Governance & Upgrades



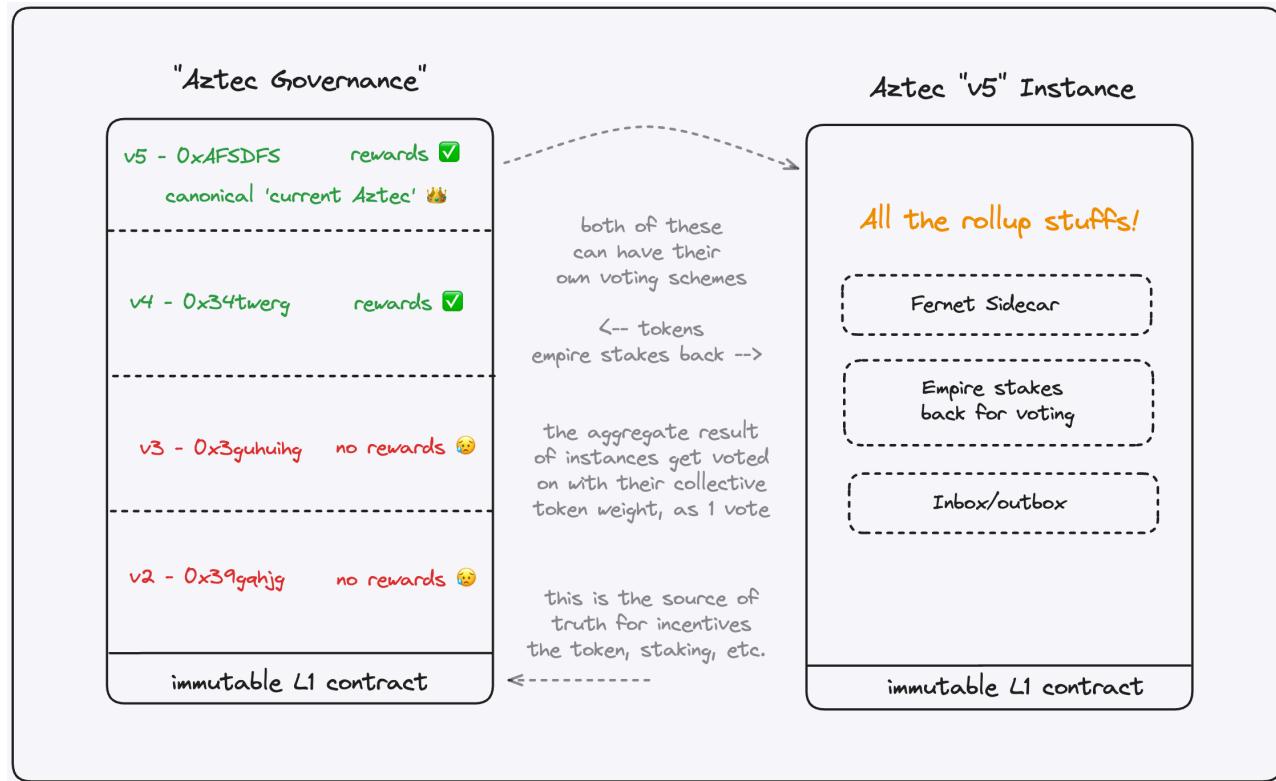
This is a first draft which articulates the latest thinking on governance & upgrades. It is subject to change and further review - ultimately needing team-wide understanding and approval. Please take this as a proposal, not as truth.

Summary

We propose an immutable governance & upgrade mechanism for The Aztec Network ("Aztec") that comprises a version registry, which points to deployments ("instances", used interchangeably) of Aztec.

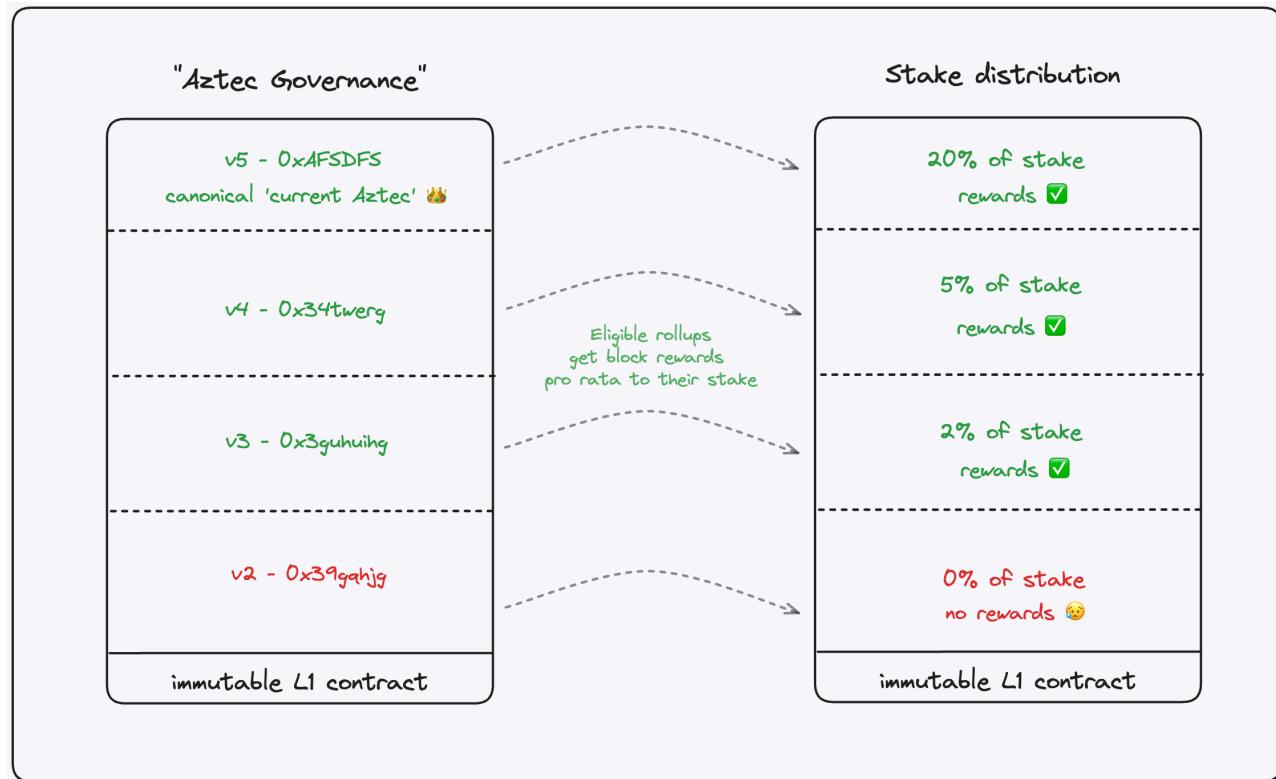
These instances may choose to be immutable themselves, or have governance that evolves over time alongside the community. The governance contract will keep track of governance votes, from the current version of Aztec, as well as direct token votes from the community, in order to provide some form of checks and balances.

The version registry will keep track of all historical versions of Aztec & provide them with incentives proportionate to their current stake. Additionally the governance contract will point to what the *current canonical* version of Aztec is, particularly relevant for 3rd parties to follow, such as centralized exchanges, or portals that wish to follow Aztec governance.



Rewards

We propose introducing a governance "version registry" which keeps track of a) which deployments of Aztec have been canonical, and b) which instances currently have tokens staked to them, specifically in order to issue a consistent, single new token in the form of *incentives* or "rollup/block rewards".



Given that deployments may be immutable, it is necessary to ensure that there are operators, i.e., sequencers & provers, running the infrastructure for a given deployment as long as users are interested in it. Therefore we suggest a model where all previous canonical instances of Aztec are rewarded pro-rata to their current proportion of stake.

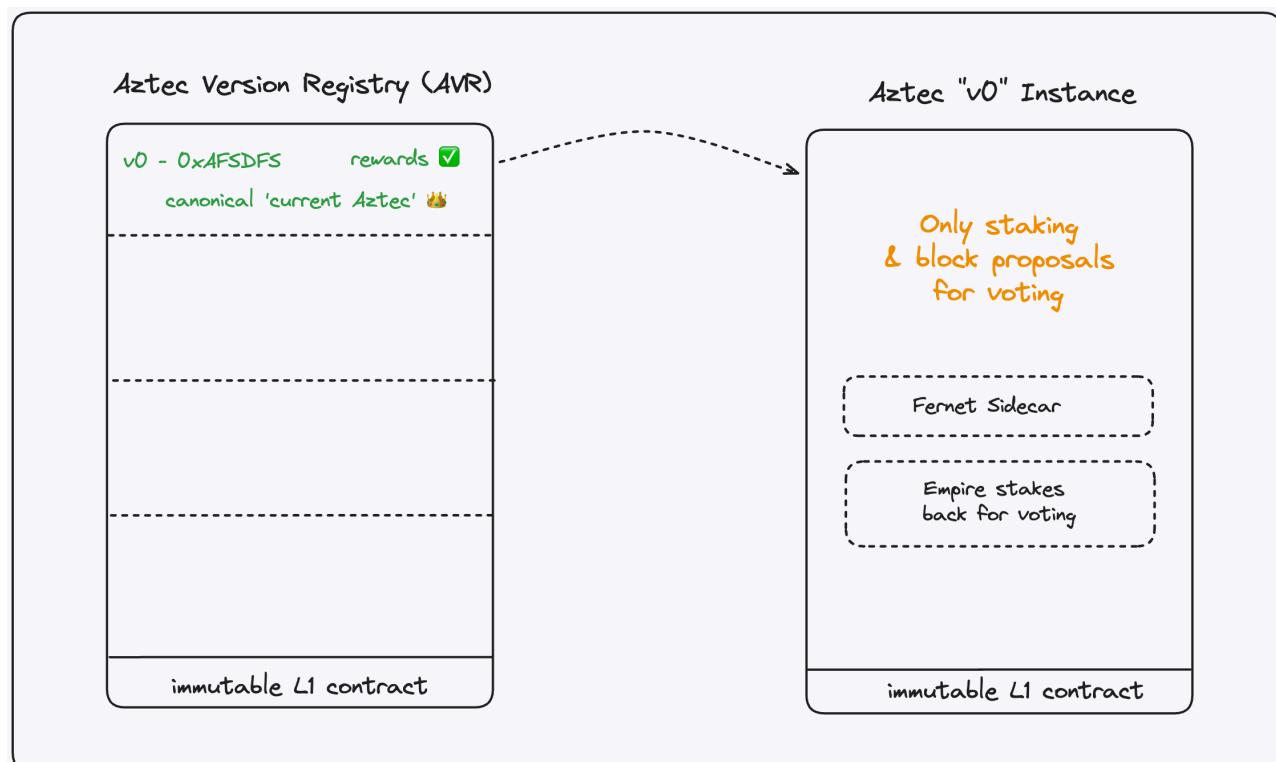
Beyond making it easier to understand for users, having a single token across all deployments is necessary to ensure that all instances are all utilizing the same token due to ecosystem cohesive and business development efforts, for example, having reliable onramps and wallets.

Initial deployment

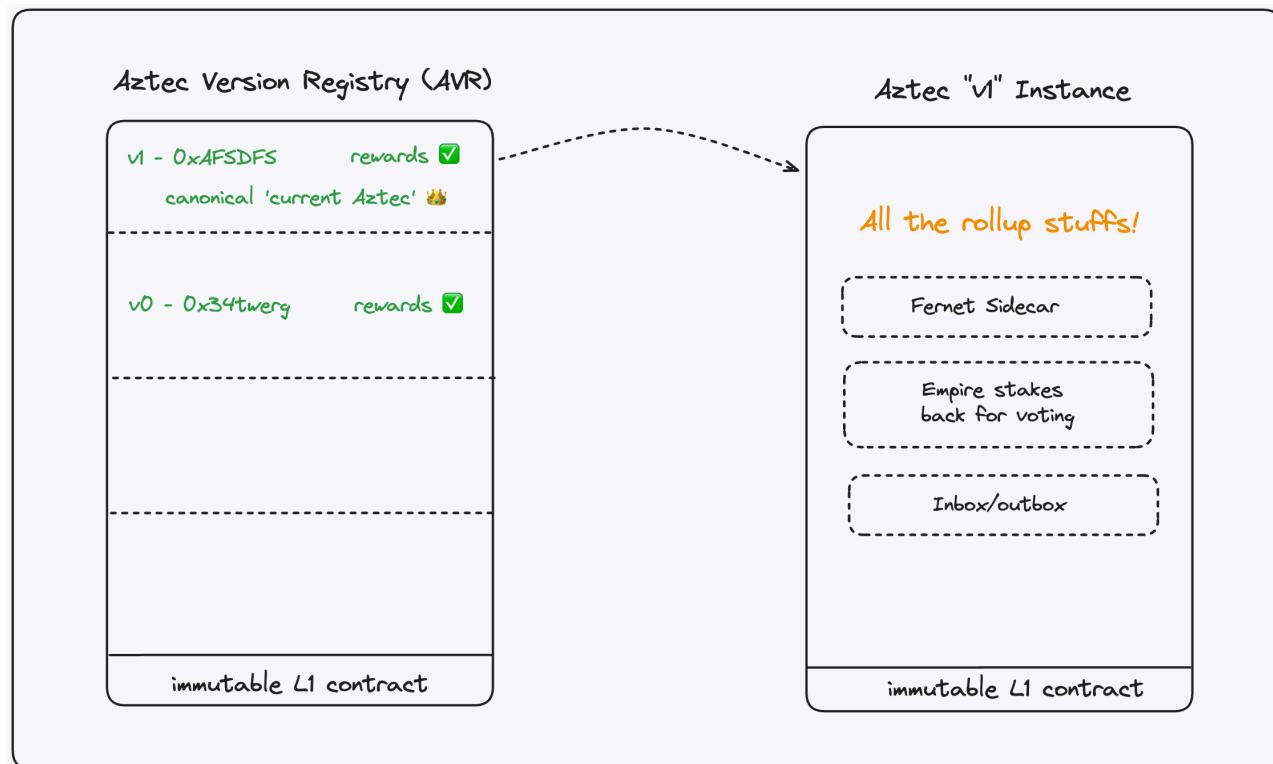
Upon initial deployment, there will be an immutable set of governance contracts which maintain the version registry, and an initial immutable instance of the rollup

which will be the first "canonical" deployment.

The initial instance will be called "Aztec v0" and (the current thinking is that v0) will not include the ability to process user transactions. Sequencers can register for Fernet's sequencer selection algorithm by staking tokens to that particular instance, and practice proposing blocks on mainnet prior to deciding to "go live" with v1, which does enable the processing of user transactions. This instance would then "restake" these tokens within the governance contract, to have a voting weight equal to the amount of tokens staked by its sequencer set. This is in order to ensure that the sequencer selection algorithm is working properly and the community of operators themselves can decide what happens to the network next, i.e., if it's ready to actually "go live" with transactions. It will also serve as a production readiness test of the upgradeability. In the event that these v0 tests are unable to be successfully completed as expected, the community (with potential foundation approval) may need to redeploy and try again.



The ability to upgrade to v1 is articulated below, and should follow a "happy path" upgrade where a majority of the v0 sequencer set must agree to upgrade by voting during their block proposals, similar to what was articulated in [the empire stakes back](#). Additionally, token holders can directly participate in the vote, or choose to delegate a vote with the weight of their tokens to another address, including the v0 rollup.



Proposing a new version

The current canonical rollup ("current rollup") can at any point propose voting on a new instance to become canonical and added to the governance version registry contracts. It can have its own logic for determining when it makes sense to do so, and trigger the formal governance vote. In the initial deployment it's expected to be done as articulated in the empire stakes back, where a sequencer must flag a desire to upgrade signal as part of Fernet's proposal phase, i.e., they won a random leader

election, and a majority of sequencers must do so over a specific time horizon, e.g., 7 days.

In addition to the current rollup implementation deciding to propose a vote, token holders can lock a sufficient amount of tokens for a sufficient amount of time in order to bypass the current rollup and propose a new version to become canonical next. This can be used in the scenario that the rollup implementation is so buggy it is unable to propose a new rollup to replace itself, or is due to potential community disagreement. In this scenario of disagreement, it is likely to be a very contentious action - as it implies a large token holder actively disagrees with the current rollup's sequencer set.

- Current thinking is this would require locking 1% of *total supply* for 2 years.
- These tokens must be eligible for voting, as defined below.

In a worst case scenario, the rollup's sequencer set could be malicious and censor potentially honest upgrade proposals from going through. In this scenario, there needs to be the ability to add a proposal "to the queue" via the token locking mechanism articulated above which is guaranteed to be executed when the previous vote completes.

Quorum

For any proposal to be considered valid and ready for voting, there must be a quorum of voting power eligible to participate. The purpose of this is to ensure that the network cannot be upgraded without a minimum of participation, keeping the governance more protected from takeover at genesis.

The exact amount of voting power required is to be determined through modelling, but we expect that around 5% of total supply. Assuming that 20% of the supply is circulating at launch and that 25% of this is staked towards the initial instance, this would allow the initial instance to reach quorum on its own.

Voting

Participation

Aztec's governance voting occurs within the governance contract, and the tokens being utilized must be "locked within governance" i.e., non-transferable.

Any token holder is able to directly vote via an interaction with the governance contract. Specifically, this includes those with locked, non-circulating tokens. The "ballot" is a simple yes/no/abstain vote on a proposal, and the amount of tokens being voted with. Note that this allows the same actor to vote multiple times, and even vote both yes and no with shares of their power. This allows for a more nuanced vote for contracts that control power for multiple users, such as a DAO, a rollup instance or a portal.

The current canonical rollup can choose to implement its internal voting however it would like, with the weight of the tokens staked in that instance. This is likely to be a majority of voting weight, which we can reliably assume will vote each time.

Generally this addresses the problems of low token holder participation! In the initial instance, we envision a version of the Empire Stakes back, where sequencers are voting during part of their block proposal phases. Not all sequencers will win a block proposal/election during the time period of the vote, this leads it to being a randomized sampling of the current sequencer set.

Exiting

The duration of the token lock depends on the action a user participated in. Tokens that have been locked to vote "yes" to changing the canonical instance are locked within the governance contract until the "upgrade" has been performed *or* when the voting period ends without the proposal gaining sufficient traction to reach quorum.

Tokens whose power did not vote "yes" are free to leave whenever they chose. This ensures that it is always possible to "ragequit" the governance if they disagree with an upgrade, and use or exit from the instance they are using.

Rollup instances themselves will need to deposit their stake into the governance, in order to earn rewards and participate within the vote. Further, they can apply their own enter/exit delays on top of the governance contract's. For example to ensure stability of the sequencer set over short timeframes, if using \$AZTC stake as a requirement for sequencing, they may wish to impose longer entry and exit queues.

Results

A vote is defined as passing if a majority of the voting weight votes "yes" to the proposal.

If the vote fails, there is no action needed.

If the vote passes, and a new rollup has been determined to be the next canonical instance, it will become canonical in the amount of days defined within the vote's timelock. It is likely there are defined limitations around this parameter, e.g., it must be a 3-30 day timelock. This is explained more in the timing section below. At this block height, portals that desire to follow governance should start referencing the new canonical instance to ensure as many bridged assets are backed on the latest version as possible.

DANGER

Portals that blindly follow governance inherently assume that the new inbox/outbox will always be backwards compatible. If it is not, it might break the portals.

Timing

Phase 1 - Setup

After the current canonical rollup, or a sufficient number of tokens are locked in governance, there is a ~3-7 day preparation period where users get their tokens "ready" for voting. i.e., withdraw & deposit/lock for the vote, or choose a suitable delegate.

Phase 2 - Voting

After setup has completed, there is a 7-30 day (TBD) period during which votes can land on the governance contract. In practice, we envision a majority of this voting happening in the current canonical instance and the voting weight of the current canonical instance being sufficient to reach quorum without any additional token delegation.

Phase 3 - Execution Delay (Timelock)

If a vote passes, there is a timelocked period before it becomes the new canonical rollup. This specific time period must be more than a minimum, e.g., 3 days, but is defined by the current rollup and in v1 may be controlled by both the sequencers in a happy path, and an emergency security council in a worst case scenario (articulated [below](#)). In a typical happy path scenario, we suggest this is at least 30 days, and in an emergency, the shortest period possible. A maximum period may also be defined, e.g., 60 days to ensure that the network cannot be kept from upgrading by having a delay of 200 years.

! INFO

It is worth acknowledging that this configurability on upgrade delay windows will likely be flagged on L2 beat as a "medium" centralization risk, due to the ability to quickly upgrade the software (e.g., a vacation attack). Explicitly this decision could cause us to be labeled a "stage 1" rather than "stage 2" rollup. However, if a vote is reasonably long, then it should be fine as you can argue that the "upgrade period" is the aggregate of all 3 periods.

Diagrams

Importantly we differentiate between Aztec Governance, and the governance of a particular instance of Aztec. This diagram articulates the high level of Aztec Governance, specifically how the network can deploy new versions over time which will be part of a cohesive ecosystem, sharing a single token. In this case, we are not concerned with how the current canonical rollup chooses to implement its decision to propose a new version, nor how it implements voting. It can be reasonably assumed that this is a version of The Empire Stakes back, where a majority of the current rollup sequencers are agreeing to propose and want to upgrade.

Happy path

"Bricked" rollup proposals

In this diagram, we articulate the scenario in which the current canonical rollup contains bugs that result in it being unable to produce not only a block, but a vote of any kind. In this scenario, someone or a group (Lasse refers to as the "unbrick DAO") may lock 1% (specific # TBD) of total supply in order to propose a new canonical rollup. It is expected that this scenario is very unlikely, however, we believe it to be a nice set of checks and balances between the token holders and the decisions of the

current rollup implementation.

Vote Delegation

Any token holder can delegate their token's voting weight to another address, including the current canonical rollup's, if it wishes to follow along in that addresses' vote. The tokens being delegated will be locked, either within the governance contract or the vesting contract.

! INFO

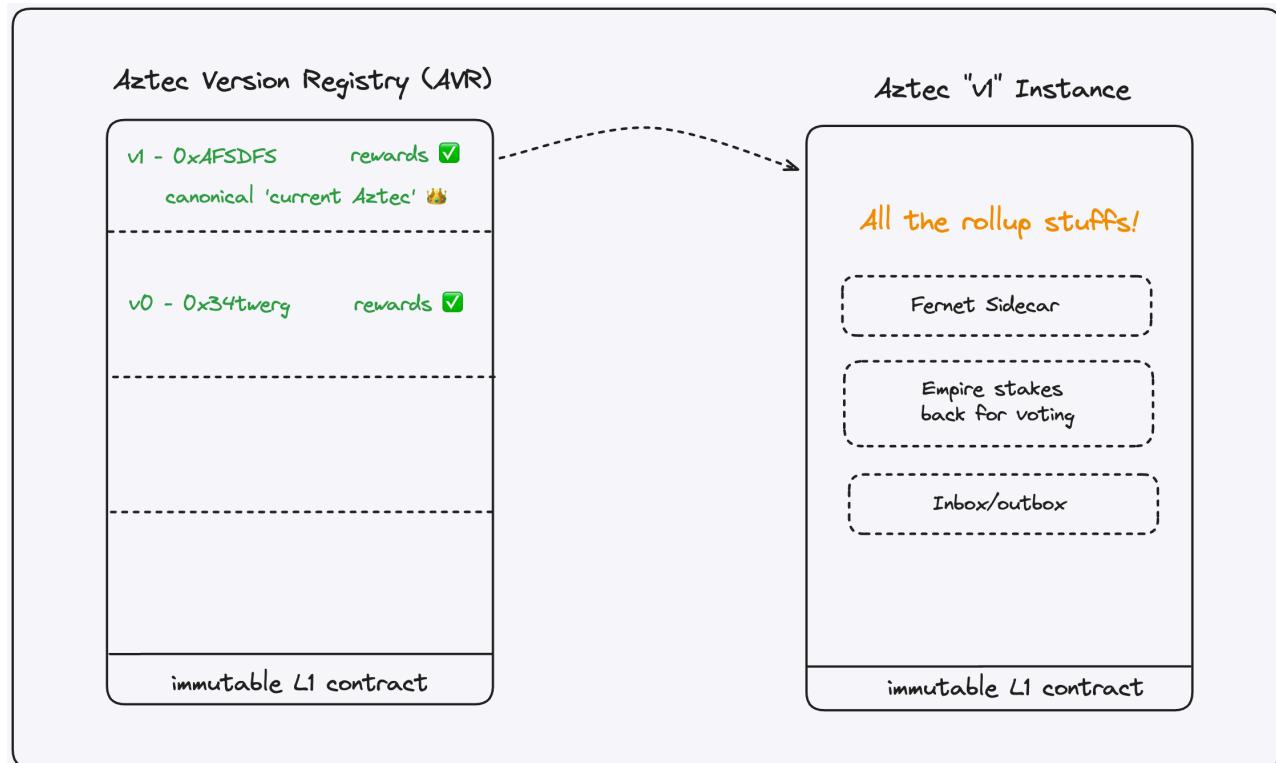
💡 Locked, non-circulating tokens can be delegated! This "economic training wheel" enables Aztec Labs, Foundation, and potential investors to participate responsibly in governance while the protocol is getting off the ground. It is TBD if these locked, non-circulating, delegated tokens will be able to earn incentives, i.e., block rewards.

The diagram below articulates calling `delegateTo(address)` on both the governance contract and specifying a particular address. Additionally calling `delegateTo()` on the current canonical rollup if you wish to align with whatever voting mechanism that system currently has in place.

Emergency mode

Emergency mode is proposed to be introduced to the initial instance "v0" or "v1" of Aztec, whatever the first instance or deployment is. Emergency mode will not be included as part of the canonical governance contracts or registry. If future deployments wish to have a similar security council, they can choose to do so. In this design, the current rollup can determine the timelock period as articulated above, within some predefined constraints, e.g., 3-30 days. Explicitly, the current rollup can give a security council the ability to define what this timelock period may be, and in

the case of a potential vulnerability or otherwise, may be well within its rights to choose the smallest value defined by the immutable governance contract to ensure that the network is able to recover and come back online as quickly as possible.



Unpausing by default

In the first instance, it's expected that this security council can *only* pause the rollup instance, not make any other changes to the instance's functionality. It is important that after N days (e.g., 180), or after another rollup has been marked canonical and Y days (e.g., 60), this rollup *must* become unpauseable eventually - otherwise it's practically bricked from the perspective of those users choosing immutable portals, and could leave funds or other things belonging to users (e.g., identity credentials or something wacky) permanently inside of it. The same is true for all future instances that have pause functionalities.

Removing the emergency mode

The emergency mode articulated here may be implemented as part of the next instance of Aztec - "v1" or whatever it ends up being called, when mainnet blocks are enabled. The current sequencer set on v0 (the initial instance) would then need to vote as outlined above on marking this new deployment as the "canonical v1" or predecessor to the initial instance. This would then have all of the portal contracts follow v1, which may or may not have other [training wheels](#). If the community wishes, they can always deploy a new instance of the rollup which removes the emergency mode and therefore the pause-only multisig.

Contract implementation

 DANGER

TO DO

Glossary

 DANGER

TO DO

Aztec Block Production

! INFO

This document aims to be the latest source of truth for the Fernet sequencer selection protocol, and reflect the decision to implement the [Sidecar](#) proving coordination protocol. Notably, this is written within the context of the first instance or deployment of Aztec. The definitions and protocol may evolve over time with each version.

Overview

This document outlines a proposal for Aztec's block production, integrating immutable smart contracts on Ethereum's mainnet (L1) to establish Aztec as a Layer-2 Ethereum network. Sequencers can register permissionlessly via Aztec's L1 contracts, entering a queue before becoming eligible for a random leader election ("Fernet"). Sequencers are free to leave, adhering to an exit queue or period. Roughly every 7-10 minutes (subject to reduction as proving and execution speeds stabilize and/or improve) sequencers create a random hash using [RANDAO](#) and their public keys. The highest-ranking hash determines block proposal eligibility. Selected sequencers either collaborate with third-party proving services or self-prove their block. They commit to a prover's L1 address, which stakes an economic deposit. Failure to submit proofs on time results in deposit forfeiture. Once L1 contracts validate proofs and state transitions, the cycle repeats for subsequent block production (forever, and ever...).

Full Nodes

Aztec full nodes are nodes that maintain a copy of the current state of the network.

They fetch blocks from the data layer, verify and apply them to its local view of the state. They also participate in the [P2P network](#) to disburse transactions and their proofs. Can be connected to a PXE which can build transaction witness using the data provided by the node (data membership proofs).

! INFO

We should probably introduce the PXE somewhere

| | Minimum | Recommended |
|---------|---------|-------------|
| CPU | 16cores | 32cores |
| Network | 32 mb/s | 128 mb/s |
| Storage | 3TB | 5TB |
| RAM | 32gb | 64gb |

! ESTIMATES

- CPU: Help
- Network: 40KB for a transaction with proof (see [P2P network](#)). Assuming gossiping grows the data upload/download 10x, ~400KB per tx. With 10 tx/s that's 4MB/s or 32mb/s.
- Storage: [~1548 bytes per transaction](#) + tree overhead, ~ 0.4 TB per year.
- RAM: Help

Sequencers

Aztec Sequencer's are full nodes that propose blocks, execute public functions and choose provers, within the Aztec Network. It is the actor coordinating state transitions and proof production. Aztec is currently planning on implementing a protocol called Fernet (Fair Election Randomized Natively on Ethereum trustlessly), which is permissionless and anyone can participate. Additionally, sequencers play a role participating within Aztec Governance, determining how to manage [protocol upgrades](#).

Hardware requirements

|  | Minimum | Recommended |
|---|---------|-------------|
| CPU | 16cores | 32cores |
| Network | 32 mb/s | 128 mb/s |
| Storage | 3TB | 5TB |
| RAM | 32gb | 64gb |

ESTIMATES

Mostly as full nodes. The network requirements might be larger since it needs to gossip the block proposal and coordinate with provers, depends on the number of peers and exact proving structure.

Provers

An Aztec Prover is a full node that is producing Aztec-specific zero knowledge (zk) proofs ([rollup proofs](#)). The current protocol, called [Sidecar](#), suggests facilitating out of protocol proving, similar to out of protocol [PBS](#). Provers in this case are fully permissionless and could be anyone - such as a vertically integrated sequencer, or a proving marketplace such as [nil](#), [gevulot](#), or [kalypso](#), as long as they choose to support the latest version of Aztec's proving system.

Hardware requirements

| | Minimum | Recommended |
|---------|---------|-------------|
| CPU | 16cores | 32cores |
| Network | 32 mb/s | 128 mb/s |
| Storage | 3TB | 5TB |
| RAM | 32gb | 64gb |

⚠ ESTIMATES

Mostly as full nodes. The compute and memory requirements might be larger since it needs to actually build the large proofs. Note, that the prover don't directly need to be a full node, merely have access to one.

Other types of network node

- [Validating Light nodes](#)

- Maintain a state root and process block headers (validate proofs), but do not store the full state.
 - The L1 bridge is a validating light node.
 - Can be used to validate correctness of information received from a data provider.
- [Transaction Pool Nodes](#)
 - Maintain a pool of transactions that are ready to be included in a block.
- Archival nodes
 - A full node that also stores the full history of the network
 - Used to provide historical membership proofs, e.g., prove that x was included in block n .
 - In the current model, it is expected that there are standardized interfaces by which well known sequencers, i.e., those operated by well respected community members or service providers, are frequently and regularly uploading historical copies of the Aztec state to immutable and decentralized storage providers such as: IPFS, Filecoin, Arweave, etc. The specific details of such is TBD and likely to be facilitated via RFP.
- Help figure it out by submitting a proposal on the [Aztec research forum](#)!

Registration

Sequencers must stake an undetermined amount of a native token on Layer-1 to join the protocol, reflecting Aztec's economic security needs. For consensus based network, they enter an *entryPeriod* before becoming active. This period aims to provide stability (and predictability) to the sequencer set over short time frames which is desirable for PoS based consensus networks when progressing blocks. For non-consensus based networks such as the initial Fernet implementation, an *entryPeriod* can be used for limiting the ability to quickly get outsized influence over governance decisions, but is not strictly necessary.

❗ INFO

What is Aztec's economic security needs? Please clarify.

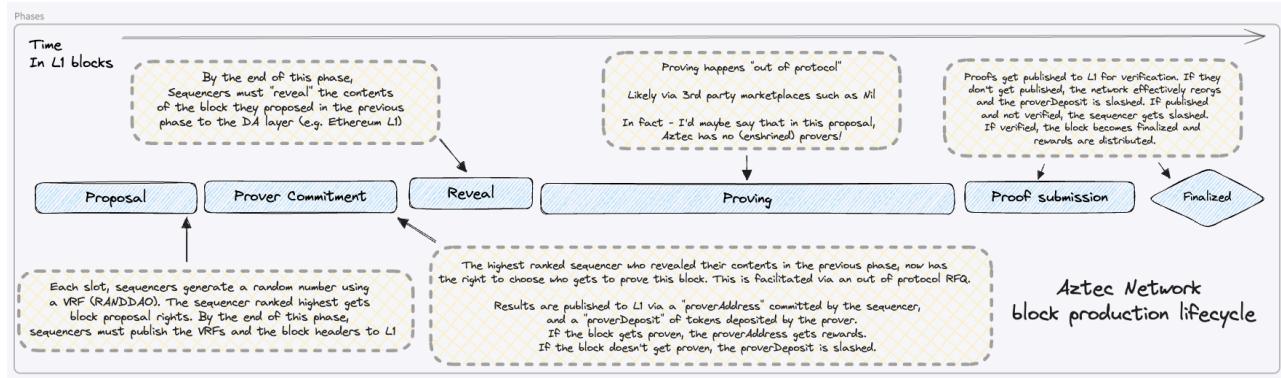
Currently, Provers don't need to register but must commit a bond during the **prover commitment phase** articulated below. This ensures economic guarantees for timely proof generation, and therefore short-term liveness. If the prover is unable or unwilling to produce a proof for which they committed to in the allotted time their bond will be slashed.

Future updates may introduce a registration process for Provers, possibly leading to a smaller, more consistent group, but this is currently not suggested to be required.

Block production

🔥 TODO

- The diagram needs to be updated with respect to "VRF".
- In **Prover commitment** phase, it is not said what the signature is used for. I'm expecting that it is used to allow the prover to publish the message on behalf of the sequencer, but it is not made clear.
- In **Backup** phase, would be useful if we add a comment on the duration
- In **Diagram**
 - add a dedicated timeline from the block production's PoV
 - get rid of "pre-confirmed"



Every staked sequencers participate in the following phases, comprising an Aztec slot:

- 1. Proposal:** Sequencers generate a hash of every other sequencer's public keys and RANDAO values. They then compare and rank these, seeing if they possibly have a "high" ranking random hash. If they do, they may choose to submit a block proposal to Layer-1. The highest ranking proposal will become canonical.
- 2. Prover commitment:** After an off-protocol negotiation with the winning sequencer, a prover submits a commitment to a particular Ethereum address that has intentions to prove the block. This commitment includes a signature from the sequencer and an amount X of funds that get slashed if the block is not finalized.
- 3. Reveal:** Sequencer uploads the block contents required for progressing the chain to whatever DA layer is decided to be implemented, e.g., Ethereum's 4844 blobs.
 - It is an active area of debate and research whether or not this phase is necessary, without intentions to implement "build ahead" or the ability to propose multiple blocks prior to the previous block being finalized. A possible implementation includes a block reward that incentivizes early reveal, but does not necessarily require it - turning the ability to reveal the block's data into another form of potential timing game.
- 4. Proving:** The prover or prover network coordinates out of protocol to build the recursive proof tree. After getting to the last, singular proof that reflects the entire block's state transitions they then upload the proof of the block to the L1 smart contracts.

5. **Finalization:** The smart contracts verify the block's proof, which triggers payouts to sequencer and prover, and the address which submits the proofs (likely the prover, but could be anyone such as a relay). Once finalized, the cycle continues!
 - For data layers that is not on the host, the host must have learned of the publication from the **Reveal** before the **Finalization** can begin.
6. **Backup:** Should no prover commitment be put down, or should the block not get finalized, then an additional phase is opened where anyone can submit a block with its proof, in a based-rollup mode. In the backup phase, the first rollup verified will become canonical.

Constraining Randao

The `RANDAO` values used in the score as part of the **Proposal** phase must be constrained by the L1 contract to ensure that the computation is stable throughout a block. This is to prevent a sequencer from proposing the same L2 block at multiple L1 blocks to increase their probability of being chosen.

Furthermore, we wish to constrain the `RANDAO` ahead of time, such that sequencers will know whether they need to produce blocks or not. This is to ensure that the sequencer can ramp up their hardware in time for the block production.

As only the last `RANDAO` value is available to Ethereum contracts we cannot simply read an old value. Instead, we must compute update it as storage in the contract.

The simplest way to do so is by storing the `RANDAO` at every block, and then use the `RANDAO` for block number $-n$ when computing the score for block number n . For the first n blocks, the value could pre-defined.

 **INFO**

Updating the `RANDAO` values used is a potential attack vector since it can be

biased. By delaying blocks by an L1 block, it is possible to change the RANDAO value stored. Consider how big this issue is, and whether it is required to mitigate it.

Exiting

In order to leave the protocol sequencers can exit via another L1 transaction. After signaling their desire to exit, they will no longer be considered `active` and move to an `exiting` status.

When a sequencer move to `exiting`, they might have to await for an additional delay before they can exit. This delay is up to the instance itself, and is dependent on whether consensus is used or not and what internal governance the instance supports. Beware that this delay is not the same as the exit delay in [Governance](#).

🔥 DANGER

@lasse to clarify what semantics he would prefer to use here instead of exiting/
active

Lasse Comment: I'm unsure what you mean by "active" here. Is active that you are able to produce blocks? Is so, active seems fine. Also, not clear to me if `exiting` means that they be unable to propose blocks? If they are voting in them, sure put a delay in there, but otherwise I don't see why they should be unable to leave (when we don't have consensus for block production).

Confirmation rules

There are various stages in the block production lifecycle that a user and/or

application developer can gain insights into where their transaction is, and when it is considered confirmed.

Notably there are no consistent, industry wide definitions for confirmation rules. Articulated here is an initial proposal for what the Aztec community could align on in order to best set expectations and built towards a consistent set of user experiences/ interactions. Alternative suggestions encouraged!

Below, we outline the stages of confirmation.

1. Executed locally
2. Submitted to the network
 - users no longer need to actively do anything
3. In the highest ranking proposed block
4. In the highest ranking proposed block, with a valid prover commitment
5. In the highest ranking proposed block with effects available on the DA Layer
6. In a proven block that has been verified / validated by the L1 rollup contracts
7. In a proven block that has been finalized on the L1

Economics

In the current Aztec model, it's expected that block rewards in the native token are allocated to the sequencer, the prover, and the entity submitting the rollup to L1 for verification. Sequencers retain the block's fees and MEV (Maximal Extractable Value). A potential addition in consideration is the implementation of MEV or fee burn. The ratio of the distribution is to be determined, via modeling and simulation.

Future Aztec versions will receive rewards based on their staked amount, as determined by the Aztec governance and [incentives contracts](#). This ensures that early versions remain eligible for rewards, provided they have active stake. Changes to the network's economic structure, especially those affecting block production and

sequencer burden, require thorough consideration due to the network's upgrade and governance model relying on an honest majority assumption and at a credibly neutral sequencer set for "easy" proposals.

INFO

With the rest of the protocol *mostly* well defined, Aztec Labs now expects to begin a series of sprints dedicated towards economic analysis and modeling with [Blockscience](#) throughout Q1-Q2 2024. This will result in a public report and associated changes to this documentation to reflect the latest thinking.

MEV-boost

SUCCESS

About MEV on Aztec

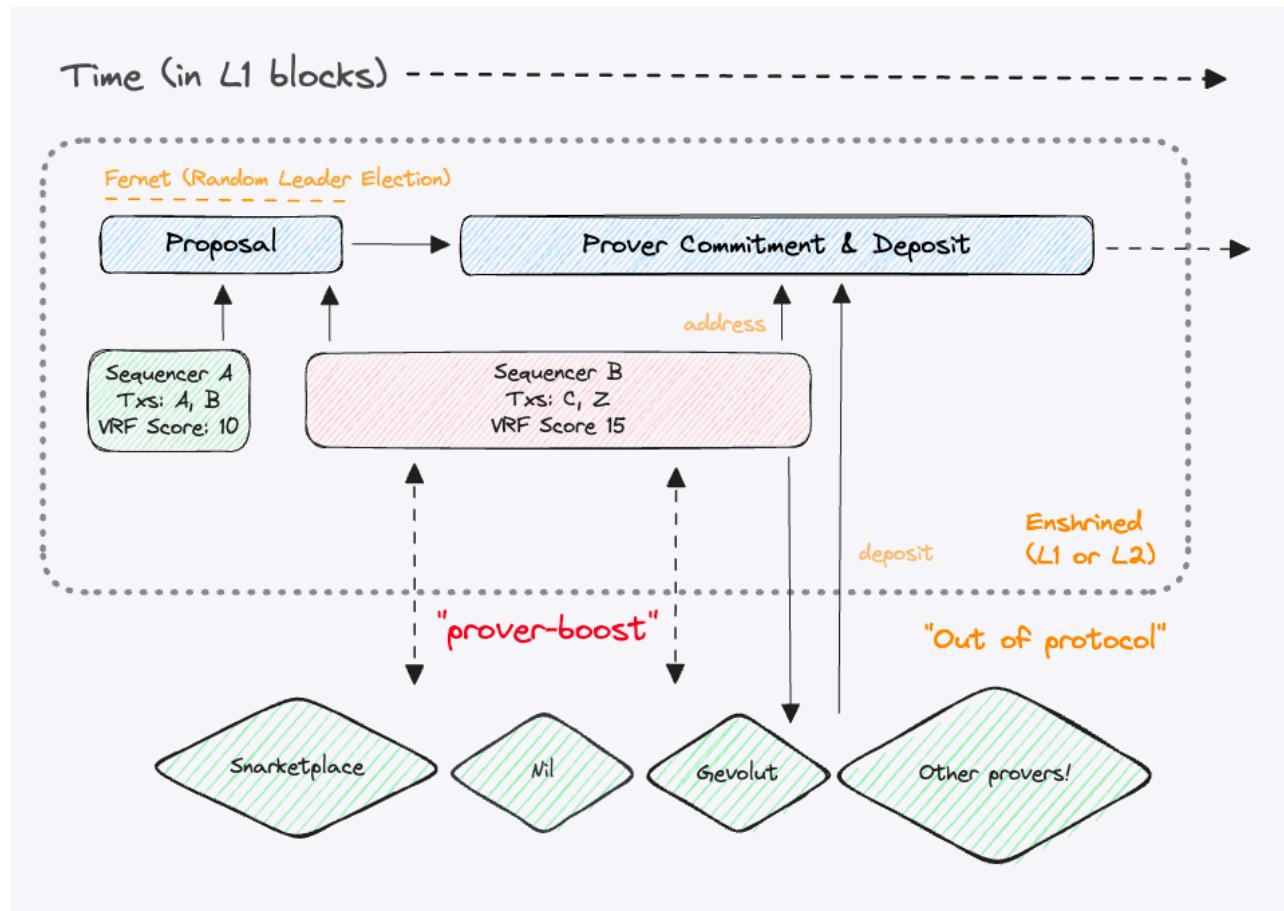
Within the Aztec Network, "MEV" (Maximal Extractable Value) can be considered "mitigated", compared to "public" blockchains where all transaction contents and their resulting state transitions are public. In Aztec's case, MEV is *generally* only applicable to [public functions](#) and those transactions that touch publicly viewable state.

It is expected that any Aztec sequencer client software will initially ship with some form of first price or priority gas auction for transaction ordering. Meaning that in general, transactions paying higher fees will get included earlier in the network's transaction history. Similar to Layer-1, eventually an opt-in, open source implementation of "out of protocol proposer builder separation" (PBS) such as [mev-boost](#) will likely emerge within the community, giving sequencers an easier to access way to earn more money during their periods as sequencers. This is an active area of research.

Proof-boost

It is likely that this proving ecosystem will emerge around a [flashbots mev-boost][<https://boost.flashbots.net/>] like ecosystem, specifically tailored towards the needs of sequencers negotiating the cost for a specific proof or set of proofs. Currently referred to as `proof-boost` or `goblin-boost` (due to goblin plonk..).

Specifically, Proof boost is expected to be open source software sequencers can optionally run alongside their clients that will facilitate a negotiation for the rights to prove this block, therefore earning block rewards in the form of the native protocol token. After the negotiation, the sequencer will commit to an address, and that address will need to put up an economic commitment (deposit) that will be slashed in the event that the block's proofs are not produced within the allotted timeframe.



Initially it's expected that the negotiations and commitment could be facilitated by a trusted relay, similar to L1 block building, but options such as onchain proving pools are under consideration. Due to the out of protocol nature of [Sidecar](#), these designs can be iterated and improved upon outside the scope of other Aztec related governance or upgrades - as long as they maintain compatibility with the currently utilized proving system(s). Eventually, any upgrade or governance mechanism may choose to enshrine a specific well adopted proving protocol, if it makes sense to do so.

Diagrams

Happy path

🔥 TODO

I'm not fully understanding the different groups, is the aztec network just the node software or ?? Maybe coloring is nice to mark what is contracts and entities or groups of entities. Otherwise seems quite nice.

Voting on upgrades

In the initial implementation of Aztec, sequencers may vote on upgrades alongside block proposals. If they wish to vote alongside an upgrade, they signal by updating their client software or an environment configuration variable. If they wish to vote no or abstain, they do nothing. Because the "election" is randomized, the voting acts as a random sampling throughout the current sequencer set. This implies that the specific duration of the vote must be sufficiently long and RANDAO sufficiently randomized to ensure that the sampling is reasonably distributed.

Backup mode

In the event that no one submits a valid block proposal, we introduce a "backup" mode which enables a first come first serve race to submit the first proof to the L1 smart contracts.

🔥 DANGER

There is an outstanding concern that this may result in L1 censorship. L1 builders may choose to not allow block proposals to land on the L1 contracts within a sufficient amount of time, triggering "backup" mode - where they could have a

block pre-built and proven, waiting L1 submission at their leisure. This scenario requires some careful consideration and modeling. A known and potential mitigation includes a longer proposal phase, with a relatively long upper bounds to submit a proposal. Given that all sequencers are able to participate, it's effectively a "priority ranked race" within some form of "[timing game](#)".

We also introduce a similar backup mode in the event that there is a valid proposal, but no valid prover commitment (deposit) by the end of the prover commitment phase.

Glossary



DANGER

TO DO - define the things

P2P Network

Requirements for a P2P Network

When a rollup is successfully published, the state transitions it produces are published along with it, making them publicly available. This broadcasted state does not depend on the Aztec network for its persistence or distribution. Transient data however, such as pending user transactions for inclusion in future rollups, does rely on the network for availability. It is important that the network provides a performant, permissionless and censorship resistant mechanism for the effective propagation of these transactions to all network participants. Without this, transactions may be disadvantaged and the throughput of the network will deteriorate.

We can derive the following broad requirements of the P2P network.

1. Support a node count up to approximately 10000.
2. Enable new participants to join the network in a permissionless fashion.
3. Propagate user transactions quickly and efficiently, throughout the network.
4. Provide protection against DoS, eclipse and sybil attacks.
5. Support a throughput of at least 10 transactions per second.
6. Support transaction sizes of ~40Kb.
7. Minimise bandwidth requirements overall and on any given node.

Network Participants

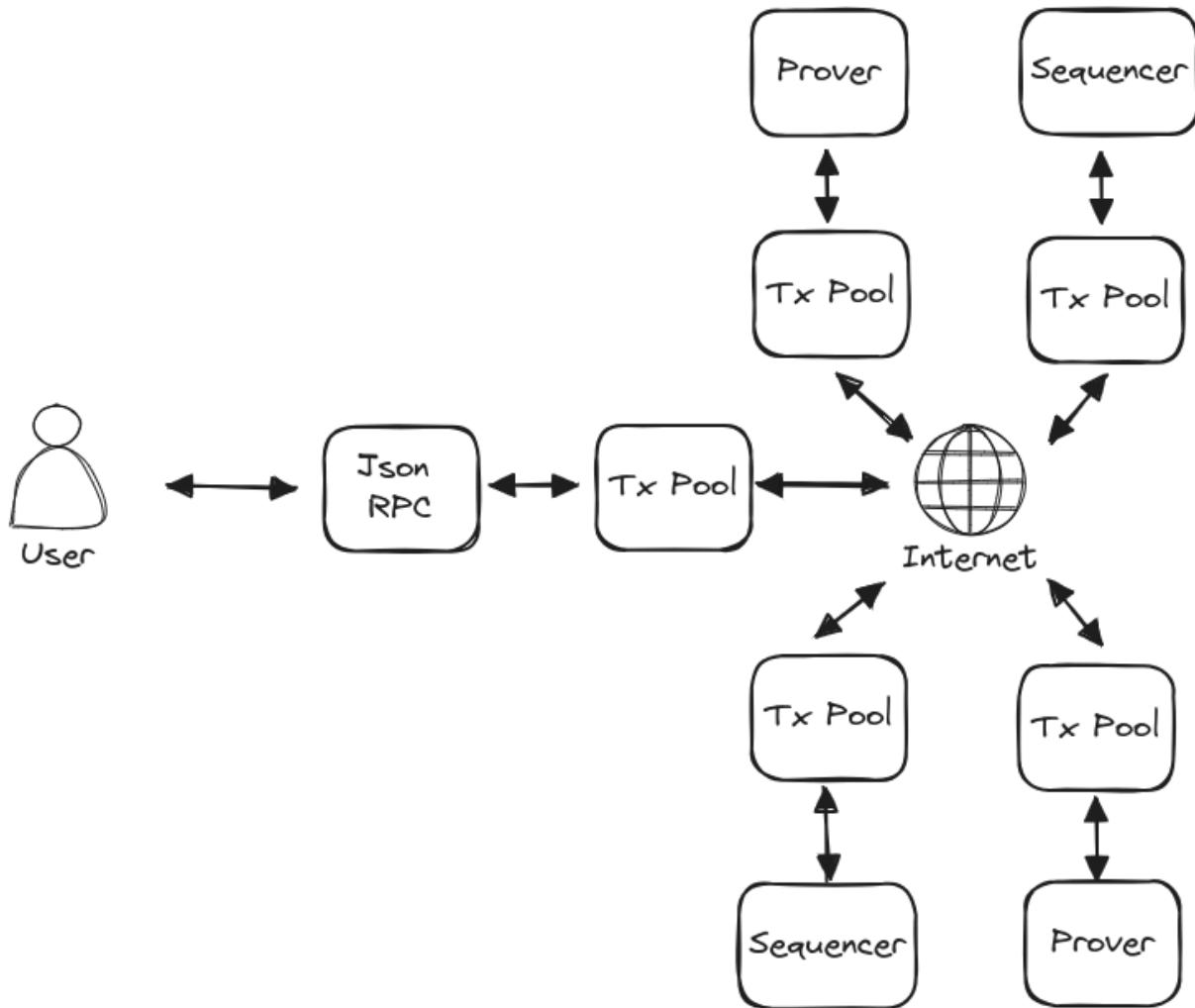
For the purpose of this discussion, we define the 'Aztec Network' as the set of components required to ensure the continual distribution of user transactions and production of rollups. The participants in such a network are:

- Sequencers - responsible for selecting transactions from the global pool and including them in rollups
- Provers - responsible for generating zk-proofs for the transaction and rollup circuits
- Transaction Pool Nodes - responsible for maintaining a local representation of the pending transaction pool
- Bootnodes - responsible for providing an entrypoint into the network for new participants

These participants will likely operate an instance of the [Aztec Node](#) configured for their specific purpose. The Transaction Pool Node listed above is intended to describe a node with the minimum amount of functionality required to fulfill the needs of a PXE. Namely access to the global transaction pool and an up-to-date instance of the [state](#).

Anyone can operate an instance of the Aztec Node configured to serve their needs, providing increased privacy and censorship resistance.

Client PXEs will interact with instances of the Aztec Node via it's JSON RPC interface.



Transaction Size

Transactions are composed of several data elements and can vary in size, determined largely by the private kernel proof and whether the transaction deploys any public bytecode. A typical transaction data footprint is shown in the following table. Any deployed contract bytecode would be in addition to this.

| Element | Size |
|--|-------|
| Public Inputs, Public Calls and Emitted Logs | ~8KB |
| Private Kernel Proof | ~32KB |

Sequencer-to-Prover Communication

Proving is an out-of-protocol activity. The nature of the communication between sequencers and provers will depend entirely on the prover/s selected by the sequencer. Provers may choose to run their own Transaction Pool Node infrastructure so that they are prepared for generating proofs and don't need to receive this data out-of-band.

LibP2P

Aztec will build it's network on [LibP2P](#) and the suite of technologies that it contains. LibP2P has demonstrated it's capabilities as the set of protocols employed by Ethereum's consensus layer. Clients of the network will need to use a subset of LibP2P's protocols.

There will be 2 primary communication domains within the network:

1. Node Discovery
2. Transaction Gossip

Node Discovery

When new participants join the network for the first time, they will need to locate

peers. Ethereum's [DiscV5](#) is an adaptation of Kademlia, storing node records rather than content within its distributed hash table. From this, nodes are able to build what can be thought of as an address book of other participants.

DiscV5

Whilst the DiscV5 specification is still under development, the protocol is currently in use by Ethereum's consensus layer with 100,000s of participants. Nodes maintain a DHT routing table of Ethereum Node Records (ENRs), periodically flushing nodes that are no longer responsive and searching for new nodes by requesting records from their neighbours.

Neighbours in this sense are not necessarily in geographical proximity. Node distance is defined as the bitwise XOR of the nodes 32 bit IDs.

```
distance(Id1, Id2) = Id1 XOR Id2
```

In some situations these distances are placed into buckets by taking the logarithmic distance.

```
log_distance(Id1, Id2) = log2(distance(Id1, Id2))
```

In principle, an ENR is simply an arbitrary set of key/value pairs accompanied by a sequence number and signed by the author node's private key. In order to be included in and propagated around the DHT, the ENR must contain the node's dialable IP address and port.

Transport

The underlying transport for DiscV5 communication is UDP. Whilst UDP is not reliable and connectionless, it has much lower overhead than TCP or other similar protocols making it ideal for speculative communication with nodes over the discovery domain.

It does mean that UDP communication is a requirement for nodes wishing to participate.

Bootstrapping

When a node wishes to join the network for the first time. It needs to locate at least 1 initial peer in order to 'discover' other nodes. This role is performed by known public 'bootnodes'. Bootnodes may not be full network participants, they may simply be entrypoints containing well populated routing tables for nodes to query.

Topics

Topics are part of the DiscV5 specification, though the spec is as yet unfinished and implementations do not yet exist. The intention of topics is for the Ethereum P2P network to efficiently support any number of applications under the same discovery scheme. To date, many other applications use Ethereum's discovery network but the only way to 'discover' other nodes for the same application is to query nodes at random and interrogate them. Topics will allow this to be done more efficiently with nodes being able to 'advertise' themselves as supporting specific applications across the network.

DiscV5 on Aztec

DANGER

The intention here is to use Ethereum's DiscV5 discovery network. This has not been prototyped and is as yet untested. The alternative would be for Aztec nodes to form their own DiscV5 network, which would still work but wouldn't inherit the security properties of Ethereum's. We need to do more work to understand this.

Using Ethereum's DiscV5 network will have significant benefits for Aztec. Network security and resistance to censorship, sybil and eclipse attacks grows as the network gets larger. In the early days of the network, node discovery may be slow as the

number of Aztec nodes will be small as a proportion of the network. This can be better understood with the deployment of testnets. Over time, as the network grows and we hopefully see the introduction of topics this node discovery process will improve.

Aztec ENRs

The node record for an Aztec node will contain the following key/value pairs. The network endpoints don't all need to be specified but nodes will require at least one ip address and port. The public key is required to verify the signature included with the node record. The id is the identity scheme with "v4" being that currently used by Ethereum.

| key | value |
|-----------|-------------------------|
| id | "v4" |
| secp256k1 | The node's public key |
| ip | ipv4 address |
| tcp | tcp port |
| ip6 | ipv6 address |
| tcp6 | tcp port for v6 address |
| aztec | The aztec chain id |
| eth | The ethereum chain id |

Transaction Gossip

Transports

LibP2P clients specify 1 or more types of transport for communicating with other nodes. Clients must specify at least the TCP transport for use within the Aztec Network. Clients may optionally specify other, more sophisticated transports but it is not guaranteed that other nodes will support them.

Clients must accept connections on either IPV4, IPV6 or both. They must be able to dial both IPv4 and IPV6 addresses.

Clients behind a NAT must be publically dialable and they must provide their publically dialable endpoint in their ENR. They must have their infrastructure configured to route traffic received at the dialable endpoint to the local listener.

Multiplexing

LibP2P supports the multiplexing of stream based transports such as TCP. There are 2 widely implemented multiplexing modules within LibP2P, [mplex](#) and the more sophisticated [yamux](#). Clients must be configured to support mplex and may choose to support yamux.

Encryption handshake

Communication between nodes within LibP2P is encrypted. This is important to protect individual nodes and the network at large from malicious actors. Establishing keys requires a secure handshake protocol. Client's must specify LibP2P's [noise](#) protocol for this purpose.

GossipSub

LibP2P's [GossipSub](#) is a protocol that provides efficient propagation of transient

messages to all participants of the gossip domain. Peers congregate around topics that they subscribe to and publish on the network. Each topic's network is further divided into 2 layers of peering.

1. Full Message Peers - A sparsely connected network gossiping the complete contents of every message
2. Metadata Only Peers - A densely connected network gossiping only message metadata

Peerings are bidirectional, meaning that for any two connected peers, both peers consider their connection to be full-message or both peers consider their connection to be metadata-only.

Either peer can change the type of connection in either direction at any time. Peers periodically evaluate their peerings and attempt to balance the number of each type of peering to a configured range. The peering degree being the configured optimal number of full messages peers for each node. Generally speaking, a higher peering degree will result in faster message propagation to all peers at the expense of increased message duplication.

These layers ensure that all messages are efficiently propagated throughout the network whilst significantly reducing redundant traffic and lowering bandwidth requirements.

Peer Scoring

To maintain the health of the network, peers are scored based on their behaviour. Peers found to misbehave are penalised on a sliding scale from a reluctance to convert them to full message peers to refusing to gossip with them altogether.

Message Validation

The gossipsub protocol requests message validation from the application. Messages deemed invalid are discarded and not propagated further. The application can

specify whether the validation failure warrants the source peer being penalised for transmitting it.

Message Cache

Clients maintain a cache of recently seen messages from which other peers can request messages that they haven't received. Typically this would be used by metadata only peers who haven't received the complete message to do so. Messages are held in the cache for a configurable length of time, though this is usually just a few seconds.

GossipSub on Aztec

Aztec will use LibP2P's GossipSub protocol for transaction propagation. Nodes must support this protocol along with the v1.1 extensions and publish/subscribe to the topic `/aztec/{aztec-chainid}/tx/{message-version}`. The inclusion of `{message-version}` within the topic allows for the message specification to change and clients of the network will have to migrate to the new topic. We will aim to strike a balance between message propagation speed and lowering overall bandwidth requirements. Aztec block times will typically be 5-10 minutes so provided the network operates quickly enough for a user's transaction to be considered for inclusion in the 'next' block, the network can be optimised to reduce redundant gossiping.

The table below contains some of the relevant configuration properties and their default values. These parameters can be validated on testnet but it is expected that for the Aztec network, clients would use similar values, perhaps reducing peering degree slightly to favour reduced bandwidth over message propagation latency.

| Parameter | Description | Value |
|-----------|----------------------------|-------|
| D | The desired peering degree | 6 |

| Parameter | Description | Value |
|--------------------|--|-------------|
| D_low | The peering degree low watermark | 4 |
| D_high | The peering degree high watermark | 12 |
| heartbeat_interval | The time between heartbeats* | 1
second |
| mcache_len | The number of history windows before messages are ejected from cache | 5 |
| mcache_gossip | The number of history windows for messages to be gossiped | 3 |

(*)Several things happen at the heartbeat interval:

1. The nature of peerings are evaluated and changed if necessary
2. Message IDs are gossiped to a randomly selected set of metadata only peers
3. The message cache is advanced by a history window

Aztec Message Validation

Because Aztec transactions are significant in size, it is important to ensure that invalid messages are not propagated.

All of the [transaction validity conditions](#) must be verified at the point a message is received and reported to the protocol.

Peers sending messages that breach any of the validity conditions should be penalised for doing so using the peer scoring system within the protocol. For nullifier validations, a grace period should be applied such that transactions containing

nullifiers within very recently published blocks do not warrant a penalty being applied. It is important however that clients don't join the gossip protocol until they are fully synched with the chain, otherwise they risk being punished for unknowingly publishing invalid transactions.

Aztec Message Encoding

The [transaction object](#) contains a considerable amount of data, much of it in the format of variable length vectors of fixed 32 byte wide fields. We will encode this object using [SSZ](#), the encoding used by Ethereum's consensus layer. This format requires a pre-defined schema but encodes the payload such that it is very efficient to deserialise reducing the burden on clients to validate messages at the point of receipt.

Messages may then be compressed using [Snappy](#). Much of the payload may be uncompressable due to the random nature of it. We will validate this during testing. Whilst Snappy's compression is not as good as other algorithms such as zip, it offers an order of magnitude greater performance.

Synchronising With The Transaction Pool

GossipSub does not include a mechanism for synchronising the global set of messages at a given time. It is designed as a system to gossip transient data and messages are removed from caches after only a few seconds. We won't provide an additional protocol to perform an immediate synchronisation of the transaction pool via the P2P network. Whilst this might be desirable, we have the following rationale for not facilitating this.

1. Aztec transactions are large, approximately 40Kb. Downloading the entire pool would require transferring in the order of 100s of MB of data. At best this is undesirable and at worst it represents a DoS vector.
2. It is largely redundant. At the point at which a node joins the network, it is likely that production of a block is already underway and many of the transactions that would be downloaded will be removed as soon as that block is published.

3. Clients will naturally synchronise the transaction pool by joining the gossiping network and waiting for 1 or 2 blocks. New transactions will be received into the client's local pool and old transactions unknown to the client will be removed as blocks are published.

High Level Topology

Overview

A transaction begins with a call to a private function, which may invoke nested calls to other private and public functions. The entire set of private function calls is executed in a secure environment, and their proofs are validated and aggregated by private kernel circuits. Meanwhile, any public function calls triggered from private functions will be enqueued. The proofs for these calls, along with those from the nested public function calls, are generated and processed through public kernel circuits in any entity possessing the correct contexts.

Once all functions in a transaction are executed, the accumulated data is outputted from a tail circuit. These values are then inserted or updated to the [state trees](#) within the base rollup circuit. The merge rollup circuit facilitates the merging of two rollup proofs. Repeating this merging process enables the inclusion of more transactions in a block. Finally, the root rollup circuit produces the final proof, which is subsequently submitted and validated onchain.

To illustrate, consider a transaction involving the following functions, where circles depict private functions, and squares denote public functions:

 **INFO**

A note for Aztec protocol developers: In this protocol spec, the order in which the kernel circuit processes calls is different from previous literature, and is different from the current implementation (as at January 2024).

This transaction contains 6 private functions (f0 to f5) and 5 public functions (F0 to F4), with  being the entrypoint. The entire transaction is processed as follows:

A few things to note:

- A transaction always starts with an [initial private kernel circuit](#).
- An [inner private kernel circuit](#) won't be required if there is only one private function in a transaction.
- A [reset private kernel circuit](#) can be executed between two private kernel circuits to "reset" transient data. The reset process can be repeated as needed.
- Public functions are "enqueued" when invoked from a private function. Public kernel circuits will be executed after the completion of all private kernel iterations.
- A [base rollup circuit](#) can accept either a [tail public kernel circuit](#), or a [tail private kernel circuit](#) in cases where no public functions are present in the transaction.
- A [merge rollup circuit](#) can merge two base rollup circuits or two merge rollup circuits.
- The final step is the execution of the [root rollup circuit](#), which combines two base rollup circuits or two merge rollup circuits.

Private Function Circuit

Requirements

Private function circuits represent smart contract functions that can: privately read and modify leaves of the note hash tree and nullifier tree; perform computations on private data; and can be executed without revealing which function or contract has been executed.

The logic of each private function circuit is tailored to the needs of a particular application or scenario, but the public inputs of every private function circuit *must* adhere to a specific format. This specific format (often referred to as the "public inputs ABI for private functions") ensures that the [private kernel circuits](#) can correctly interpret the actions of every private function circuit.

Private Inputs

The private inputs of a private function circuit are customizable.

Public Inputs

The public inputs of *every* private function *must* adhere to the following ABI:

| Field | Type | Description |
|----------------------------|---|---|
| <code>call_context</code> | <code>CallContext</code> | Context of the call corresponding to this function execution. |
| <code>args_hash</code> | <code>field</code> | Hash of the function arguments. |
| <code>return_values</code> | <code>[field; RETURN_VALUES_LENGTH]</code> | Return values of this function call. |
| <code>note_hashes</code> | <code>[NoteHash; MAX_NOTE_HASHES_PER_CALL]</code> | New note hashes created in this |

| Field | Type | Description |
|--------------------------------|---|---|
| | | function call. |
| nullifiers | [Nullifier ; MAX_NULLIFIERS_PER_CALL] | New nullifiers created in this function call. |
| l2_to_l1_messages | [L2toL1Message ; MAX_L2_TO_L1_MSGS_PER_CALL] | New L2 to L1 messages created in this function call. |
| unencrypted_log_hashes | [UnencryptedLogHash ; MAX_UNENCRYPTED_LOG_HASHES_PER_CALL] | Hashes of the unencrypted logs emitted in this function call. |
| encrypted_log_hashes | [EncryptedLogHash ; MAX_ENCRYPTED_LOG_HASHES_PER_CALL] | Hashes of the encrypted logs emitted in this function call. |
| encrypted_note_preimage_hashes | [EncryptedNotePreimageHash ; MAX_ENCRYPTED_NOTE_PREIMAGE_HASHES_PER_CALL] | Hashes of the encrypted note preimages emitted in this function call. |
| note_hash_read_requests | [ReadRequest ; MAX_NOTE_HASH_READ_REQUESTS_PER_CALL] | Requests to prove the note hashes being read exist. |
| nullifier_read_requests | [ReadRequest ; MAX_NULLIFIER_READ_REQUESTS_PER_CALL] | Requests to prove the nullifiers being read exist. |
| key_validation_requests | [ParentSecretKeyValidationRequest ; MAX_KEY_VALIDATION_REQUESTS_PER_CALL] | Requests to validate keys |

| Field | Type | Description |
|------------------------------------|---|--|
| | | used in this function call. |
| public_call_requests | [PublicCallRequest ; MAX_PUBLIC_CALL_STACK_LENGTH_PER_CALL] | Requests to call public functions. |
| private_call_requests | [PrivateCallRequest ; MAX_PRIVATE_CALL_STACK_LENGTH_PER_CALL] | Requests to call Private functions. |
| counter_start | u32 | Counter at which the function call was initiated. |
| counter_end | u32 | Counter at which the function call ended. |
| min_reversible_side_effect_counter | u32 | Counter below which the side effects are non-reversible. |
| block_header | BlockHeader | Information about the trees used for the transaction. |
| chain_id | field | Chain ID of the transaction. |
| version | field | Version of the transaction. |

After generating a proof for a private function circuit, that proof (and associated public inputs) will be passed into a private kernel circuit as private inputs. Private kernel circuits use the private function's proof, public inputs, and

verification key, to verify the correct execution of the private function. Private kernel circuits then perform a number of checks and computations on the private function's public inputs.

Types

CallContext

| Field | Type | Description |
|-------------------------|--------------|---|
| msg_sender | AztecAddress | Address of the caller contract. |
| contract_address | AztecAddress | Address of the contract against which all state changes will be stored. |
| portal_contract_address | AztecAddress | Address of the portal contract to the storage contract. |
| is_static_call | bool | A flag indicating whether the call is a static call . |

GasSettings

| Field | Type | Description |
|-----------------------|-------|--|
| da.gas_limit | u32 | Total limit for DA gas for the transaction. |
| da.teardown_gas_limit | u32 | Limit for DA gas specific to the teardown phase. |
| da.max_fee_per_gas | field | Maximum amount that the sender is willing to pay per unit of DA gas. |
| l2.gas_limit | u32 | Total limit for L2 gas for the transaction. |
| l2.teardown_gas_limit | u32 | Limit for L2 gas specific to the teardown phase. |
| l2.max_fee_per_gas | field | Maximum amount that the sender is willing to pay per unit of L2 gas. |
| inclusion_fee | field | Flat fee the user pays for inclusion. |

NoteHash

| Field | Type | Description |
|---------|-------|---|
| value | field | Hash of the note. |
| counter | u32 | Counter at which the note hash was created. |

Nullifier

| Field | Type | Description |
|-------------------|-------|--|
| value | field | Value of the nullifier. |
| counter | u32 | Counter at which the nullifier was created. |
| note_hash_counter | u32 | Counter of the transient note the nullifier is created for. 0 if the nullifier does not associate with a transient note. |

L2toL1Message

| Field | Type | Description |
|---------|-------|---|
| value | field | L2-to-L1 message. |
| counter | u32 | Counter at which the message was emitted. |

UnencryptedLogHash

| Field | Type | Description |
|---------|-------|--|
| hash | field | Hash of the unencrypted log. |
| length | field | Number of fields of the log preimage. |
| counter | u32 | Counter at which the hash was emitted. |

EncryptedLogHash

| Field | Type | Description |
|------------|-------|--|
| hash | field | Hash of the encrypted log. |
| length | field | Number of fields of the log preimage. |
| counter | u32 | Counter at which the hash was emitted. |
| randomness | field | A random value to hide the contract address. |

EncryptedNotePreimageHash

| Field | Type | Description |
|-------------------|-------|---|
| hash | field | Hash of the encrypted note preimage. |
| length | field | Number of fields of the note preimage. |
| counter | u32 | Counter at which the hash was emitted. |
| note_hash_counter | u32 | Counter of the corresponding note hash. |

ReadRequest

| Field | Type | Description |
|------------------|--------------|--|
| value | field | Value being read. |
| contract_address | AztecAddress | Address of the contract the value was created. |
| counter | u32 | Counter at which the request was made. |

ParentSecretKeyValidationRequest

| Field | Type | Description |
|---------------------------|---------------|--|
| parent_public_key | GrumpkinPoint | Claimed parent public key of the secret key. |
| hardened_child_secret_key | fq | Secret key passed to the function. |

PublicCallRequest

| Field | Type | Description |
|----------------------|-------|--|
| call_stack_item_hash | field | Hash of the call stack item. |
| counter | u32 | Counter at which the request was made. |

PrivateCallRequest

| Field | Type | Description |
|----------------------|-------|--|
| call_stack_item_hash | field | Hash of the call stack item. |
| counter_start | u32 | Counter at which the call was initiated. |
| counter_end | u32 | Counter at which the call ended. |

BlockHeader

| Field | Type | Description |
|-----------------------------|-------|-------------------------------------|
| note_hash_tree_root | field | Root of the note hash tree. |
| nullifier_tree_root | field | Root of the nullifier tree. |
| l1_to_l2_messages_tree_root | field | Root of the l1-to-l2 messages tree. |
| public_data_tree_root | field | Root of the public data tree. |

| Field | Type | Description |
|------------------------------------|--------------------|---|
| <code>archive_tree_root</code> | <code>field</code> | Root of the state roots tree archived at the block prior to when the transaction was assembled. |
| <code>global_variables_hash</code> | <code>field</code> | Hash of the previous global variables. |

Private Kernel Circuit - Initial

Requirements

In the initial kernel iteration, the process involves taking a `transaction_request` and private call data , performing checks on this data (see below), and preparing the necessary data for subsequent circuits to operate. This "initial" circuit is an optimization over the `inner private kernel circuit`, as there is no "previous kernel" to verify at the beginning of a transaction. Additionally, this circuit executes tasks that need only occur once per transaction.

Key Checks within this Circuit

This first function call of the transaction must match the caller's intent

The following data in the `private_inputs.private_call` must match the corresponding fields of the user's `private_inputs.transaction_request`:

- `contract_address`
- `function_data`
- `args_hash`: Hash of the function arguments.

Notice: a `transaction_request` doesn't explicitly contain a signature. Aztec implements `account abstraction`, so the process for authorizing a transaction (if at all) is dictated by the logic of the functions of that transaction. In particular, an account contract can be called as an 'entrypoint' to a transaction, and there, custom authorization logic can be executed.

It must be a standard `synchronous function call`

For the `private_inputs.private_call.call_stack_item.public_inputs.call_context: CallContext`, the circuit checks that:

- It must not be a delegate call:
 - `call_context.is_delegate_call == false`
- It must not be a static call:
 - `call_context.is_static_call == false`

The `transaction_request` must be unique

It must emit the hash of the `private_inputs.transaction_request` as the `first nullifier`.

The hash is computed as:

```
let { origin, function_data, args_hash, tx_context } =
  private_inputs.transaction_request;
let tx_hash = hash(origin, function_data.hash(), args_hash, tx_context.hash());
```

Where `function_data.hash()` and `tx_context.hash()` are the hashes of the serialized field elements.

This nullifier serves multiple purposes:

- Identifying a transaction.
- Non-malleability. Preventing the signature of a transaction request from being reused in another transaction.
- Generating values that should be maintained within the transaction's scope. For example, it is utilized to [compute the note nonces](#) for all the note hashes in a transaction.

Note that the final transaction data is not deterministic for a given transaction request. The production of new notes, the destruction of notes, and various other values are likely to change based on the time and conditions when a transaction is being composed. However, the intricacies of implementation should not be a concern for the entity initiating the transaction.

Processing a Private Function Call

The function being called must exist within the contract class of the called `contract_address`

With the following data provided from `private_inputs.private_call`:

- `contract_address` in `private_call.call_stack_item`.
- `contract_instance`
- `contract_class`
- `function_data` in `private_call.call_stack_item`.

This circuit validates the existence of the function in the contract through the following checks:

1. Verify that the `contract_address` can be derived from the `contract_instance`:

Refer to the details [here](#) for the process of computing the address for a contract instance.

2. Verify that the `contract_instance.contract_class_id` can be derived from the given `contract_class`:

Refer to the details [here](#) for the process of computing the `contract_class_id`.

3. Verify that `contract_class_data.private_functions` includes the function being called:

- i. Compute the hash of the verification key:

■ `vk_hash = hash(private_call.vk)`

- ii. Compute the function leaf:

■ `hash(function_data.selector, vk_hash, private_call.bytecode_hash)`

- iii. Perform a membership check; that the function leaf exists within the function tree, where:

■ The index and sibling path are provided through `private_call.function_leaf_membership_witness`.
■ The root is `contract_class.private_functions`.

The private function proof must verify

It verifies that the private function was executed successfully with the provided proof data, verification key, and the public inputs of the [private function circuit](#). I.e. `private_inputs.private_call.vk`, `private_inputs.private_call.proof`, and `private_inputs.private_call.call_stack_item.public_inputs`.

Validate the counters.

In ensuring the integrity of emitted data, counters play a crucial role in establishing the chronological order of elements generated during a transaction. This validation process not only guards against misinterpretations but also reinforces trust in the sequence of transactions.

Counters are employed to validate the following aspects:

1. **Read Requests:** Verify that a read request is reading a value created before the request is submitted.
 - Refer to [Read Request Reset Private Kernel Circuit](#) for verification details.
2. **Ordered Emission:** Ensure that side effects (note hashes, nullifiers, logs) are emitted in the same order as they were created.
 - Refer to [Tail Private Kernel Circuit](#) for order enforcement.

For these operations to be effective, specific requirements for the counters must be met:

- Each counter for values of the same type must be unique, avoiding confusion about the order.
- Values emitted within a function call must not be mistaken for values emitted from another function call.

The circuit undergoes the following validations for data within `private_inputs.private_call.public_inputs`:

1. Validate the counter range of `call_stack_item`.
 - The `counter_start` must be `0`.
 - This check can be skipped for [inner private kernel circuit](#).
 - The `counter_end` must be strictly greater than the `counter_start`.

The counter range (`counter_start` to `counter_end`) is later used to restrict counters emitted within the call.

2. Validate the counter ranges of non-empty requests in `private_call_requests`.
 - The `counter_end` of each request must be strictly greater than its `counter_start`.
 - The `counter_start` of the first request must be strictly greater than the `counter_start` of the `call_stack_item`.
 - The `counter_start` of the second and each of the subsequent requests must be strictly greater than the `counter_end` of the previous request.
 - The `counter_end` of the last request must be strictly less than the `counter_end` of the `call_stack_item`.

When a `request` is [popped](#) in a nested iteration, its counter range is checked against the `call_stack_item`, as described [here](#). By enforcing that the counter ranges of all nested `private_call_requests` do not overlap with one

another in this step, a function circuit will not be able to emit a value whose counter falls in the range of another call.

3. Validate the counters of the non-empty elements in the following arrays:

- o note_hashes
- o nullifiers
- o 12_to_11_messages
- o unencrypted_log_hashes
- o encrypted_log_hashes
- o encrypted_note_preimage_hashes
- o note_hash_read_requests
- o nullifier_read_requests
- o public_call_requests

i. For each of the above "ordered" array, the counters of the non-empty elements must be in a strictly-increasing order:

- The counter of the first element must be strictly greater than the counter_start of the call_stack_item.
- The counter of each subsequent element must be strictly greater than the counter of the previous item.
- The counter of the last element must be strictly less than the counter_end of the call_stack_item.

ii. Additionally, the counters must not fall within the counter range of any nested private call.

Get the value NE, which is the number of non-empty requests in private_call_requests. If NE is greater than 0, the circuit checks the following:

For each note_hash at index i in note_hashes:

- Find the request_index at hints.note_hash_range_hints[i], which is the index of the private_call_requests with the smallest counter_start that was emitted after the note_hash.
- If request_index equals NE, indicating no request was emitted after the note_hash, its counter must be greater than the counter_end of the last request.
- If request_index equals 0, indicating no request was emitted before the note_hash. Its counter must be less than the counter_start of the first request.
- Otherwise, the request was emitted after the note_hash, and its immediate previous request was emitted before the note_hash. Its counter must fall between those two requests.

The code simplifies as:

```
let NE = count_non_empty_elements(private_call_requests);
for i in 0..note_hashes.len() {
    let note_hash = note_hashes[i];
    if !note_hash.is_empty() {
        let request_index = note_hash_range_hints[i];
        if request_index != NE {
            note_hash.counter < private_call_requests[request_index].counter_start;
        }
        if request_index != 0 {
            note_hash.counter > private_call_requests[request_index - 1].counter_end;
        }
    }
}
```

Repeat the above process for emitted data, including:

- `nullifiers`
- `unencrypted_log_hashes`
- `encrypted_log_hashes`
- `encrypted_note_preimage_hashes`

Validating Public Inputs

Verifying the `TransientAccumulatedData`.

The various side effects of the `private_inputs.private_call.call_stack_item.public_inputs: PrivateFunctionPublicInputs` are formatted and pushed to the various arrays of the `public_inputs.transient_accumulated_data: TransientAccumulatedData`.

This circuit verifies that the values in `private_inputs.private_call.call_stack_item.public_inputs: PrivateFunctionPublicInputs` are aggregated into the various arrays of the `public_inputs.transient_accumulated_data: TransientAccumulatedData` correctly.

1. Ensure that the specified values in the following arrays match those in the corresponding arrays in the `private_function_public_inputs`:

- `note_hash_contexts`
 - `value, counter`
- `nullifier_contexts`
 - `value, counter`
- `l2_to_l1_message_contexts`
 - `value, counter`
- `note_hash_read_requests`

- `value`, `contract_address`, `counter`
- `nullifier_read_requests`
 - `value`, `contract_address`, `counter`
- `key_validation_request_contexts`
 - `parent_public_key`, `hardened_child_secret_key`
- `unencrypted_log_hash_contexts`
 - `hash`, `length`, `counter`
- `encrypted_log_hash_contexts`
 - `hash`, `length`, `randomness`, `counter`
- `encrypted_note_preimage_hash_contexts`
 - `hash`, `length`, `counter`, `note_hash_counter`
- `public_call_request_contexts`
 - `call_stack_item_hash`, `counter`

1. Check that the values in `private_call_request_stack` align with the values in `private_call_requests` within `private_function_public_inputs`, but in **reverse** order.

It's important that the `private_call_requests` are "pushed" to the `private_call_request_stack` in reverse order to ensure that they are executed in chronological order.

2. For each non-empty call request in both `private_call_request_stack` and `public_call_request_contexts` within `public_inputs.transient_accumulated_data`:

- The `caller_contract_address` equals the `private_call.call_stack_item.contract_address`.
- The following values in `caller_context` are either empty or align with the values in the `private_inputs.private_call.call_stack_item.public_inputs.call_context`:
 - `(caller_context.msg_sender == 0) & (caller_context.storage_contract_address == 0)`
 - Or `(caller_context.msg_sender == call_context.msg_sender) & (caller_context.storage_contract_address == call_context.storage_contract_address)`
- The `is_static_call` flag must be propagated:
 - `caller_context.is_static_call == call_context.is_static_call`

The caller context in a call request may be empty for standard calls. This precaution is crucial to prevent information leakage, particularly as revealing the `msg_sender` of this private function when calling a public function could pose security risks.

3. For each non-empty item in the following arrays, its `contract_address` must equal the `storage_contract_address` in `private_inputs.private_call.call_stack_item.public_inputs.call_context`:

- `note_hash_contexts`
- `nullifier_contexts`

- `l2_to_l1_message_contexts`
- `key_validation_request_contexts`
- `unencrypted_log_hash_contexts`
- `encrypted_log_hash_contexts`
- `encrypted_note_preimage_hash_contexts`

Ensuring the alignment of the contract addresses is crucial, as it is later used to [silo the values](#) and to establish associations with values within the same contract.

- For each non-empty item in `l2_to_l1_message_contexts`, its `portal_contract_address` must equal the `portal_contract_address` defined in `private_function_public_inputs.call_context`.
 - For each `note_hash_context: NoteHashContext` in the `note_hash_contexts`, validate its `nullifier_counter`.
The value of the `nullifier_counter` can be:
 - Zero: if the note is not nullified in the same transaction.
 - Strictly greater than `note_hash.counter`: if the note is nullified in the same transaction.
- Nullifier counters are used in the [reset private kernel circuit](#) to ensure a read happens **before** a transient note is nullified.
- Zero can be used to indicate a non-existing transient nullifier, as this value can never serve as the counter of a nullifier. It corresponds to the `counter_start` of the first function call.

Note that the verification process outlined above is also applicable to the inner private kernel circuit. However, given that the `transient_accumulated_data` for the inner private kernel circuit comprises both values from previous iterations and the `private_call`, the above process specifically targets the values stemming from the `private_call`. The inner kernel circuit performs an [extra check](#) to ensure that the `transient_accumulated_data` also contains values from the previous iterations.

Verifying the constant data.

It verifies that:

- The `tx_context` in the `constant_data` matches the `tx_context` in the `transaction_request`.
- The `block_header` must align with the one used in the private function circuit, as verified [earlier](#).

Verifying the `min_revertible_side_effect_counter`.

It verifies that the `min_revertible_side_effect_counter` equals the value in the `public_inputs` of the private function circuit.

Diagram

This diagram flows from the private inputs (which can be considered "inputs") to the public inputs (which can be considered "outputs").

Key:

The diagram:

PrivateInputs

| Field | Type | Description |
|---------------------|--------------------|-------------|
| transaction_request | TransactionRequest | |
| private_call | PrivateCall | |

TransactionRequest

Data that represents the caller's intent.

| Field | Type | Description |
|---------------|--------------------|---------------------------------------|
| origin | AztecAddress | Address of the entrypoint contract. |
| function_data | FunctionData | Data of the function being called. |
| args_hash | field | Hash of the function arguments. |
| tx_context | TransactionContext | Information about the transaction. |
| gas_settings | GasSettings | User-defined gas limits and max fees. |

PrivateCall

Data that holds details about the current private function call.

| Field | Type | Description |
|----------------------------------|----------------------|--|
| call_stack_item | PrivateCallStackItem | Information about the current private function call. |
| proof | Proof | Proof of the private function circuit. |
| vk | VerificationKey | Verification key of the private function circuit. |
| bytecode_hash | field | Hash of the function bytecode. |
| contract_instance | ContractInstance | Data of the contract instance being called. |
| contract_class | ContractClass | Data of the contract class. |
| function_leaf_membership_witness | MembershipWitness | Membership witness for the function being called. |

Hints

| Field | Type | Description |
|-----------------------------|--|---|
| note_hash_range_hints | [field, MAX_NOTE_HASHES_PER_CALL] | Indices of the next emitted private call requests for note hashes. |
| nullifier_range_hints | [field, MAX_NULLIFIERS_PER_CALL] | Indices of the next emitted private call requests for nullifiers. |
| unencrypted_log_range_hints | [field, MAX_UNENCRYPTED_LOG_HASHES_PER_CALL] | Indices of the next emitted private call requests for unencrypted logs. |
| encrypted_log_range_hints | [field, MAX_ENCRYPTED_LOG_HASHES_PER_CALL] | Indices of the next emitted private call requests for encrypted logs. |
| encrypted_note_range_hints | [field, | Indices of the next |

| Field | Type | Description |
|-------|---|--|
| | [MAX_ENCRYPTED_NOTE_PREIMAGE_HASHES_PER_CALL] | emitted private call requests for encrypted notes. |

PublicInputs

| Field | Type | Description |
|---|--|-------------|
| <code>constant_data</code> | ConstantData | |
| <code>transient_accumulated_data</code> | TransientAccumulatedData | |
| <code>min_revertible_side_effect_counter</code> | u32 | |

ConstantData

Data that remains the same throughout the entire transaction.

| Field | Type | Description |
|-------------------------|------------------------------------|--|
| <code>header</code> | Header | Header of a block which was used when assembling the tx. |
| <code>tx_context</code> | TransactionContext | Context of the transaction. |

TransientAccumulatedData

| Field | Type | Description |
|---------------------------------|--|--|
| <code>note_hash_contexts</code> | [NoteHashContext ; MAX_NOTE_HASHES_PER_TX] | Note hashes with extra data aiding verification. |
| <code>nullifier_contexts</code> | [NullifierContext ; MAX_NULLIFIERS_PER_TX] | Nullifiers with extra data aiding |

| Field | Type | Description |
|--|---|---|
| | | verification. |
| <code>l2_to_l1_message_contexts</code> | <code>[L2toL1MessageContext];
MAX_L2_TO_L1_MSGS_PER_TX]</code> | L2-to-I1 messages with extra data aiding verification. |
| <code>unencrypted_log_hash_contexts</code> | <code>[UnencryptedLogHashContext];
MAX_UNENCRYPTED_LOG_HASHES_PER_TX]</code> | Hashes of the unencrypted logs with extra data aiding verification. |
| <code>encrypted_log_hash_contexts</code> | <code>[EncryptedLogHashContext];
MAX_ENCRYPTED_LOG_HASHES_PER_TX]</code> | Hashes of the encrypted logs with extra data aiding verification. |
| <code>encrypted_note_preimage_hash_contexts</code> | <code>[EncryptedNotePreimageHashContext];
MAX_ENCRYPTED_NOTE_PREIMAGE_HASHES_PER_TX]</code> | Hashes of the encrypted note preimages with extra data aiding verification. |
| <code>note_hash_read_requests</code> | <code>[ReadRequest];
MAX_NOTE_HASH_READ_REQUESTS_PER_TX]</code> | Requests to prove the note hashes being read exist. |
| <code>nullifier_read_requests</code> | <code>[ReadRequest];
MAX_NULLIFIER_READ_REQUESTS_PER_TX]</code> | Requests to prove the nullifiers being read exist. |

| Field | Type | Description |
|--|--|---|
| <code>key_validation_request_contexts</code> | [<code>ParentSecretKeyValidationRequestContext</code> ; <code>MAX_KEY_VALIDATION_REQUESTS_PER_TX</code>] | Requests to validate nullifier keys. |
| <code>public_call_request_contexts</code> | [<code>PublicCallRequestContext</code> ; <code>MAX_PUBLIC_CALL_STACK_LENGTH_PER_TX</code>] | Requests to call publics functions. |
| <code>private_call_request_stack</code> | [<code>PrivateCallRequestContext</code> ; <code>MAX_PRIVATE_CALL_STACK_LENGTH_PER_TX</code>] | Requests to call private functions.
Pushed to the stack in reverse order so that they will be executed in chronological order. |

Types

FunctionData

| Field | Type | Description |
|--------------------------------|--|--|
| <code>function_selector</code> | <code>u32</code> | Selector of the function being called. |
| <code>function_type</code> | <code>private</code> <code>public</code> | Type of the function being called. |

ContractClass

| Field | Type | Description |
|---------------------------------|---------------------------|--|
| <code>version</code> | <code>u8</code> | Version identifier. |
| <code>registerer_address</code> | <code>AztecAddress</code> | Address of the canonical contract used for registering this class. |

| Field | Type | Description |
|-------------------------|-------|---|
| artifact_hash | field | Hash of the contract artifact. |
| private_functions | field | Merkle root of the private function tree. |
| public_functions | field | Merkle root of the public function tree. |
| unconstrained_functions | field | Merkle root of the unconstrained function tree. |

TransactionContext

| Field | Type | Description |
|----------|------------------------------------|------------------------------|
| tx_type | standard fee_paying fee_rebate | Type of the transaction. |
| chain_id | field | Chain ID of the transaction. |
| version | field | Version of the transaction. |

PrivateCallStackItem

| Field | Type | Description |
|------------------|-----------------------------|---|
| contract_address | AztecAddress | Address of the contract on which the function is invoked. |
| function_data | FunctionData | Data of the function being called. |
| public_inputs | PrivateFunctionPublicInputs | Public inputs of the private function circuit. |

PrivateCallRequestContext

| Field | Type | Description |
|----------------------|-------|--|
| call_stack_item_hash | field | Hash of the call stack item. |
| counter_start | u32 | Counter at which the call was initiated. |

| Field | Type | Description |
|-------------------------|---------------|---|
| counter_end | u32 | Counter at which the call ended. |
| caller_contract_address | AztecAddress | Address of the contract calling the function. |
| caller_context | CallerContext | Context of the contract calling the function. |

CallerContext

| Field | Type | Description |
|--------------------------|--------------|--|
| msg_sender | AztecAddress | Address of the caller contract. |
| storage_contract_address | AztecAddress | Storage contract address of the caller contract. |
| is_static_call | bool | A flag indicating whether the call is a static call. |

NoteHashContext

| Field | Type | Description |
|-------------------|--------------|--|
| value | field | Hash of the note. |
| counter | u32 | Counter at which the note hash was created. |
| nullifier_counter | field | Counter at which the nullifier for the note was created. |
| contract_address | AztecAddress | Address of the contract the note was created. |

NullifierContext

| Field | Type | Description |
|---------|-------|---|
| value | field | Value of the nullifier. |
| counter | u32 | Counter at which the nullifier was created. |

| Field | Type | Description |
|-------------------|--------------|--|
| note_hash_counter | u32 | Counter of the transient note the nullifier is created for. 0 if the nullifier does not associate with a transient note. |
| contract_address | AztecAddress | Address of the contract the nullifier was created. |

L2toL1MessageContext

| Field | Type | Description |
|-------------------------|--------------|--|
| value | field | L2-to-I2 message. |
| counter | u32 | Counter at which the message was emitted. |
| portal_contract_address | AztecAddress | Address of the portal contract to the contract. |
| contract_address | AztecAddress | Address of the contract the message was created. |

ParentSecretKeyValidationRequestContext

| Field | Type | Description |
|---------------------------|---------------|---|
| parent_public_key | GrumpkinPoint | Claimed parent public key of the secret key. |
| hardened_child_secret_key | fq | Secret key passed to the contract. |
| contract_address | AztecAddress | Address of the contract the request was made. |

UnencryptedLogHashContext

| Field | Type | Description |
|---------|-------|--|
| hash | field | Hash of the unencrypted log. |
| length | field | Number of fields of the log preimage. |
| counter | u32 | Counter at which the hash was emitted. |

| Field | Type | Description |
|------------------|--------------|--|
| contract_address | AztecAddress | Address of the contract the log was emitted. |

EncryptedLogHashContext

| Field | Type | Description |
|------------------|--------------|--|
| hash | field | Hash of the encrypted log. |
| length | field | Number of fields of the log preimage. |
| counter | u32 | Counter at which the hash was emitted. |
| contract_address | AztecAddress | Address of the contract the log was emitted. |
| randomness | field | A random value to hide the contract address. |

EncryptedNotePreimageHashContext

| Field | Type | Description |
|-------------------|--------------|--|
| hash | field | Hash of the encrypted note preimage. |
| length | field | Number of fields of the note preimage. |
| counter | u32 | Counter at which the hash was emitted. |
| contract_address | AztecAddress | Address of the contract the log was emitted. |
| note_hash_counter | field | Counter of the corresponding note hash. |

MembershipWitness

| Field | Type | Description |
|--------------|------------|--|
| leaf_index | field | Index of the leaf in the tree. |
| sibling_path | [field; H] | Sibling path to the leaf in the tree. H represents the height of the tree. |

Private Kernel Circuit - Inner

Requirements

Each inner kernel iteration processes a private function call and the results of a previous kernel iteration.

Verification of the Previous Iteration

Verifying the previous kernel proof.

It verifies that the previous iteration was executed successfully with the provided proof data, verification key, and public inputs, sourced from `private_inputs`, `previous_kernel`.

The preceding proof can be:

- Initial private kernel proof.
- Inner private kernel proof.
- Reset private kernel proof.

The previous proof and the proof for the current function call are verified using recursion.

Processing Private Function Call

Ensuring the function being called exists in the contract.

This section follows the same `process` as outlined in the initial private kernel circuit.

Ensuring the current call matches the call request.

The top item in the `private_call_request_stack` of the `previous_kernel` must pertain to the current function call.

This circuit will:

1. Pop the call request from the stack:

```
◦ call_request =  
    previous_kernel.public_inputs.transient_accumulated_data.private_call_request_stack.pop()
```

2. Compare the hash with that of the current function call:

```
◦ call_request.call_stack_item_hash == private_call.call_stack_item.hash()  
◦ The hash of the call_stack_item is computed as:  
    ■ hash(contract_address, function_data.hash(), public_inputs.hash())
```

- Where `function_data.hash()` and `public_inputs.hash()` are the hashes of the serialized field elements.

Ensuring this function is called with the correct context.

For the `call_context` in the `public_inputs` of the `private_call`.`call_stack_item` and the `call_request` popped in the previous step, this circuit checks that:

1. If it is a standard call: `call_context.is_delegate_call == false`

- The `msg_sender` of the current iteration must be the same as the caller's `contract_address`:
 - `call_context.msg_sender == call_request.caller_contract_address`
- The `storage_contract_address` of the current iteration must be the same as its `contract_address`:
 - `call_context.storage_contract_address == call_stack_item.contract_address`

2. If it is a delegate call: `call_context.is_delegate_call == true`

- The `caller_context` in the `call_request` must not be empty. Specifically, the following values of the caller must not be zeros:
 - `msg_sender`
 - `storage_contract_address`
- The `msg_sender` of the current iteration must equal the caller's `msg_sender`:
 - `call_context.msg_sender == caller_context.msg_sender`
- The `storage_contract_address` of the current iteration must equal the caller's `storage_contract_address`:
 - `call_context.storage_contract_address == caller_context.storage_contract_address`
- The `storage_contract_address` of the current iteration must not equal the `contract_address`:
 - `call_context.storage_contract_address != call_stack_item.contract_address`

3. If it is NOT a static call: `call_context.is_static_call == false`

- The previous iteration must not be a static call:
 - `caller_context.is_static_call == false`

Verifying the private function proof.

It verifies that the private function was executed successfully with the provided proof data, verification key, and the public inputs, sourced from `private_inputs`.`private_call`.

This circuit verifies this proof and the proof of the previous kernel iteration using recursion, and generates a single proof. This consolidation of multiple proofs into one is what allows the private kernel circuits to gradually merge private function proofs into a single proof of execution that represents the entire private section of a transaction.

Verifying the public inputs of the private function circuit.

It ensures the private function circuit's intention by checking the following in `private_call.call_stack_item.public_inputs`:

- The `block_header` must match the one in the `constant_data`.
- If it is a static call (`public_inputs.call_context.is_static_call == true`), it ensures that the function does not induce any state changes by verifying that the following arrays are empty:
 - `note_hashes`
 - `nullifiers`
 - `l2_to_l1_messages`
 - `unencrypted_log_hashes`
 - `encrypted_log_hashes`
 - `encrypted_note_preimage_hashes`

Verifying the counters.

This section follows the same `process` as outlined in the initial private kernel circuit.

Additionally, it verifies that for the `call_stack_item`, the `counter_start` and `counter_end` must match those in the `call_request` popped from the `private_call_request_stack` in a previous step.

Validating Public Inputs

Verifying the transient accumulated data.

The `transient_accumulated_data` in this circuit's `public_inputs` includes values from both the previous iterations and the `private_call`.

For each array in the `transient_accumulated_data`, this circuit verifies that:

1. It is populated with the values from the previous iterations, specifically:

- `public_inputs.transient_accumulated_data.ARRAY[0..N] == private_inputs.previous_kernel.public_inputs.transient_accumulated_data.ARRAY[0..N]`

| It's important to note that the top item in the `private_call_request_stack` from the `previous_kernel` won't be included, as it has been removed in a `previous step`.

2. As for the subsequent items appended after the values from the previous iterations, they constitute the values from the `private_call`, and each must undergo the same `verification` as outlined in the initial private kernel circuit.

Verifying other data.

It verifies that the `constant_data` and the `min_revertible_side_effect_counter` in the `public_inputs` align with the corresponding values in `private_inputs`.`previous_kernel`.`public_inputs`.

Private Inputs

PreviousKernel

Data of the previous kernel iteration.

| Field | Type | Description |
|---------------------------------|---|--|
| <code>public_inputs</code> | <code>InitialPrivateKernelPublicInputs</code> | Public inputs of the proof. |
| <code>proof</code> | <code>Proof</code> | Proof of the kernel circuit. |
| <code>vk</code> | <code>VerificationKey</code> | Verification key of the kernel circuit. |
| <code>membership_witness</code> | <code>MembershipWitness</code> | Membership witness for the verification key. |

PrivateCall

The format aligns with the `PrivateCall` of the initial private kernel circuit.

PublicInputs

The format aligns with the `Public Inputs` of the initial private kernel circuit.

Private Kernel Circuit - Reset

Requirements

A reset circuit is designed to abstain from processing individual private function calls. Instead, it injects the outcomes of an initial, inner, or another reset private kernel circuit, scrutinizes the public inputs, and clears the verifiable data within its scope. A reset circuit can be executed either preceding the tail private kernel circuit, or as a means to "reset" public inputs at any point between two private kernels, allowing data to accumulate seamlessly in subsequent iterations.

There are 3 variations of reset circuits:

- Read Request Reset Private Kernel Circuit.
- Parent Secret Key Validation Request Reset Private Kernel Circuit.
- Transient Note Reset Private Kernel Circuit.

The incorporation of these circuits not only enhances the modularity and repeatability of the "reset" process but also diminishes the overall workload. Rather than conducting resource-intensive computations such as membership checks in each iteration, these tasks are only performed as necessary within the reset circuits.

Read Request Reset Private Kernel Circuit.

This reset circuit conducts verification on some or all accumulated read requests and subsequently removes them from the `transient_accumulated_data` within the `public_inputs` of the `previous_kernel`.

Depending on the value specified in `hints.reset_type`, it can target different read requests for resetting:

- For `reset_type == note_hash`: `target_read_requests = note_hash_read_requests`
- For `reset_type == nullifier`: `target_read_requests = nullifier_read_requests`

A read request can pertain to one of two types of values:

- A settled value: generated in a prior successful transaction and included in the tree.
- A pending value: created in the current transaction, not yet part of the tree.

1. To clear read requests for settled values, the circuit performs membership checks for the target read requests using the `hints` provided via `private_inputs`.

For each `persistent_read_index` at index `i` in `hints.persistent_read_indices`:

- i. If the `persistent_read_index` equals the length of the `target_read_requests` array, there is no read request to be verified. Skip the rest.
- ii. Locate the `read_request` using the index:
 - `read_request = target_read_requests[persistent_read_index]`
- iii. Perform a membership check on the value being read. Where:
 - The leaf corresponds to the value: `read_request.value`
 - The index and sibling path are in: `hints.read_request_membership_witnesses[i]`.
 - The root is sourced from the `block_header` within `public_inputs.constant_data`:
 - For note hash: `note_hash_tree_root`
 - For nullifier: `nullifier_tree_root`

Following the above process, at most `N` read requests will be cleared, where `N` is the length of the `persistent_read_indices` array. It's worth noting that there can be multiple versions of this reset circuit, each with a different value of `N`.

2. To clear read requests for pending values, the circuit ensures that the values were created before the corresponding read operations, utilizing

the hints provided via `private_inputs`.

For each `transient_read_index` at index `i` in `hints.transient_read_indices`:

- i. If the `transient_read_index` equals the length of the `target_read_requests` array, there is no read request to be verified. Skip the rest.
- ii. Locate the `read_request` using the index:
 - `read_request = target_read_requests[transient_read_index]`
- iii. Locate the `target` being read using the index `hints.pending_value_indices[i]`:
 - For note hash: `target = note_hash_contexts[index]`
 - For nullifier: `target = nullifier_contexts[index]`
- iv. Verify the following:
 - `read_request.value == target.value`
 - `read_request.contract_address == target.contract_address`
 - `read_request.counter > target.counter`
- v. When resetting a note hash, verify that the target note hash is not nullified before the read happens:
 - `(target.nullifier_counter > read_request.counter) | (target.nullifier_counter == 0)`

Given that a reset circuit can execute between two private kernel circuits, there's a possibility that the value being read is emitted in a nested execution and hasn't been included in the `public_inputs`. In such cases, the read request cannot be verified in the current reset circuit and must be processed in another reset circuit after the value has been aggregated to the `public_inputs`.

3. This circuit then ensures that the read requests that haven't been verified should remain in the `transient_accumulated_data` within its `public_inputs`.

For each `read_request` at index `i` in the `target_read_requests`, find its `status` at `hints.read_request_statuses[i]`. Verify the following:

- If `status.state == persistent`, `i == persistent_read_indices[status.index]`.
- If `status.state == transient`, `i == transient_read_indices[status.index]`.
- If `status.state == nada`, `read_request == public_inputs.transient_accumulated_data.target_read_requests[status.index]`.

Parent Secret Key Validation Request Reset Private Kernel Circuit.

This reset circuit validates the correct derivation of secret keys used in private functions, and subsequently removes them from the `transient_accumulated_data` within the `public_inputs` of the `previous_kernel`.

Initialize `requests_kept` to `0`.

For each `request` at index `i` in `key_validation_request_contexts`, locate the `master_secret_key` at `master_secret_keys[i]` and the relevant `app_secret_key` generator at `app_secret_keys_generators[i]`, provided as `hints` through `private_inputs`.

1. If `master_secret_key == 0`, ensure the request remain within the `public_inputs`:
 - `public_inputs.transient_accumulated_data.key_validation_request_contexts[requests_kept] == request`
 - Increase `requests_kept` by 1: `requests_kept += 1`
2. Else:
 - Verify that the public key is associated with the `master_secret_key`: `request.parent_public_key == master_secret_key * G`
 - Verify that the secret key was correctly derived for the contract: `request.hardened_child_secret_key == hash(master_secret_key, request.contract_address)`

Transient Note Reset Private Kernel Circuit.

In the event that a pending note is nullified within the same transaction, its note hash, nullifier, and all encrypted note preimage hashes can be removed from the public inputs. This not only avoids redundant data being broadcasted, but also frees up space for additional note hashes and nullifiers in the subsequent iterations.

1. Ensure that each note hash is either propagated to the `public_inputs` or nullified in the same transaction.

Initialize both `notes_kept` and `notes_removed` to 0.

For each `note_hash` at index `i` in `note_hash_contexts` within the `private_inputs`, find the index of its nullifier at `transient_nullifier_indices[i]`, provided as `hints`:

- If `transient_nullifier_indices[i] == nullifier_contexts.len()`:
 - Verify that the `note_hash` remains within the `transient_accumulated_data` in the `public_inputs`: `note_hash == public_inputs.transient_accumulated_data.note_hash_contexts[notes_kept]`
 - Increment `notes_kept` by 1: `notes_kept += 1`
- Else, locate the `nullifier` at `nullifier_contexts[transient_nullifier_indices[i]]`:
 - Verify that the nullifier is associated with the note:
 - `nullifier.contract_address == note_hash.contract_address`
 - `nullifier.note_hash_counter == note_hash.counter`
 - `nullifier.counter == note_hash.nullifier_counter`
 - Increment `notes_removed` by 1: `notes_removed += 1`
 - Ensure that an empty `note_hash` is appended to the end of `note_hash_contexts` in the `public_inputs`:
 - `public_inputs.transient_accumulated_data.note_hash_contexts[N - notes_removed].is_empty() == true`
 - Where `N` is the length of `note_hash_contexts`.

Note that the check `nullifier.counter > note_hash.counter` is not necessary as the `nullifier_counter` is assured to be greater than the counter of the note hash when `propagated` from either the initial or inner private kernel circuits.

2. Ensure that nullifiers not associated with note hashes removed in the previous step are retained within the `transient_accumulated_data` in the `public_inputs`.

Initialize both `nullifiers_kept` and `nullifiers_removed` to 0.

For each `nullifier` at index `i` in the `nullifier_contexts` within the `private_inputs`, find the index of its corresponding transient nullifier at `nullifier_index_hints[i]`, provided as `hints`:

- If `nullifier_index_hints[i] == transient_nullifier_indices.len()`:
 - Verify that the `nullifier` remains within the `transient_accumulated_data` in the `public_inputs`: `nullifier == public_inputs.transient_accumulated_data.nullifier_contexts=nullifiers_kept]`
 - Increment `nullifiers_kept` by 1: `nullifiers_kept += 1`
- Else, compute `transient_nullifier_index` as `transient_nullifier_indices=nullifier_index_hints[i]`:
 - Verify that: `transient_nullifier_index == i`
 - Increment `nullifiers_removed` by 1: `nullifiers_removed += 1`
 - Ensure that an empty `nullifier` is appended to the end of `nullifier_contexts` in the `public_inputs`:
 - `public_inputs.transient_accumulated_data.nullifier_contexts[N - nullifiers_removed].is_empty() == true`
 - Where `N` is the length of `nullifier_contexts`.

After these steps, ensure that all nullifiers associated with transient note hashes have been identified and removed:

```
nullifiers_removed == notes_removed
```

3. Ensure that `encrypted_note_preimage_hashes` not associated with note hashes removed in the previous step are retained within the `[transient_accumulated_data](./private-kernel-initial#transientaccumulateddata)` in the `public_inputs`.

Initialize both `hashes_kept` and `hashes_removed` to 0.

For each `preimage_hash` at index `i` in the `encrypted_note_preimage_hash_contexts` within the `private_inputs`, find the `index_hint` of its corresponding hash within `public_inputs` at `encrypted_note_preimage_hash_index_hints[i]`, provided as `hints`:

- If `index_hint == encrypted_note_preimage_hash_contexts.len()`:
 - Ensure that the associated note hash is removed:
 - Locate the `note_hash` at `private_inputs.transient_accumulated_data.note_hash_contexts[log_note_hash_hints[i]]`.
 - Verify that the `preimage_hash` is associated with the `note_hash`:
 - `preimage_hash.note_hash_counter == note_hash.counter`
 - `preimage_hash.contract_address == note_hash.contract_address`
 - Confirm that the `note_hash` has a corresponding nullifier and has been removed in the first step of this section:
 - `transient_nullifier_indices[log_note_hash_hints[i]] != nullifier_contexts.len()`
 - Increment `hashes_removed` by 1: `hashes_removed += 1`
 - Ensure that an empty item is appended to the end of `encrypted_note_preimage_hash_contexts` in the `public_inputs`:
 - `encrypted_note_preimage_hash_contexts[N - hashes_removed].is_empty() == true`
 - Where `N` is the length of `encrypted_note_preimage_hash_contexts`.
 - Else, find the `mapped_preimage_hash` at `encrypted_note_preimage_hash_contexts[index_hint]` within `public_inputs`:
 - Verify that the context is aggregated to the `public_inputs` correctly:
 - `index_hint == hashes_kept`
 - `mapped_preimage_hash == preimage_hash`
 - Ensure that the associated note hash is retained in the `public_inputs`:
 - Locate the `note_hash` at `public_inputs.transient_accumulated_data.note_hash_contexts[log_note_hash_hints[i]]`.
 - Verify that the `preimage_hash` is associated with the `note_hash`:
 - `preimage_hash.note_hash_counter == note_hash.counter`
 - `preimage_hash.contract_address == note_hash.contract_address`
 - Increment `hashes_kept` by 1: `hashes_kept += 1`

Note that this reset process may not necessarily be applied to all transient notes at a time. In cases where a note will be read in a yet-to-be-processed nested execution, the transient note hash and its nullifier must be retained in the `public_inputs`. The reset can only occur in a later reset circuit after all associated read requests have been verified and cleared.

Common Verifications

Below are the verifications applicable to all reset circuits:

Verifying the previous kernel proof.

It verifies that the previous iteration was executed successfully with the given proof data, verification key, and public inputs, sourced from `private_inputs.previous_kernel`.

The preceding proof can be:

- Initial private kernel proof.
- Inner private kernel proof.
- Reset private kernel proof.

Verifying the accumulated data.

It ensures that the `accumulated_data` in the `public_inputs` matches the `accumulated_data` in `private_inputs.previous_kernel.public_inputs`.

Verifying the transient accumulated data.

All arrays in the `transient_accumulated_data` in the `public_inputs` must equal their corresponding arrays in `private_inputs.previous_kernel.public_inputs.transient_accumulated_data`, with the exception of those modified by the reset circuits:

1. Read request reset circuit (for note hashes): `note_hash_read_requests`
2. Read request reset circuit (for nullifiers): `nullifier_read_requests`
3. Parent secret key validation request reset circuit (for nullifier keys): `key_validation_request_contexts`
4. Transient note reset circuit: `note_hash_contexts` and `nullifier_contexts`

Verifying other data.

This section follows the same `process` as outlined in the inner private kernel circuit.

PrivateInputs

PreviousKernel

The format aligns with the `PreviousKernel` of the inner private kernel circuit.

Hints for Read Request Reset Private Kernel Circuit

| Field | Type | Description |
|--|---|--|
| <code>reset_type</code> | <code>note_hash</code> <code>nullifier</code> | The type of read requests to be reset. |
| <code>transient_read_indices</code> | <code>[field; N]</code> | Indices of the read requests for transient values. |
| <code>pending_value_indices</code> | <code>[field; N]</code> | Indices of the values for transient reads. |
| <code>persistent_read_indices</code> | <code>[field; M]</code> | Indices of the read requests for settled values. |
| <code>read_request_membership_witnesses</code> | <code>[MembershipWitness; M]</code> | Membership witnesses for the settled values. |
| <code>read_request_statuses</code> | <code>[ReadRequestStatus; C]</code> | Statuses of the values being read. <code>C</code> equals <code>MAX_NOTE_HASH_READ_REQUESTS_PER_TX</code> when <code>reset_type</code> is <code>note_hash</code> ; <code>MAX_NULLIFIER_READ_REQUESTS_PER_TX</code> when <code>reset_type</code> is <code>nullifier</code> . |

There can be multiple versions of the read request reset private kernel circuit, each with a different values of `N` and `M`.

ReadRequestStatus

| Field | Type | Description |
|-------|-------------------------------------|---|
| state | [persistent] [transient] [nada] | State of the read request. |
| index | field | Index of the hint for the read request. |

Hints for Parent Secret Key Validation Request Reset Private Kernel Circuit

| Field | Type | Description |
|----------------------------|--|---|
| master_secret_keys | [field;
MAX_KEY_VALIDATION_REQUESTS_PER_TX] | Master secret to try to derive app secret keys and pub keys from. |
| app_secret_keys_generators | [field;
MAX_KEY_VALIDATION_REQUESTS_PER_TX] | App secret key generators to assist with ^. |

Hints for Transient Note Reset Private Kernel Circuit

| Field | Type | Description |
|--|---|---|
| transient_nullifier_indices | [field; MAX_NOTE_HASHES_PER_TX] | Indices of the nullifiers for transient notes. |
| nullifier_index_hints | [field; MAX_NULLIFIERS_PER_TX] | Indices of the transient_nullifier_indices for transient nullifiers. |
| encrypted_note_preimage_hash_index_hints | [field;
MAX_ENCRYPTED_NOTE_PREIMAGE_HASHES_PER_TX] | Indices of the encrypted_note_preimage_hash_contexts for transient preimage hashes. |
| log_note_hash_hints | [field; MAX_NOTE_HASHES_PER_TX] | Indices of the note_hash_contexts for transient preimage hashes. |

PublicInputs

The format aligns with the [PublicInputs](#) of the initial private kernel circuit.

Private Kernel Circuit - Tail

Requirements

The tail circuit abstains from processing individual private function calls. Instead, it incorporates the outcomes of a private kernel circuit and conducts additional processing essential for generating the final public inputs suitable for submission to the transaction pool, subsequently undergoing processing by Sequencers and Provers. The final public inputs must safeguard against revealing any private information unnecessary for the execution of public kernel circuits and rollup circuits.

Verification of the Previous Iteration

Verifying the previous kernel proof.

It verifies that the previous iteration was executed successfully with the given proof data, verification key, and public inputs, sourced from `private_inputs.previous_kernel`.

The preceding proof can be:

- Initial private kernel proof.
- Inner private kernel proof.
- Reset private kernel proof.

An inner iteration may be omitted when there's only a single private function call for the transaction. And a reset iteration can be skipped if there are no read requests and transient notes in the public inputs from the last iteration.

Ensuring the previous iteration is the last.

It checks the data within `private_inputs.previous_kernel.public_inputs.transient_accumulated_data` to ensure that no further private kernel iteration is needed.

1. The following must be empty to ensure all the private function calls are processed:

- `private_call_request_stack`

2. The following must be empty to ensure a comprehensive final reset:

- `note_hash_read_requests`
- `nullifier_read_requests`
- `key_validation_request_contexts`
- The `nullifier_counter` associated with each note hash in `note_hash_contexts`.
- The `note_hash_counter` associated with each nullifier in `nullifier_contexts`.

A `reset iteration` should ideally precede this step. Although it doesn't have to be executed immediately before the tail circuit, as long as it effectively clears the specified values.

Processing Final Outputs

Siloing values.

Siloing a value with the address of the contract generating the value ensures that data produced by a contract is accurately attributed to the correct contract and cannot be misconstrued as data created in a different contract. This circuit guarantees the following siloed values:

1. Silo `nullifiers`:

For each `nullifier` at index `i > 0` in the `nullifier_contexts` within `private_inputs`, if `nullifier.value != 0`:

```
    nullifier_contexts[i].value = hash(nullifier.contract_address, nullifier.value)
```

This process does not apply to `nullifier_contexts[0]`, which is the [hash of the transaction request](#) created by the initial private kernel circuit.

2. Silo `note_hashes`:

For each `note_hash` at index `i` in the `note_hash_contexts` within `private_inputs`, if `note_hash.value != 0`:

```
    note_hash_contexts[i].value = hash(note_nonce, siloed_hash)
```

Where:

- `note_nonce = hash(first_nullifier, index)`
 - `first_nullifier = nullifier_contexts[0].value`.
 - `index = note_hash_hints[i]`, which is the index of the same note hash within `public_inputs.note_hashes`. Where `note_hash_hints` is provided as [hints](#) via `private_inputs`.
- `siloed_hash = hash(note_hash.contract_address, note_hash.value)`

Siloing with a `note_nonce` guarantees that each final note hash is a unique value in the note hash tree.

3. Silo `l2_to_l1_messages`:

For each `l2_to_l1_message` at index `i` in `l2_to_l1_message_contexts` within `[private_inputs]`, if `l2_to_l1_message.value != 0`:

```
    l2_to_l1_message_contexts[i].value = hash(l2_to_l1_message.contract_address, version_id,  
    l2_to_l1_message.portal_contract_address, chain_id, l2_to_l1_message.value)
```

Where `version_id` and `chain_id` are defined in `public_inputs.constant_data.tx_context`.

4. Silo `unencrypted_log_hashes`:

For each `log_hash` at index `i` in the `unencrypted_log_hash_contexts` within `private_inputs`, if `log_hash.hash != 0`:

```
    unencrypted_log_hash_contexts[i].value = hash(log_hash.hash, log_hash.contract_address)
```

5. Silo `encrypted_log_hashes`:

For each `log_hash` at index `i` in the `encrypted_log_hash_contexts` within `private_inputs`, if `log_hash.hash != 0`:

```
    encrypted_log_hash_contexts[i].value = hash(log_hash.hash, contract_address_tag)
```

Where `contract_address_tag = hash(log_hash.contract_address, log_hash.randomness)`

Verifying and splitting ordered data.

The initial and inner kernel iterations may produce values in an unordered state due to the serial nature of the kernel, contrasting with the stack-based nature of code execution.

This circuit ensures the correct ordering of the following:

- `note_hashes`
- `nullifiers`
- `l2_to_l1_messages`
- `unencrypted_log_hashes`

- `encrypted_log_hashes`
- `encrypted_note_preimage_hashes`
- `public_call_requests`

In addition, the circuit split the ordered data into `non_revertible_accumulated_data` and `revertible_accumulated_data` using `min_revertible_side_effect_counter`.

1. Verify ordered `public_call_requests`:

Initialize `num_non_revertible` and `num_revertible` to 0.

For each `request` at index `i` in the `unordered public_call_request_contexts` within `private_inputs.previous_kernel.public_inputs.transient_accumulated_data`:

- Find its associated `mapped_request` in `public_call_requests[public_call_request_hints[i]]` within `public_inputs`.
 - If `request.counter < min_revertible_side_effect_counter`:
 - The `public_call_requests` is in `non_revertible_accumulated_data`.
 - `num_added = num_non_revertible`.
 - If `request.counter >= min_revertible_side_effect_counter`:
 - The `public_call_requests` is in `revertible_accumulated_data`.
 - `num_added = num_revertible`.
- If `request.call_stack_item_hash != 0`, verify that:
 - `request == mapped_request`
 - If `num_added > 0`, verify that:
 - `public_call_requests[num_added].counter < public_call_requests[num_added - 1].counter`
 - Increment `num_added` by 1: `num_(non_)revertible += 1`
- Else:
 - All the subsequent requests (`index >= i`) in `public_call_request_contexts` must be empty.
 - All the subsequent requests (`index >= num_non_revertible`) in `non_revertible_accumulated_data.public_call_requests` must be empty.
 - All the subsequent requests (`index >= num_revertible`) in `revertible_accumulated_data.public_call_requests` must be empty.

Note that requests in `public_call_requests` must be arranged in descending order to ensure the calls are executed in chronological order.

2. Verify the rest of the ordered arrays:

Initialize `num_non_revertible` and `num_revertible` to 0.

For each `note_hash_context` at index `i` in the `unordered note_hash_contexts` within `private_inputs.previous_kernel.public_inputs.transient_accumulated_data`:

- Find its associated `note_hash` in `note_hashes[note_hash_hints[i].index]` within `public_inputs`.
 - If `note_hash_context.counter < min_revertible_side_effect_counter`:
 - The `note_hashes` is in `non_revertible_accumulated_data`.
 - `num_added = num_non_revertible`.
 - If `note_hash_context.counter >= min_revertible_side_effect_counter`:
 - The `note_hashes` is in `revertible_accumulated_data`.
 - `num_added = num_revertible`.
- If `note_hash_context.value != 0`, verify that:

- `note_hash == note_hash_context.value`
- `note_hash_hints[note_hash_hints[i].index].counter_(non_)revertible == note_hash_context.counter`
- If `num_added > 0`, verify that:
 - `note_hash_hints[num_added].counter_(non_)revertible > note_hash_hints[num_added - 1].counter_(non_)revertible`
 - Increment `num_added` by 1: `num_(non_)revertible += 1`
- Else:
 - All the subsequent elements (`index >= i`) in `note_hash_contexts` must be empty.
 - All the subsequent elements (`index >= num_non_revertible`) in `non_revertible_accumulated_data.note_hashes` must be empty.
 - All the subsequent elements (`index >= num_revertible`) in `revertible_accumulated_data.note_hashes` must be empty.

Repeat the same process for `nullifiers`, `l2_to_l1_messages`, `unencrypted_log_hashes`, `encrypted_log_hashes`, and `encrypted_note_preimage_hashes`, where:

- Ordered `nullifiers` and `l2_to_l1_messages` are within `public_inputs`.
- `ordered_unencrypted_log_hashes_(non_)revertible`, `ordered_encrypted_log_hashes_(non_)revertible`, and `ordered_encrypted_note_preimage_hashes_(non_)revertible` are provided as `hints` through `private_inputs`.

While ordering could occur gradually in each kernel iteration, the implementation is much simpler and typically more efficient to be done once in the tail circuit.

Recalibrating counters.

While the `counter` of a `public_call_request` is initially assigned in the private function circuit to ensure proper ordering within the transaction, it should be modified in this step. As using `counter` values obtained from private function circuits may leak information.

The requests in the `public_call_requests` within `public_inputs` have been sorted in descending order in the previous step. This circuit recalibrates their counters through the following steps:

- The `counter` of the last non-empty request is set to 1.
- The `counter`s of the other non-empty requests are continuous values in descending order:
 - `public_call_requests[i].counter = public_call_requests[i + 1].counter + 1`

It's crucial for the `counter` of the last request to be 1, as it's assumed in the tail public kernel circuit that no storage writes have a counter 1.

Validating Public Inputs

Verifying the (non-)revertible accumulated data.

1. The following must align with the results after ordering, as verified in a previous step:

- `note_hashes`
- `nullifiers`
- `l2_to_l1_messages`

2. The `public_call_requests` must adhere to a specific order with recalibrated counters, as verified in the previous steps.

3. The hashes and lengths for all logs are accumulated as follows:

For each non-empty `log_hash` at index `i` in `ordered_unencrypted_log_hashes_(non_)revertible`, which is provided as `hints`, and the ordering was verified against the siloed hashes in previous steps:

- `accumulated_logs_hash = hash(accumulated_logs_hash, log_hash.hash)`
 - If `i == 0`: `accumulated_logs_hash = log_hash.hash`
- `accumulated_logs_length += log_hash.length`

Check the values in the `public_inputs` are correct:

- `unencrypted_logs_hash == accumulated_logs_hash`
- `unencrypted_log_preimages_length == accumulated_logs_length`

Repeat the same process for `encrypted_logs_hashes` and `encrypted_note_preimages_hashes`.

Verifying the transient accumulated data.

It ensures that all data in the `transient_accumulated_data` within `public_inputs` is empty.

Verifying other data.

This section follows the same [process](#) as outlined in the inner private kernel circuit.

In addition, it checks that the following are empty:

- `old_public_data_tree_snapshot`
- `new_public_data_tree_snapshot`

PrivateInputs

PreviousKernel

The format aligns with the [PreviousKernel](#) of the inner private kernel circuit.

Hints

Data that aids in the verifications carried out in this circuit:

| Field | Type | Description |
|--|---|---|
| <code>note_hash_hints</code> | <code>[OrderHint; MAX_NOTE_HASHES_PER_TX]</code> | Hints for ordering <code>note_hash_contexts</code> . |
| <code>nullifier_hints</code> | <code>[OrderHint; MAX_NULLIFIERS_PER_TX]</code> | Hints for ordering <code>nullifier_contexts</code> . |
| <code>public_call_request_hints</code> | <code>[field;
MAX_PUBLIC_CALL_STACK_LENGTH_PER_TX]</code> | Indices of ordered <code>public_call_request_contexts</code> . |
| <code>unencrypted_log_hash_hints</code> | <code>[OrderHint;
MAX_UNENCRYPTED_LOG_HASHES_PER_TX]</code> | Hints for ordering <code>unencrypted_log_hash_contexts</code> . |
| <code>ordered_unencrypted_log_hashes_revertible</code> | <code>[field; MAX_UNENCRYPTED_LOG_HASHES_PER_TX]</code> | Ordered revertible <code>unencrypted_log_hashes</code> . |
| <code>ordered_unencrypted_log_hashes_non_revertible</code> | <code>[field; MAX_UNENCRYPTED_LOG_HASHES_PER_TX]</code> | Ordered non-revertible <code>unencrypted_log_hashes</code> . |
| <code>encrypted_log_hash_hints</code> | <code>[OrderHint;
MAX_ENCRYPTED_LOG_HASHES_PER_TX]</code> | Hints for ordering <code>encrypted_log_hash_contexts</code> . |
| <code>ordered_encrypted_log_hashes_revertible</code> | <code>[field; MAX_ENCRYPTED_LOG_HASHES_PER_TX]</code> | Ordered revertible <code>encrypted_log_hashes</code> . |

| Field | Type | Description |
|--|--|--|
| <code>ordered_encrypted_log_hashes_non_revertible</code> | [field; <code>MAX_ENCRYPTED_LOG_HASHES_PER_TX</code>] | Ordered non-revertible <code>encrypted_log_hashes</code> . |
| <code>encrypted_note_preimage_hints</code> | [<code>OrderHint</code> ;
<code>MAX_ENCRYPTED_NOTE_PREIMAGE_HASHES_PER_TX</code>] | Hints for ordering <code>encrypted_note_preimage_hash_content</code> . |
| <code>ordered_encrypted_note_preimage_hashes_revertible</code> | [field;
<code>MAX_ENCRYPTED_NOTE_PREIMAGE_HASHES_PER_TX</code>] | Ordered revertible <code>encrypted_note_preimage_hashes</code> . |
| <code>ordered_encrypted_note_preimage_hashes_non_revertible</code> | [field;
<code>MAX_ENCRYPTED_NOTE_PREIMAGE_HASHES_PER_TX</code>] | Ordered non-revertible <code>encrypted_note_preimage_hashes</code> . |

`OrderHint`

| Field | Type | Description |
|-------------------------------------|--------------------|--|
| <code>index</code> | <code>field</code> | Index of the mapped element in the ordered array. |
| <code>counter_revertible</code> | <code>u32</code> | Counter of the element at index i in the revertible ordered array. |
| <code>counter_non_revertible</code> | <code>u32</code> | Counter of the element at index i in the non-revertible ordered array. |

PublicInputs

The format aligns with the [Public Inputs](#) of the tail public kernel circuit.

Public Kernel Circuit - Initial

🔥 DANGER

The public kernel circuits are being redesigned to accommodate the latest AVM designs. This page is therefore highly likely to change significantly.

Requirements

The initial public kernel iteration undergoes processes to prepare the necessary data for the executions of the public function calls.

Verification of the Previous Iteration

Verifying the previous kernel proof.

It verifies that the previous iteration was executed successfully with the given proof data, verification key, and public inputs, sourced from `private_inputs`.`previous_kernel`.

The preceding proof can only be:

- Tail private kernel proof.

Public Inputs Data Reset

Recalibrating counters.

While the counters outputted from the tail private kernel circuit preserve the correct ordering of the `public_call_requests`, they do not reflect the actual number of side effects each public call entails. This circuit allows the recalibration of counters for `public_call_requests`, ensuring subsequent public kernels can be executed with the correct counter range.

For each `request` at index i in the `public_call_requests` within `public_inputs`.`transient_accumulated_data`:

1. Its hash must match the corresponding item in the `public_call_requests` within the previous kernel's public inputs:

- o `request.hash ==`
`private_inputs.previous_kernel_public_inputs.public_call_requests[i].hash`

2. Its `counter_end` must be greater than its `counter_start`.
3. Its `counter_start` must be greater than the `counter_end` of the item at index `i + 1`.
4. If it's the last item, its `counter_start` must be `1`.

It's crucial for the `counter_start` of the last item to be `1`, as it's assumed in the [tail public kernel circuit](#) that no storage writes have a counter `1`.

Validating Public Inputs

Verifying the accumulated data.

It ensures that the `accumulated_data` in the `public_inputs` matches the `accumulated_data` in `private_inputs.previous_kernel.public_inputs`.

Verifying the transient accumulated data.

It ensures that all data in the `transient_accumulated_data` within `public_inputs` is empty, with the exception of the `public_call_requests`.

The values in `public_call_requests` are verified in a [previous step](#).

Verifying the constant data.

This section follows the same [process](#) as outlined in the inner private kernel circuit.

PrivateInputs

PreviousKernel

The format aligns with the `PreviousKernel`` of the tail public kernel circuit.

PublicInputs

The format aligns with the `PublicInputs` of the tail public kernel circuit.

Public Kernel Circuit - Inner

🔥 DANGER

The public kernel circuits are being redesigned to accommodate the latest AVM designs. This page is therefore highly likely to change significantly.

Requirements

In the public kernel iteration, the process involves taking a previous iteration and public call data, verifying their integrity, and preparing the necessary data for subsequent circuits to operate.

Verification of the Previous Iteration

Verifying the previous kernel proof.

It verifies that the previous iteration was executed successfully with the given proof data, verification key, and public inputs, sourced from `private_inputs.previous_kernel`.

The preceding proof can be:

- Initial public kernel proof.
- Inner public kernel proof.

Processing Public Function Call

Ensuring the function being called exists in the contract.

This section follows the same `process` as outlined in the initial private kernel circuit.

Ensuring the contract instance being called is deployed.

It verifies the public deployment of the contract instance by conducting a membership proof, where:

- The leaf is a nullifier emitting from the deployer contract, computed as `hash(deployer_address, contract_address)`, where:
 - `deployer_address` is defined in `private_inputs.public_call.contract_data`.
 - `contract_data` is defined in `private_inputs.public_call.call_stack_item`.
- The index and sibling path are provided in `contract_deployment_membership_witness` through

`private_inputs`.`public_call`.

- The root is the `nullifier_tree_root` in the `header` within `public_inputs`.`constant_data`.

Ensuring the current call matches the call request.

The top item in the `public_call_requests` of the `previous_kernel` must pertain to the current function call.

This circuit will:

1. Pop the request from the stack:

- `call_request = previous_kernel.public_inputs.transient_accumulated_data.public_call_requests.pop()`

2. Compare the hash with that of the current function call:

- `call_request.hash == public_call.call_stack_item.hash()`
- The hash of the `call_stack_item` is computed as:
 - `hash(contract_address, function_data.hash(), public_inputs.hash(), counter_start, counter_end)`
 - Where `function_data.hash()` and `public_inputs.hash()` are the hashes of the serialized field elements.

Ensuring this function is called with the correct context.

This section follows the same `process` as outlined in the inner private kernel circuit.

Verifying the public function proof.

It verifies that the public function was executed with the provided proof data, verification key, and the public inputs of the VM circuit. The result of the execution is specified in the public inputs, which will be used in subsequent steps to enforce the conditions they must satisfy.

Verifying the public inputs of the public function circuit.

It ensures the public function's intention by checking the following in

`public_call`.`call_stack_item`.`public_inputs`:

- The `header` must match the one in the `constant_data`.
- If it is a static call (`public_inputs.call_context.is_static_call == true`), it ensures that the function does not induce any state changes by verifying that the following arrays are empty:
 - `note_hashes`

- `nullifiers`
- `l2_to_l1_messages`
- `storage_writes`
- `unencrypted_log_hashes`

Verifying the counters.

It verifies that each value listed below is associated with a legitimate counter.

1. For the `call_stack_item`:

- The `counter_start` and `counter_end` must match those in the `call_request` popped from the `public_call_requests` in a previous step.

2. For items in each ordered array in `call_stack_item.public_inputs`:

- The counter of the first item must be greater than the `counter_start` of the current call.
- The counter of each subsequent item must be greater than the counter of the previous item.
- The counter of the last item must be less than the `counter_end` of the current call.

The ordered arrays include:

- `storage_reads`
- `storage_writes`

3. For the last `N` non-empty requests in `public_call_requests` within `public_inputs.transient_accumulated_data`:

- The `counter_end` of each request must be greater than its `counter_start`.
- The `counter_start` of the first request must be greater than the `counter_start` of the `call_stack_item`.
- The `counter_start` of the second and subsequent requests must be greater than the `counter_end` of the previous request.
- The `counter_end` of the last request must be less than the `counter_end` of the `call_stack_item`.

`N` is the number of non-zero hashes in the `public_call_stack_item_hashes` in `private_inputs.public_call.public_inputs`.

Validating Public Inputs

Verifying the accumulated data.

1. It verifies that the following in the `accumulated_data` align with their corresponding values in `public_call.call_stack_item.public_inputs`.
 - o `note_hashes`
 - o `nullifiers`
 - o `l2_to_l1_messages`
 - o `encrypted_logs_hash`
 - o `encrypted_log_preimages_length`
 - o `encrypted_note_preimages_hash`
 - o `encrypted_note_preimages_length`
 - o `old_public_data_tree_snapshot`
 - o `new_public_data_tree_snapshot`

Verifying the transient accumulated data.

The `transient_accumulated_data` in this circuit's `public_inputs` includes values from both the previous iterations and the `public_call`.

For each array in the `transient_accumulated_data`, this circuit verifies that it is populated with the values from the previous iterations, specifically:

- `public_inputs.transient_accumulated_data.ARRAY[0..N] == private_inputs.previous_kernel.public_inputs.transient_accumulated_data.ARRAY[0..N]`

It's important to note that the top item in the `public_call_requests` from the *previous_kernel* won't be included, as it has been removed in a *previous step*.

For the subsequent items appended after the values from the previous iterations, they constitute the values from `private_inputs.public_call.call_stack_item.public_inputs` (`public_function_public_inputs`), and must undergo the following verifications:

1. Ensure that the specified values in the following arrays match those in the corresponding arrays in the `public_function_public_inputs`:

- o `note_hash_contexts`
 - `value, counter`

- `nullifier_contexts`
 - `value, counter`
- `l2_to_l1_message_contexts`
 - `value`
- `storage_reads`
 - `value, counter`
- `storage_writes`
 - `value, counter`
- `unencrypted_log_hash_contexts`
 - `hash, length, counter`

2. For `public_call_requests`:

- The hashes align with the values in the `public_call_stack_item_hashes` within `public_function_public_inputs`, but in **reverse** order.
- The `caller_contract_address` equals the `contract_address` in `public_call .call_stack_item`.
- The `caller_context` aligns with the values in the `call_context` within `public_function_public_inputs`.

It's important that the call requests are arranged in reverse order to ensure they are executed in chronological order.

3. The `contract_address` for each non-empty item in the following arrays must equal the `storage_contract_address` defined in `public_function_public_inputs.call_context`:

- `note_hash_contexts`
- `nullifier_contexts`
- `l2_to_l1_message_contexts`
- `storage_reads`
- `storage_writes`
- `unencrypted_log_hash_contexts`

Ensuring the alignment of the contract addresses is crucial, as it is later used to **silo the values** and to establish associations with values within the same contract.

4. The *portal_contract_address* for each non-empty item in `l2_to_l1_message_contexts` must equal the *portal_contract_address* defined in `public_function_public_inputs.call_context`.

5. For each `storage_write` in `storage_writes`, verify that it is associated with an *override_counter*. The value of the *override_counter* can be:

- Zero: if the `storage_slot` does not change later in the same transaction.
- Greater than `storage_write.counter`: if the `storage_slot` is written again later in the same transaction.

Override counters are used in the tail public kernel circuit to ensure a read happens before the value is changed in a subsequent write.

Zero serves as an indicator for an unchanged update, as this value can never act as the counter of a write.

Verifying the constant data.

This section follows the same `process` as outlined in the inner private kernel circuit.

PrivateInputs

PreviousKernel

The format aligns with the `PreviousKernel` of the tail public kernel circuit.

PublicCall

Data that holds details about the current public function call.

| Field | Type | Description |
|------------------------------|----------------------------------|---|
| <code>call_stack_item</code> | <code>PublicCallStackItem</code> | Information about the current public function call. |
| <code>proof</code> | <code>Proof</code> | Proof of the public function circuit. |

| Field | Type | Description |
|--|-------------------|---|
| vk | VerificationKey | Verification key of the public function circuit. |
| bytecode_hash | field | Hash of the function bytecode. |
| contract_data | ContractInstance | Data of the contract instance being called. |
| contract_class_data | ContractClass | Data of the contract class. |
| function_leaf_membership_witness | MembershipWitness | Membership witness for the function being called. |
| contract_deployment_membership_witness | MembershipWitness | Membership witness for the deployment of the contract being called. |

PublicInputs

The format aligns with the [PublicInputs](#) of the tail public kernel circuit.

Types

PublicCallStackItem

| Field | Type | Description |
|------------------|----------------------------|---|
| contract_address | AztecAddress | Address of the contract on which the function is invoked. |
| function_data | FunctionData | Data of the function being called. |
| public_inputs | PublicFunctionPublicInputs | Public inputs of the public vm circuit. |

| Field | Type | Description |
|---------------|-------|---|
| counter_start | field | Counter at which the function call was initiated. |
| counter_end | field | Counter at which the function call ended. |

PublicFunctionPublicInputs

| Field | Type | Description |
|-------------------------------|----------------------------|---|
| call_context | CallContext | Context of the call corresponding to this function execution. |
| args_hash | field | Hash of the function arguments. |
| return_values | [field; C] | Return values of this function call. |
| note_hashes | [NoteHash ; C] | New note hashes created in this function call. |
| nullifiers | [Nullifier ; C] | New nullifiers created in this function call. |
| l2_to_l1_messages | [field; C] | New L2 to L1 messages created in this function call. |
| storage_reads | [StorageRead_ ; C] | Data read from the public data tree. |
| storage_writes | [StorageWrite ; C] | Data written to the public data tree. |
| unencrypted_log_hashes | [UnencryptedLogHash;
C] | Hashes of the unencrypted logs emitted in this function call. |
| public_call_stack_item_hashes | [field; C] | Hashes of the public function calls initiated by this function. |
| header | Header | Information about the trees used for |

| Field | Type | Description |
|----------|-------|------------------------------|
| | | the transaction. |
| chain_id | field | Chain ID of the transaction. |
| version | field | Version of the transaction. |

The above Cs represent constants defined by the protocol. Each C might have a different value from the others.

Public Kernel Circuit - Tail

🔥 DANGER

The public kernel circuits are being redesigned to accommodate the latest AVM designs. This page is therefore highly likely to change significantly.

Requirements

The tail circuit refrains from processing individual public function calls. Instead, it integrates the results of inner public kernel circuit and performs additional verification and processing necessary for generating the final public inputs.

Verification of the Previous Iteration

Verifying the previous kernel proof.

It verifies that the previous iteration was executed successfully with the given proof data, verification key, and public inputs, sourced from `private_inputs` `.previous_kernel`.

The preceding proof can only be:

- Inner public kernel proof.

Ensuring the previous iteration is the last.

The following must be empty to ensure all the public function calls are processed:

- `public_call_requests` in both `revertible_accumulated_data` and `non_revertible_accumulated_data` within `private_inputs` `.previous_kernel` `.public_inputs`.

Processing Final Outputs

Siloing values.

This section follows the same `process` as outlined in the tail private kernel circuit.

Additionally, it silos the `storage_slot` of each non-empty item in the following arrays:

- `storage_reads`
- `storage_writes`

The siloed storage slot is computed as: `hash(contract_address, storage_slot)`.

Verifying ordered arrays.

The iterations of the public kernel may yield values in an unordered state due to the serial nature of the kernel, which contrasts with the stack-based nature of code execution.

This circuit ensures the correct ordering of the following:

- note_hashes
- nullifiers
- storage_reads
- storage_writes
- ordered_unencrypted_log_hashes

1. For note_hashes, nullifiers, and ordered_unencrypted_log_hashes, they undergo the same process as outlined in the tail private kernel circuit. With the exception that the loop starts from index offset + i, where offset is the number of non-zero values in the note_hashes and nullifiers arrays within private_inputs.previous_kernel.public_inputs.accumulated_data.
2. For storage_reads, an ordered_storage_reads and storage_read_hints are provided as hints through private_inputs. This circuit checks that:

For each read at index i in storage_reads[i], the associated mapped_read is at ordered_storage_reads[storage_read_hints[i]].

- If read.is_empty() == false, verify that:
 - All values in read align with those in mapped_read:
 - read.contract_address == mapped_read.contract_address
 - read.storage_slot == mapped_read.storage_slot
 - read.value == mapped_read.value
 - read.counter == mapped_read.counter
 - If i > 0, verify that:
 - mapped_read[i].counter > mapped_read[i - 1].counter
 - Else:
 - All the subsequent reads (index >= i) in both storage_reads and ordered_storage_reads must be empty.
3. For storage_writes, an ordered_storageWrites and storage_write_hints are provided as hints through private_inputs. The verification is the same as the process for storage_reads.

Verifying public data snaps.

The public_data_snaps is provided through private_inputs, serving as hints for storage_reads to prove that the value in the tree aligns with the read operation. For storage_writes, it substantiates the presence or absence of the

storage slot in the public data tree.

A `public_data_snap` contains:

- A `storage_slot` and its `value`.
- An `override_counter`, indicating the counter of the first `storage_write` that writes to the storage slot. Zero if the storage slot is not written in this transaction.
- A flag `exists` indicating its presence or absence in the public data tree.

This circuit ensures the uniqueness of each snap in `public_data_snaps`. It verifies that:

For each snap at index `i`, where `i > 0`:

- If `snap.is_empty() == false`
 - `snap.storage_slot > public_data_snaps[i - 1].storage_slot`

It is crucial for each snap to be unique, as duplicated snaps would disrupt a group of writes for the same storage slot. This could enable the unauthorized act of reading the old value after it has been updated.

Grouping storage writes.

To facilitate the verification of `storage_reads` and streamline `storage_writes`, it is imperative to establish connections between writes targeting the same storage slot. Furthermore, the first write in a group must be linked to a `public_data_snap`, ensuring the dataset has progressed from the right initial state.

A new field, `prev_counter`, is incorporated to the `ordered_storage_writes` to indicate whether each write has a preceding snap or write. Another field, `exists`, is also added to signify the presence or absence of the storage slot in the tree.

1. For each `snap` at index `i` in `public_data_snaps`:

- Skip the remaining steps if it is empty or if its `override_counter` is `0`.
- Locate the `write` at `ordered_storage_writes[storage_write_indices[i]]`.
- Verify the following:
 - `write.storage_slot == snap.storage_slot`
 - `write.counter == snap.override_counter`
 - `write.prev_counter == 0`
- Update the hints in `write`:
 - `write.prev_counter = 1`
 - `write.exists = snap.exists`

The value `1` can be utilized to signify a preceding `snap`, as this value can never serve as the counter of a `storage_write`. Because the `counter_start` for the first public function call must be `1`, the counters for all

subsequent side effects should exceed this initial value.

2. For each `write` at index `i` in `ordered_storage_writes`:

- Skip the remaining steps if its `next_counter` is `0`.
- Locate the `next_write` at `ordered_storage_writes[next_storage_write_indices[i]]`.
- Verify the following:
 - `write.storage_slot == next_write.storage_slot`
 - `write.next_counter == next_write.counter`
 - `write.prev_counter == 0`
- Update the hints in `next_write`:
 - `next_write.prev_counter = write.counter`
 - `next_write.exists = write.exists`

3. Following the previous two steps, verify that all non-empty writes in `ordered_storage_writes` have a non-zero `prev_counter`.

Verifying storage reads.

A storage read can be reading:

- An uninitialized storage slot: the value is zero. There isn't a leaf in the public data tree representing its storage slot, nor in the `storage_writes`.
- An existing storage slot: written in a prior successful transaction. The value being read is the value in the public data tree.
- An updated storage slot: initialized or updated in the current transaction. The value being read is in a `storage_write`.

For each non-empty `read` at index `i` in `ordered_storage_reads`, it must satisfy one of the following conditions:

1. If reading an uninitialized or an existing storage slot, the value is in a `snap`:

- Locate the `snap` at `public_data_snaps[persistent_read_hints[i]]`.
- Verify the following:
 - `read.storage_slot == snap.storage_slot`
 - `read.value == snap.value`
 - `(read.counter < snap.override_counter) | (snap.override_counter == 0)`
- If `snap.exists == false`:
 - `read.value == 0`

Depending on the value of the `exists` flag in the `snap`, verify its presence or absence in the public data tree:

- If `exists` is true:
 - It must pass a membership check on the leaf.

- If `exists` is false:
 - It must pass a non-membership check on the low leaf. The preimage of the low leaf is at `storage_read_low_leaf_preimages[i]`.

The (non-)membership checks are executed against the root in `old_public_data_tree_snapshot`. The membership witnesses for the leaves are in `storage_read_membership_witnesses`, provided as `hints` through `private_inputs`.

2. If reading an updated storage slot, the value is in a `storage_write`:

- Locates the `storage_write` at `ordered_storageWrites[transient_read_hints[i]]`.
- Verify the following:
 - `read.storage_slot == storage_write.storage_slot`
 - `read.value == storage_write.value`
 - `read.counter > storage_write.counter`
 - `(read.counter < storage_write.next_counter) | (storage_write.next_counter == 0)`

A zero `next_counter` indicates that the value is not written again in the transaction.

Updating the public data tree.

It updates the public data tree with the values in `storage_writes`. The `latest_root` of the tree is `old_public_data_tree_snapshot.root`.

For each non-empty `write` at index `i` in `ordered_storage_writes`, the circuit processes it base on its type:

1. Transient write.

If `write.next_counter != 0`, the same storage slot is written again by another storage write that occurs later in the same transaction. This transient `write` can be ignored as the final state of the tree won't be affected by it.

2. Updating an existing storage slot.

For a non-transient `write` (`write.next_counter == 0`), if `write.exists == true`, it is updating an existing storage slot. The circuit does the following for such a write:

- Performs a membership check, where:
 - The leaf if for the existing storage slot.
 - `leaf.storage_slot = write.storage_slot`
 - The old value is the value in a `snap`:
 - `leaf.value = public_data_snaps[public_data_snap_indices[i]].value`
 - The index and the sibling path are in `storage_write_membership_witnesses`, provided as `hints` through

`private_inputs`.

- The root is the `latest_root` after processing the previous write.
- Derives the `latest_root` for the `latest_public_data_tree` with the updated leaf, where `leaf.value = write.value`.

3. Creating a new storage slot.

For a non-transient `write` (`write.next_counter == 0`), if `write.exists == false`, it is initializing a storage slot. The circuit adds it to a subtree:

- Perform a membership check on the low leaf in the `latest_public_data_tree` and in the subtree. One check must succeed.
 - The low leaf preimage is at `storage_write_low_leaf_preimages[i]`.
 - The membership witness for the public data tree is at `storage_write_membership_witnesses[i]`.
 - The membership witness for the subtree is at `subtree_membership_witnesses[i]`.
 - The above are provided as `hints` through `private_inputs`.
- Update the low leaf to point to the new leaf:
 - `low_leaf.next_slot = write.storage_slot`
 - `low_leaf.next_index = old_public_data_tree_snapshot.next_available_leaf_index + number_of_new_leaves`
- If the low leaf is in the `latest_public_data_tree`, derive the `latest_root` from the updated low leaf.
- If the low leaf is in the subtree, derive the `subtree_root` from the updated low leaf.
- Append the new leaf to the subtree. Derive the `subtree_root`.
- Increment `number_of_new_leaves` by 1.

The subtree and `number_of_new_leaves` are initialized to empty and 0 at the beginning of the process.

After all the storage writes are processed:

- Batch insert the subtree to the public data tree.
 - The insertion index is `old_public_data_tree_snapshot.next_available_leaf_index`.
- Verify the following:
 - `latest_root == new_public_data_tree_snapshot.root`
 - `new_public_data_tree_snapshot.next_available_leaf_index == old_public_data_tree_snapshot.next_available_leaf_index + number_of_new_leaves`

Validating Public Inputs

Verifying the accumulated data.

1. The following must align with the results after siloing, as verified in a [previous step](#):

- `l2_to_l1_messages`

2. The following must align with the results after ordering, as verified in a [previous step](#):

- `note_hashes`
- `nullifiers`

3. The hashes and lengths for unencrypted logs are accumulated as follows:

Initialize `accumulated_logs_hash` to be the `unencrypted_logs_hash` within `private_inputs.previous_kernel.[public_inputs].accumulated_data`.

For each non-empty `log_hash` at index `i` in `ordered_unencrypted_log_hashes`, which is provided as [hints](#), and the [ordering](#) was verified against the [siloed hashes](#) in previous steps:

- `accumulated_logs_hash = hash(accumulated_logs_hash, log_hash.hash)`
- `accumulated_logs_length += log_hash.length`

Check the values in the `public_inputs` are correct:

- `unencrypted_logs_hash == accumulated_logs_hash`
- `unencrypted_log_preimages_length == accumulated_logs_length`

4. The following is referenced and verified in a [previous step](#):

- `old_public_data_tree_snapshot`
- `new_public_data_tree_snapshot`

Verifying the transient accumulated data.

It ensures that the transient accumulated data is empty.

Verifying the constant data.

This section follows the same [process](#) as outlined in the inner private kernel circuit.

PrivateInputs

PreviousKernel

| Field | Type | Description |
|----------------------------|--|------------------------------|
| <code>public_inputs</code> | PublicKernelPublicInputs | Public inputs of the proof. |
| <code>proof</code> | <code>Proof</code> | Proof of the kernel circuit. |

| Field | Type | Description |
|---------------------------------|--------------------------------|--|
| <code>vk</code> | <code>VerificationKey</code> | Verification key of the kernel circuit. |
| <code>membership_witness</code> | <code>MembershipWitness</code> | Membership witness for the verification key. |

Hints

Data that aids in the verifications carried out in this circuit:

| Field | Type | Description |
|---|--------------------------------------|--|
| <code>note_hash_indices</code> | <code>[field; C]</code> | Indices of <code>note_hashes</code> for <code>note_hash_contexts</code> . <code>C</code> equals the length of <code>note_hashes</code> . |
| <code>note_hash_hints</code> | <code>[field; C]</code> | Indices of <code>note_hash_contexts</code> for ordered <code>note_hashes</code> . <code>C</code> equals the length of <code>note_hash_contexts</code> . |
| <code>nullifier_hints</code> | <code>[field; C]</code> | Indices of <code>nullifier_contexts</code> for ordered <code>nullifiers</code> . <code>C</code> equals the length of <code>nullifier_contexts</code> . |
| <code>ordered_unencrypted_log_hashes</code> | <code>[field; C]</code> | Ordered <code>unencrypted_log_hashes</code> . <code>C</code> equals the length of <code>unencrypted_log_hashes</code> . |
| <code>unencrypted_log_hash_hints</code> | <code>[field; C]</code> | Indices of <code>ordered_unencrypted_log_hashes</code> for <code>unencrypted_log_hash_contexts</code> . <code>C</code> equals the length of <code>unencrypted_log_hash_contexts</code> . |
| <code>ordered_storage_reads</code> | <code>[StorageReadContext; C]</code> | Ordered <code>storage_reads</code> . <code>C</code> equals the length of <code>storage_reads</code> . |
| <code>storage_read_hints</code> | <code>[field; C]</code> | Indices of reads for <code>ordered_storage_reads</code> . <code>C</code> equals the length of <code>storage_reads</code> . |

| Field | Type | Description |
|--|--|---|
| <code>ordered_storage_writes</code> | <code>[StorageWriteContext; C]</code> | Ordered <code>storage_writes</code> . <code>C</code> equals the length of <code>storage_writes</code> . |
| <code>storage_write_hints</code> | <code>[field; C]</code> | Indices of writes for <code>ordered_storage_writes</code> . <code>C</code> equals the length of <code>storage_writes</code> . |
| <code>public_data_snaps</code> | <code>[PublicDataSnap; C]</code> | Data that aids verification of storage reads and writes. <code>C</code> equals the length of <code>ordered_storage_writes</code> + <code>ordered_storage_reads</code> . |
| <code>storage_write_indices</code> | <code>[field; C]</code> | Indices of <code>ordered_storage_writes</code> for <code>public_data_snaps</code> . <code>C</code> equals the length of <code>public_data_snaps</code> . |
| <code>transient_read_hints</code> | <code>[field; C]</code> | Indices of <code>ordered_storage_writes</code> for transient reads. <code>C</code> equals the length of <code>ordered_storage_reads</code> . |
| <code>persistent_read_hints</code> | <code>[field; C]</code> | Indices of <code>ordered_storage_writes</code> for persistent reads. <code>C</code> equals the length of <code>ordered_storage_reads</code> . |
| <code>public_data_snap_indices</code> | <code>[field; C]</code> | Indices of <code>public_data_snaps</code> for persistent write. <code>C</code> equals the length of <code>ordered_storage_writes</code> . |
| <code>storage_read_low_leaf_preimages</code> | <code>[PublicDataLeafPreimage; C]</code> | Preimages for public data leaf. <code>C</code> equals the length of <code>ordered_storage_writes</code> . |
| <code>storage_read_membership_witnesses</code> | <code>[MembershipWitness; C]</code> | Membership witnesses for persistent reads. <code>C</code> equals the length of <code>ordered_storage_writes</code> . |

| Field | Type | Description |
|---|--|---|
| <code>storage_write_low_leaf_preimages</code> | <code>[PublicDataLeafPreimage; C]</code> | Preimages for public data. <code>C</code> equals the length of <code>ordered_storage_writes</code> . |
| <code>storage_write_membership_witnesses</code> | <code>[MembershipWitness; C]</code> | Membership witnesses for public data tree. <code>C</code> equals the length of <code>ordered_storage_writes</code> . |
| <code>subtree_membership_witnesses</code> | <code>[MembershipWitness; C]</code> | Membership witnesses for the public data subtree. <code>C</code> equals the length of <code>ordered_storage_writes</code> . |

Public Inputs

| Field | Type | Description |
|--|---|---|
| <code>constant_data</code> | <code>ConstantData</code> | |
| <code>revertible_accumulated_data</code> | <code>RevertibleAccumulatedData</code> | |
| <code>non_revertible_accumulated_data</code> | <code>NonRevertibleAccumulatedData</code> | |
| <code>transient_accumulated_data</code> | <code>TransientAccumulatedData</code> | |
| <code>old_public_data_tree_snapshot</code> | <code>[TreeSnapshot]</code> | Snapshot of the public data tree prior to this transaction. |
| <code>new_public_data_tree_snapshot</code> | <code>[TreeSnapshot]</code> | Snapshot of the public data tree after this transaction. |

ConstantData

These are constants that remain the same throughout the entire transaction. Its format aligns with the `ConstantData` of the initial private kernel circuit.

RevertibleAccumulatedData

Data accumulated during the execution of the transaction.

| Field | Type | Description |
|----------------------------------|-------------------------------|---|
| note_hashes | [field; C] | Note hashes created in the transaction. |
| nullifiers | [field; C] | Nullifiers created in the transaction. |
| l2_to_l1_messages | [field; C] | L2-to-L1 messages created in the transaction. |
| unencrypted_logs_hash | field | Hash of the accumulated unencrypted logs. |
| unencrypted_log_preimages_length | field | Length of all unencrypted log preimages. |
| encrypted_logs_hash | field | Hash of the accumulated encrypted logs. |
| encrypted_log_preimages_length | field | Length of all encrypted log preimages. |
| encrypted_note_preimages_hash | field | Hash of the accumulated encrypted note preimages. |
| encrypted_note_preimages_length | field | Length of all encrypted note preimages. |
| public_call_requests | [PublicCallRequestContext; C] | Requests to call public functions. |

The above C's represent constants defined by the protocol. Each C might have a different value from the others.

NonRevertibleAccumulatedData

Data accumulated during the execution of the transaction.

| Field | Type | Description |
|-------------|------------|---|
| note_hashes | [field; C] | Note hashes created in the transaction. |

| Field | Type | Description |
|----------------------------------|-------------------------------|---|
| nullifiers | [field; C] | Nullifiers created in the transaction. |
| l2_to_l1_messages | [field; C] | L2-to-L1 messages created in the transaction. |
| unencrypted_logs_hash | field | Hash of the accumulated unencrypted logs. |
| unencrypted_log_preimages_length | field | Length of all unencrypted log preimages. |
| encrypted_logs_hash | field | Hash of the accumulated encrypted logs. |
| encrypted_log_preimages_length | field | Length of all encrypted log preimages. |
| encrypted_note_preimages_hash | field | Hash of the accumulated encrypted note preimages. |
| encrypted_note_preimages_length | field | Length of all encrypted note preimages. |
| public_call_requests | [PublicCallRequestContext; C] | Requests to call public functions. |

The above C's represent constants defined by the protocol. Each C might have a different value from the others.

TransientAccumulatedData

| Field | Type | Description |
|---------------------------|---------------------------|--|
| note_hash_contexts | [NoteHashContext; C] | Note hashes with extra data aiding verification. |
| nullifier_contexts | [NullifierContext; C] | Nullifiers with extra data aiding verification. |
| l2_to_l1_message_contexts | [L2toL1MessageContext; C] | L2-to-L1 messages with extra data aiding verification. |
| storage_reads | [StorageRead; C] | Reads of the public data. |

| Field | Type | Description |
|-----------------------------|--------------------------------|----------------------------|
| <code>storage_writes</code> | <code>[StorageWrite; C]</code> | Writes of the public data. |

The above `C`s represent constants defined by the protocol. Each `C` might have a different value from the others.

Types

TreeSnapshot

| Field | Type | Description |
|--|--------------------|-----------------------------------|
| <code>root</code> | <code>field</code> | Root of the tree. |
| <code>next_available_leaf_index</code> | <code>field</code> | The index to insert new value to. |

StorageRead

| Field | Type | Description |
|-------------------------------|---------------------------|-------------------------------------|
| <code>contract_address</code> | <code>AztecAddress</code> | Address of the contract. |
| <code>storage_slot</code> | <code>field</code> | Storage slot. |
| <code>value</code> | <code>field</code> | Value read from the storage slot. |
| <code>counter</code> | <code>u32</code> | Counter at which the read happened. |

StorageWrite

| Field | Type | Description |
|-------------------------------|---------------------------|--|
| <code>contract_address</code> | <code>AztecAddress</code> | Address of the contract. |
| <code>storage_slot</code> | <code>field</code> | Storage slot. |
| <code>value</code> | <code>field</code> | New value written to the storage slot. |

| Field | Type | Description |
|---------|------|--------------------------------------|
| counter | u32 | Counter at which the write happened. |

StorageReadContext

| Field | Type | Description |
|------------------|--------------|-------------------------------------|
| contract_address | AztecAddress | Address of the contract. |
| storage_slot | field | Storage slot. |
| value | field | Value read from the storage slot. |
| counter | u32 | Counter at which the read happened. |

StorageWriteContext

| Field | Type | Description |
|------------------|--------------|--|
| contract_address | AztecAddress | Address of the contract. |
| storage_slot | field | Storage slot. |
| value | field | New value written to the storage slot. |
| counter | u32 | Counter at which the write happened. |
| prev_counter | field | Counter of the previous write to the storage slot. |
| next_counter | field | Counter of the next write to the storage slot. |
| exists | bool | A flag indicating whether the storage slot is in the public data tree. |

PublicDataSnap

| Field | Type | Description |
|------------------|-------|--|
| storage_slot | field | Storage slot. |
| value | field | Value of the storage slot. |
| override_counter | field | Counter at which the storage_slot is first written in the transaction. |
| exists | bool | A flag indicating whether the storage slot is in the public data tree. |

PublicDataLeafPreimage

| Field | Type | Description |
|--------------|-------|--------------------------------|
| storage_slot | field | Storage slot. |
| value | field | Value of the storage slot. |
| next_slot | field | Storage slot of the next leaf. |
| next_index | field | Index of the next leaf. |

PublicCallRequestContext

| Field | Type | Description |
|-------------------------|---------------|---|
| call_stack_item_hash | field | Hash of the call stack item. |
| counter | u32 | Counter at which the request was made. |
| caller_contract_address | AztecAddress | Address of the contract calling the function. |
| caller_context | CallerContext | Context of the contract calling the function. |

Rollup Circuits

Overview

Together with the [validating light node](#), the rollup circuits must ensure that incoming blocks are valid, that state is progressed correctly, and that anyone can rebuild the state.

To support this, we construct a single proof for the entire block, which is then verified by the validating light node. This single proof consist of three main components: It has **two** sub-trees for transactions, and **one** tree for L1 to L2 messages. The two transaction trees are then merged into a single proof and combined with the roots of the message tree to form the final proof and output. Each of these trees are built by recursively combining proofs from a lower level of the tree. This structure allows us to keep the workload of each individual proof small, while making it very parallelizable. This works very well for the case where we want many actors to be able to participate in the proof generation.

Note that we have two different types of "merger" circuits, depending on what they are combining.

For transactions we have:

- The `merge` rollup
 - Merges two rollup proofs of either `base` or `merge` and constructs outputs for further proving
- The `root` rollup
 - Merges two rollup proofs of either `base` or `merge` and constructs outputs for L1

And for the message parity we have:

- The `root_parity` circuit
 - Merges N `root` or `base_parity` proofs
- The `base_parity` circuit
 - Merges N L1 to L2 messages in a subtree

In the diagram the size of the tree is limited for demonstration purposes, but a larger tree would have more layers of merge rollups proofs. Exactly how many layers and what combination of `base` and/or `merge` circuits are consumed is based on filling a [wonky tree](#) with N transactions. Circles mark the different types of proofs, while squares mark the different circuit types.

To understand what the circuits are doing and what checks they need to apply it is useful to understand what data is going into the circuits and what data is coming out.

Below is a figure of the data structures thrown around for the block proof creation. Note that the diagram does not include much of the operations for kernels, but mainly the data structures that are used for the rollup circuits.

COMBINEDACCUMULATEDDATA

Note that the `CombinedAccumulatedData` contains elements that we won't be using throughout the rollup circuits. However, as the data is used for the kernel proofs (when it is build recursively), we will include it here anyway.

Since the diagram can be quite overwhelming, we will go through the different data structures and what they are used for along with the three (3) different rollup circuits.

Higher-level tasks

Before looking at the circuits individually, it can however be a good idea to recall the reason we had them in the first place. For this, we are especially interested in the tasks that span multiple circuits and proofs.

State consistency

While the individual kernels are validated on their own, they might rely on state changes earlier in the block. For the block to be correctly validated, this means that when validating kernel n , it must be executed on top of the state after all kernels $< n$ have been applied. For example, when kernel 3

is executed, it must be executed on top of the state after kernels 0, 1 and 2 have been applied. If this is not the case, the kernel proof might be valid, but the state changes invalid which could lead to double spends.

It is therefore of the highest importance that the circuits ensure that the state is progressed correctly across circuit types and proofs. Logically, taking a few of the kernels from the above should be executed/proven as shown below, k_n applied on top of the state that applied k_{n-1}

State availability

To ensure that state is made available, we could broadcast all of a block's input data as public inputs of the final root rollup proof, but a proof with so many public inputs would be very expensive to verify onchain.

Instead, we can reduce the number of public inputs by committing to the block's body and iteratively "build" up the commitment at each rollup circuit iteration. At the very end, we will have a commitment to the transactions that were included in the block (`TxsHash`), the messages that were sent from L2 to L1 (`OutHash`) and the messages that were sent from L1 to L2 (`InHash`).

To check that the body is published an Aztec node can simply reconstruct the hashes from available data. Since we define finality as the point where the block is validated and included in the state of the [validating light node](#), we can define a block as being "available" if the validating light node can reconstruct the commitment hashes.

Since the `InHash` is directly computed by the `Inbox` contract on L1, the data is obviously available to the contract without doing any more work. Furthermore, the `OutHash` is computed from a subset of the data in `TxsHash` so if it is possible to reconstruct `TxsHash` it is also possible to reconstruct `OutHash`.

Since we strive to minimize the compute requirements to prove blocks, we amortize the commitment cost across the full tree. We can do so by building merkle trees of partial "commitments", whose roots are ultimately computed in the final root rollup circuit. Below, we outline the `TxsHash` merkle tree that is based on the `TxEffec...ts` and a `OutHash` which is based on the `l2_to_l1_msgs` (cross-chain messages) for each transaction, with four transactions in this rollup. While the `TxsHash` implicitly includes the `OutHash` we need it separately such that it can be passed to the `Outbox` for consumption by the portals with minimal work.

While the `TxsHash` merely require the data to be published and known to L1, the `InHash` and `OutHash` needs to be computable on L1 as well. This reason require them to be efficiently computable on L1 while still being non-horrible inside a snark - leading us to rely on SHA256.

The L2 to L1 messages from each transaction form a variable height tree. In the diagram above, transactions 0 and 3 have four messages, so require a tree with two layers, whereas the others only have two messages and so require a single layer tree. The base rollup calculates the root of this tree and passes it as the to the next layer. Merge rollups simply hash both of these roots together and pass it up as the `OutHash`.

Next Steps

Base Rollup

The base rollup circuit is the most complex of the rollup circuits, as it has to interpret the output data of a kernel proof and perform the state updates and transaction validation...

Merge Rollup

The Merge rollup circuit is our in-between circuit, it doesn't need to perform any state updates, but mainly check the consistency of its inputs.

L1 to L2 Message Parity

To support easy consumption of L1 to L2 messages inside the proofs, we need to convert the tree of messages to a snark-friendly format.

Root Rollup

The root rollup circuit is our top circuit, it applies the state changes passed through its children and the cross-chain messages. Essentially, it is the last step that allows us to ...

Base Rollup

The base rollup circuit is the most complex of the rollup circuits, as it has to interpret the output data of a kernel proof and perform the state updates and transaction validation. While this makes the data structures complex to follow, the goal of the circuit is fairly straight forward:

Take `BaseRollupInputs` as an input value, and transform it to `BaseOrMergeRollupPublicInputs` as an output value while making sure that the validity conditions are met.

Overview

Below is a subset of the figure from [earlier](#) (granted, not much is removed). The figure shows the data structures related to the Base Rollup circuit.



TODO

Fee structs and contract deployment structs will need to be revised, in line with newer ideas.

Validity Conditions

```
def BaseRollupCircuit(  
    state_diff_hints: StateDiffHints,  
    historical_header_membership_witnesses: HeaderMembershipWitness,  
    kernel_data: KernelData,  
    partial: PartialStateReference,  
    constants: ConstantRollupData,  
) -> BaseOrMergeRollupPublicInputs:
```


Merge Rollup

The Merge rollup circuit is our in-between circuit, it doesn't need to perform any state updates, but mainly check the consistency of its inputs.

Overview

Below is a subset of the data structures figure from earlier for easy reference.

Validity Conditions

```
def MergeRollupCircuit(
    left: ChildRollupData,
    right: ChildRollupData
) -> BaseOrMergeRollupPublicInputs:
    assert left.proof.is_valid(left.public_inputs)
    assert right.proof.is_valid(right.public_inputs)

    assert left.public_inputs.constants == right.public_inputs.constants
    assert left.public_inputs.end == right.public_inputs.start
    assert left.public_inputs.num_txs >=
        right.public_inputs.num_txs

    return BaseOrMergeRollupPublicInputs(
        type=1,
        num_txs=left.public_inputs.num_txs +
        right.public_inputs.num_txs,
        txs_effect_hash=SHA256(left.public_inputs.txs_effect_hash
        | right.public_inputs.txs_effect_hash),
        out_hash=SHA256(left.public_inputs.out_hash |
        right.public_inputs.out_hash),
        start=left.public_inputs.start,
```


L1 to L2 Message Parity

To support easy consumption of L1 to L2 messages inside the proofs, we need to convert the tree of messages to a snark-friendly format.

If you recall back in [L1 smart contracts](#) we were building a message tree on the L1. We used SHA256 to compute the tree which is cheap to compute on L1. As SHA256 is not snark-friendly, weak devices would not be able to prove inclusion of messages in the tree.

This circuit is responsible for converting the tree such that users can easily build the proofs. We essentially use this circuit to front-load the work needed to prove the inclusion of messages in the tree. As earlier we are using a tree-like structure. Instead of having a `base`, `merge` and `root` circuits, we will have only `base` and `root` parity circuits. We only need these two, since what would have been the `merge` is doing the same as the `root` for this case.

The output of the "combined" circuit will be the `converted_root` which is the root of the snark-friendly message tree. And the `sha_root` which must match the root of the sha256 message tree from the L1 Inbox. The circuit computes the two trees using the same inputs, and then we ensure that the elements of the trees match the inbox later in the [state transitioner](#). It proves parity of the leaves in the two trees.

The logic of the circuits is quite simple - build both a SHA256 and a snark-friendly tree from the same inputs. For optimization purposes, it can be useful to have the layers take more than 2 inputs to increase the task of every layer. If each just take 2 inputs, the overhead of recursing through the layers might be higher than the actual work done. Recall that all the inputs are already chosen by the L1, so we don't need to worry about which to chose.

```
def base_parity_circuit(inputs: BaseParityInputs) ->
```


Root Rollup

The root rollup circuit is our top circuit, it applies the state changes passed through its children and the cross-chain messages. Essentially, it is the last step that allows us to prove that the state transition function $\mathcal{T}(S, B) \mapsto S'$ was applied correctly for a state S and a block B . Note, that the root rollup circuit's public inputs do not comprise the block entirely as it would be too costly to verify. Given a `ProvenBlock` and proof a node can derive the public inputs and validate the correctness of the state progression.

For rollup purposes, the node we want to convince of the correctness is the [validating light node](#) that we put on L1. We will cover it in more detail in the [cross-chain communication](#) section.

⚠️ SQUISHERS

This might practically happen through a series of "squisher" circuits that will wrap the proof in another proof that is cheaper to verify on-chain. For example, wrapping a ultra-plonk proof in a standard plonk proof.

Overview

Validity Conditions

```
def RootRollupCircuit(  
    l1_to_l2_roots: RootParityInput,  
    l1_to_l2_msgs_sibling_path: List[Fr],  
    parent: Header,  
    parent_sibling_path: List[Fr],
```

The `RootRollupPublicInputs` can then be used together with `Body` to build a `ProvenBlock` which can be used to convince the [validating light node](#) of state progression.

Aztec (Public) Virtual Machine

The Aztec Virtual Machine (AVM) executes the public section of a transaction.

Introduction

REFERENCE

Many terms and definitions are borrowed from the [Ethereum Yellow Paper](#).

An Aztec transaction may include one or more **public execution requests**. A public execution request is a request to execute a specified contract's public bytecode given some arguments. Execution of a contract's public bytecode is performed by the **Aztec Virtual Machine (AVM)**.

A public execution request may originate from a public call enqueued by a transaction's private segment (`enqueuedPublicFunctionCalls`), or from a public **fee preparation** or **fee distribution** call.

In order to execute public contract bytecode, the AVM requires some context. An **execution context** contains all information necessary to initiate AVM execution, including the relevant contract's bytecode and all state maintained by the AVM. A **contract call** initializes an execution context and triggers AVM execution within that context.

Instruction-by-instruction, the AVM **executes** the bytecode specified in its context. An **instruction** is a decoded bytecode entry that, when executed, modifies the AVM's execution context (in particular its **state**) according to the instruction's definition in the "**AVM Instruction Set**". Execution within a context ends when the AVM encounters a **halt**.

During execution, additional contract calls may be made. While an **initial contract call** initializes a new execution context directly from a public execution request, a **nested contract call** occurs *during* AVM execution and is triggered by a **contract call instruction** (`CALL`, or `STATICCALL`). It initializes a new execution context (**nested**

context) from the current one (calling context) and triggers execution within it. When nested call's execution completes, execution proceeds in the calling context.

A caller is a contract call's initiator. The caller of an initial contract call is an Aztec sequencer. The caller of a nested contract call is the AVM itself executing in the calling context.

Outline

- **State**: the state maintained by the AVM
- **Memory model**: the AVM's type-tagged memory model
- **Execution context**: the AVM's execution context and its initialization for initial contract calls
- **Execution**: control flow, gas tracking, normal halting, and exceptional halting
- **Nested contract calls**: the initiation of a contract call from an instruction as well as the processing of nested execution results, gas refunds, and state reverts
- **Instruction set**: the list of all instructions supported by the AVM
- **AVM Circuit**: the AVM as a SNARK circuit for proving execution

The sections prior to the "AVM Circuit" are meant to provide a high-level definition of the Aztec Virtual Machine as opposed to a specification of its SNARK implementation. They therefore mostly omit SNARK or circuit-centric verbiage except when particularly relevant to the high-level architecture.

For an explanation of the AVM's bytecode, refer to "[AVM Bytecode](#)".

Public contract bytecode

A contract's public bytecode is a series of execution instructions for the AVM. Refer to the "[AVM Instruction Set](#)" for the details of all supported instructions along with how

they modify AVM state.

The entirety of a contract's public code is represented as a single block of bytecode with a maximum of `MAX_PUBLIC_INSTRUCTIONS_PER_CONTRACT` ($2^{15} = 32768$) instructions. The mechanism used to distinguish between different "functions" in an AVM bytecode program is left as a higher-level abstraction (e.g. similar to Solidity's concept of a function selector).

::: warning Ultimately, function selectors *may* be removed as an enshrined protocol mechanism as described above. For now, each public function on a contract has a distinct bytecode that can be selected for execution via a function selector. :::

See the [Bytecode Validation Circuit](#) to see how a contract's bytecode can be validated and committed to.

State

This section describes the types of state maintained by the AVM.

Machine State

Machine state is transformed on an instruction-per-instruction basis. Each execution context has its own machine state.

MachineState

| Field | Type | Description |
|--------------------------------|----------------------------------|---|
| <code>l2GasLeft</code> | <code>field</code> | Tracks the amount of L2 gas remaining at any point during execution. Initialized from contract call arguments. |
| <code>daGasLeft</code> | <code>field</code> | Tracks the amount of DA gas remaining at any point during execution. Initialized from contract call arguments. |
| <code>pc</code> | <code>field</code> | Index into the contract's bytecode indicating which instruction to execute. Initialized to 0 during context initialization. |
| <code>internalCallStack</code> | <code>Vector<field></code> | A stack of program counters pushed to and popped from by <code>INTERNALCALL</code> and <code>INTERNALRETURN</code> instructions. Initialized as empty during context initialization. |
| <code>memory</code> | <code>[field; 2^32]</code> | A 2^{32} entry memory space accessible by user code (AVM instructions). All 2^{32} entries are assigned default value 0 during context initialization. See " Memory Model " for a complete description of AVM memory. |

World State

AVM's access to Aztec State

Aztec's global state is implemented as a few merkle trees. These trees are exposed to the AVM as follows:

| State | Tree | Merkle Tree Type | AVM Access |
|-------------------|-----------------------|------------------|---|
| Public Storage | Public Data Tree | Updatable | membership-checks (latest), reads, writes |
| Note Hashes | Note Hash Tree | Append-only | membership-checks (start-of-block), appends |
| Nullifiers | Nullifier Tree | Indexed | membership-checks (latest), appends |
| L1-to-L2 Messages | L1-to-L2 Message Tree | Append-only | membership-checks (start-of-block) |
| Headers | Archive Tree | Append-only | membership-checks, leaf-preimage-reads |
| Contracts* | - | - | - |

* As described in "[Contract Deployment](#)", contracts are not stored in a dedicated tree. A [contract class](#) is [represented](#) as an unencrypted log containing the [ContractClass](#) structure (which contains the bytecode) and a nullifier representing the class identifier. A [contract instance](#) is [represented](#) as an unencrypted log containing the [ContractInstance](#) structure and a nullifier representing the contract address.

AVM World State

The AVM does not directly modify Aztec state. Instead, it stages modifications to be applied later pending successful execution. As part of each execution context, the AVM maintains **world state** which is a representation of Aztec state that includes *staged* modifications.

As the AVM executes contract code, instructions may read or modify the world state within the current context. If execution within a particular context reverts, staged world state modifications are rejected by the caller. If execution succeeds, staged world state modifications are accepted.

AvmWorldState

The following table defines an AVM context's world state interface:

| Field | AVM Instructions & Access |
|----------------|---|
| contracts | <code>*CALL</code> (special case, see below*) |
| publicStorage | <code>SLOAD</code> (membership-checks (latest) & reads), <code>SSTORE</code> (writes) |
| noteHashes | <code>NOTEHASHEXISTS</code> (membership-checks (start-of-block)), <code>EMITNOTEHASH</code> (appends) |
| nullifiers | <code>NULLIFIERSEXISTS</code> membership-checks (latest), <code>EMITNULLIFIER</code> (appends) |
| l1ToL2Messages | <code>L1TOL2MSGEXISTS</code> (membership-checks (start-of-block)) |

* `*CALL` is short for `CALL` / `STATICCALL`.

* For the purpose of the AVM, the world state's `contracts` member is readable for [bytecode fetching](#), and it is effectively updated when a new contract class or instance is created (along with a nullifier for the contract class identifier or contract address).

World State Access Trace

The circuit implementation of the AVM does *not* prove that its world state accesses are valid and properly sequenced, and does not perform actual tree updates. Thus, *all* world state accesses, regardless of whether they are rejected due to a revert, must be traced and eventually handed off to downstream circuits (public kernel and rollup circuits) for comprehensive validation and tree updates.

This trace of an AVM session's contract calls and world state accesses is named the **world state access trace**.

The world state access trace is also important for enforcing limitations on the maximum number of allowable world state accesses.

WorldStateAccessTrace

Each entry in the world state access trace is listed below along with its type and the instructions that append to it:

| Field | Relevant State | Type | Instructions |
|---------------------|-------------------|----------------------------------|---------------------------------------|
| accessCounter | all state | field | incremented by all instructions below |
| contractCalls | Contracts | Vector<TracedContractCall> | *CALL |
| publicStorageReads | Public Storage | Vector<TracedStorageRead> | SLOAD |
| publicStorageWrites | Public Storage | Vector<TracedStorageWrite> | SSTORE |
| noteHashChecks | Note Hashes | Vector<TracedNoteHashCheck> | NOTEHASHEXISTS |
| noteHashes | Note Hashes | Vector<TracedNoteHash> | EMITNOTEHASH |
| nullifierChecks | Nullifiers | Vector<TracedNullifierCheck> | NULIFIERSEXISTS |
| nullifiers | Nullifiers | Vector<TracedNullifier> | EMITNULLIFIER |
| l1ToL2MessageChecks | L1-To-L2 Messages | Vector<TracedL1ToL2MessageCheck> | L1TOL2MSGEXISTS |

The types tracked in these trace vectors are defined [here](#).

*CALL is short for CALL / STATICCALL.

Aztec tree operations like membership checks, appends, or leaf updates are performed in-circuit by downstream circuits (public kernel and rollup circuits), *after* AVM execution. The world state access trace is a list of requests made by the AVM for later circuits to perform.

Accrued Substate

The term "accrued substate" is borrowed from the [Ethereum Yellow Paper](#).

Accrued substate is accrued throughout a context's execution, but updates to it are strictly never relevant to subsequent instructions, contract calls, or transactions. An execution context is always initialized with empty accrued substate. Its vectors are append-only, and the instructions listed below append to these vectors. If a contract call's execution succeeds, its accrued substate is appended to the caller's. If a contract's execution reverts, its accrued substate is ignored.

AccruedSubstate

| Field | Type | Instructions |
|--------------------|---------------------------|------------------------------------|
| unencryptedLogs | Vector<UnencryptedLog> | EMITUNENCRYPTEDLOG |
| sentL2ToL1Messages | Vector<SentL2ToL1Message> | SENDL1TOL2MSG |

The types tracked in these vectors are defined [here](#).

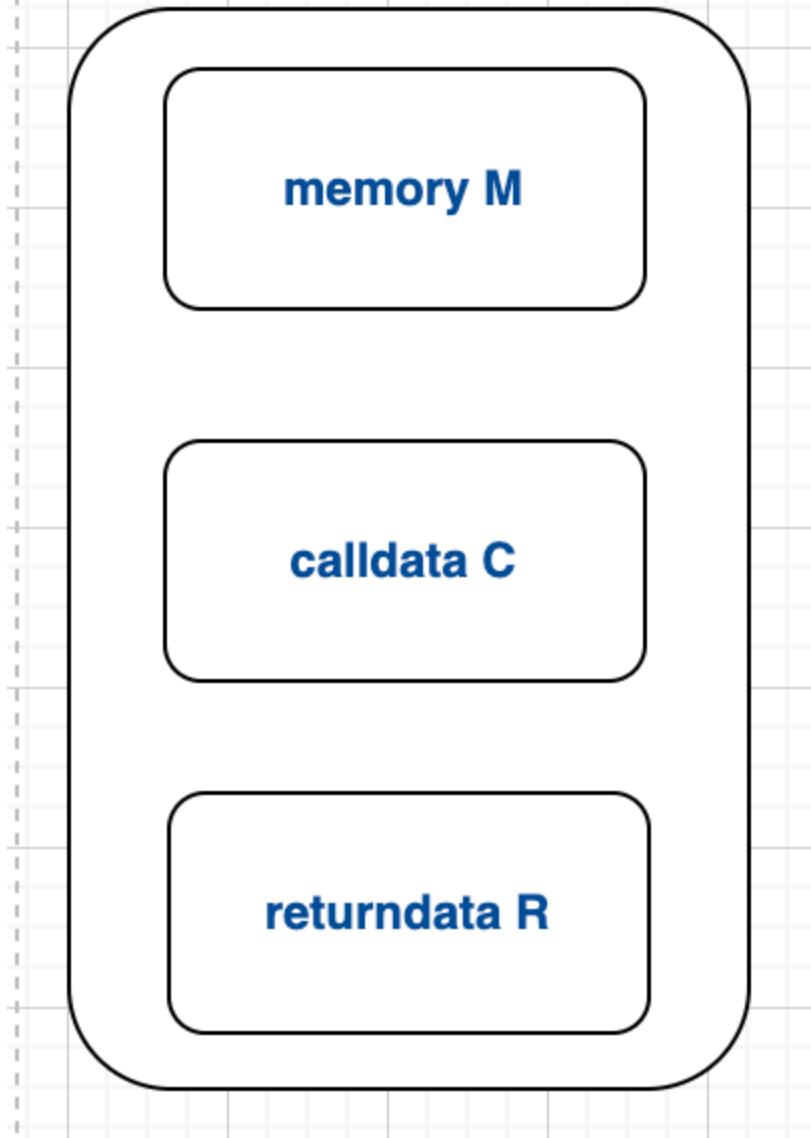
Memory Model

This section describes the AVM memory model, and in particular specifies "internal" VM abstractions that can be mapped to the VM's circuit architecture.

A memory-only state model

The AVM possesses three distinct data regions, accessed via distinct VM instructions: memory, calldata and returndata

Memory pages



All data regions are linear blocks of memory where each memory cell stores a finite field element.

Main Memory

Main memory stores the internal state of the current program being executed. Can be

written to as well as read.

The main memory region stores *type tags* alongside data values.

Calldata

Read-only data structure that stores the input data when executing a public function.

Returndata

When a function is called from within the public VM, the return parameters of the called function are present in returndata.

Registers (and their absence in the AVM)

The AVM does not have external registers. i.e. a register that holds a persistent value that can be operated on from one opcode to the next.

For example, in the x86 architecture, there exist 8 registers (%rax, %rbx etc).

Instructions can operate either directly on register values (e.g. `add %rax %rbx`) or on values in memory that the register values point to (e.g. `add (%rax) (%rbx)`).

The AVM does not support registers as this would require each register to exist as a column in the VM execution trace. "registers" can be implemented as a higher-level abstraction by a compiler producing AVM bytecode, by reserving fixed regions of memory to represent registers.

Memory addressing mode

In the AVM, an instruction operand `X` can refer to one of three quantities:

1. A literal value `X`
2. A memory address `M[X]`

3. An indirect memory address $M[M[X]]$

Indirect memory addressing is required in order to support read/writes into dynamically-sized data structures (the address parameter X is part of the program bytecode, which is insufficient to describe the location in memory of a dynamically-sized data structure).

Memory addresses must be tagged to be a $u32$ type.

Types and Tagged Memory

Terminology/legend

- $M[X]$: main memory cell at offset X
- tag : a value referring to a memory cell's type (its maximum potential value)
- $T[X]$: the tag associated with memory cell at offset X
- $inTag$: an instruction's tag to check input operands against. Present for many but not all instructions.
- $dstTag$: the target type of a $CAST$ instruction, also used to tag the destination memory cell
- $ADD<X>$: shorthand for an ADD instruction with $inTag = X$
- $ADD<X> aOffset bOffset dstOffset$: a full ADD instruction with $inTag = X$. See [here](#) for more details.
- $CAST<X>$: a $CAST$ instruction with $dstTag: X$. $CAST$ is the only instruction with a $dstTag$. See [here](#) for more details.

Tags and tagged memory

A tag refers to the maximum potential value of a cell of main memory. The following

tags are supported:

| tag value | maximum memory cell value | shorthand |
|-----------|---------------------------|---------------|
| 0 | 0 | uninitialized |
| 1 | $2^8 - 1$ | u8 |
| 2 | $2^{16} - 1$ | u16 |
| 3 | $2^{32} - 1$ | u32 |
| 4 | $2^{64} - 1$ | u64 |
| 5 | $2^{128} - 1$ | u128 |
| 6 | $p - 1$ | field |
| 7 | reserved | reserved |

Note: p describes the modulus of the finite field that the AVM circuit is defined over (i.e. number of points on the BN254 curve). Note: u32 is used for offsets into the VM's 32-bit addressable main memory

The purpose of a tag is to inform the VM of the maximum possible length of an operand value that has been loaded from memory.

Checking input operand tags

Many AVM instructions explicitly operate over range-constrained input parameters (e.g. ADD<inTag>). The maximum allowable value for an instruction's input

parameters is defined via an `inTag` (instruction/input tag). Two potential scenarios result:

1. A VM instruction's tag value matches the input parameter tag values
2. A VM instruction's tag value does *not* match the input parameter tag values

If case 2 is triggered, an error flag is raised and the current call's execution reverts.

Writing into memory

It is required that all VM instructions that write into main memory explicitly define the tag of the destination value and ensure the value is appropriately constrained to be consistent with the assigned tag. You can see an instruction's "Tag updates" in its section of the instruction set document (see [here for ADD](#) and [here for CAST](#)).

Standard tagging example: ADD

```
# ADD<u32> aOffset bOffset dstOffset
assert T[aOffset] == T[bOffset] == u32 // check inputs against
inTag, revert on mismatch
T[dstOffset] = u32                  // tag destination with
inTag
M[dstOffset] = M[aOffset] + M[bOffset] // perform the addition
```

MOV and tag preservation

The `MOV` instruction copies data from one memory cell to another, preserving tags. In other words, the destination cell's tag will adopt the value of the source:

```
# MOV srcOffset dstOffset
T[dstOffset] = T[srcOffset] // preserve tag
M[dstOffset] = M[srcOffset] // perform the move
```

Note that `MOV` does not have an `inTag` and therefore does not need to make any assertions regarding the source memory cell's type.

`CAST` and tag conversions

The only VM instruction that can be used to cast between tags is `CAST`. Two potential scenarios result:

1. The destination tag describes a maximum value that is *less than* the source tag
2. The destination tag describes a maximum value that is *greater than or equal to* the source tag

For Case 1, range constraints must be applied to ensure the destination value is consistent with the source value after tag truncations have been applied.

Case 2 is trivial as no additional consistency checks must be performed between source and destination values.

```
# CAST<u64> srcOffset dstOffset
T[dstOffset] = u64                                // tag destination
with dstTag
M[dstOffset] = cast<to: u64>(M[srcOffset]) // perform cast
```

Indirect `MOV` and extra tag checks

A `MOV` instruction may flag its source and/or destination offsets as "indirect". An indirect memory access performs `M[M[offset]]` instead of the standard `M[offset]`. Memory offsets must be `u32`s since main memory is a 32-bit addressable space, and so indirect memory accesses include additional checks.

Additional checks for a `MOV` with an indirect source offset:

```

# MOV srcOffset dstOffset      // with indirect source
assert T[srcOffset] == u32    // enforce that `M[srcOffset]` is
itself a valid memory offset
T[dstOffset] = T[T[srcOffset]] // tag destination to match
indirect source tag
M[dstOffset] = M[M[srcOffset]] // perform move from indirect
source

```

Additional checks for a `MOV` with an indirect destination offset:

```

# MOV srcOffset dstOffset      // with indirect destination
assert T[dstOffset] == u32    // enforce that `M[dstOffset]` is
itself a valid memory offset
T[T[dstOffset]] = T[srcOffset] // tag indirect destination to
match source tag
M[M[dstOffset]] = M[srcOffset] // perform move to indirect
destination

```

Additional checks for a `MOV` with both indirect source and destination offsets:

```

# MOV srcOffset dstOffset          // with indirect
source and destination
assert T[srcOffset] == T[dstOffset] == u32 // enforce that
`M[*Offset]` are valid memory offsets
T[T[dstOffset]] = T[T[srcOffset]]        // tag indirect
destination to match indirect source tag
M[M[dstOffset]] = M[M[srcOffset]]       // perform move to
indirect destination

```

Calldata/returndata and tag conversions

All elements in calldata/returndata are implicitly tagged as field elements (i.e. maximum value is $p - 1$). To perform a tag conversion, calldata/returndata must be

copied into main memory (via `CALCDATACOPY` or `RETURN`'s `offset` and `size`), followed by an appropriate `CAST` instruction.

```
# Copy calldata to memory and cast a word to u64
CALLDATACOPY cdOffset size offsetA // copy calldata to memory at
offsetA
CAST<u64> offsetA dstOffset           // cast first copied word to a
u64
```

This would perform the following:

Execution Context

ⓘ REMINDER

Many terms and definitions here are borrowed from the [Ethereum Yellow Paper](#).

An **execution context** contains the information and state relevant to a contract call's execution. When a contract call is made, an execution context is **initialized** before the contract code's execution begins.

AvmContext

| Field | Type |
|-----------------------|-----------------------|
| environment | ExecutionEnvironment |
| machineState | MachineState |
| worldState | AvmWorldState |
| worldStateAccessTrace | WorldStateAccessTrace |
| accruedSubstate | AccruedSubstate |
| results | ContractCallResults |

Execution Environment

A context's execution environment remains constant throughout a contract call's execution. When a contract call initializes its execution context, it [fully specifies the execution environment](#).

ExecutionEnvironment

| Field | Type | Description |
|---------------------|--------------|--|
| address | AztecAddress | |
| sender | AztecAddress | |
| functionSelector | u32 | |
| transactionFee | field | Computed transaction fee based on gas fees, inclusion fee, and gas usage. Zero in all phases but teardown. |
| contractCallDepth | field | Depth of the current call (how many nested calls deep is it). |
| contractCallPointer | field | Uniquely identifies each contract call processed by an AVM session. An initial call is assigned pointer value of 1 |

| Field | Type | Description |
|--------------|----------------------------|---|
| | | (expanded on in the AVM circuit section's "Call Pointer" subsection). |
| globals | PublicGlobalVariables | |
| isStaticCall | boolean | |
| calldata | [field; <calldata-length>] | |

Contract Call Results

When a contract call halts, it sets the context's **contract call results** to communicate results to the caller.

ContractCallResults

| Field | Type | Description |
|----------|--------------------------|-------------|
| reverted | boolean | |
| output | [field; <output-length>] | |

Context initialization

Initial contract calls

An **initial contract call** initializes a new execution context from a public execution request.

```
context = AvmContext {  
    environment = INITIAL_EXECUTION_ENVIRONMENT,  
    machineState = INITIAL_MACHINE_STATE,  
    worldState = <latest world state>,  
    worldStateAccessTrace = INITIAL_WORLD_STATE_ACCESS_TRACE,  
    accruedSubstate = { [], ... [], }, // all substate vectors  
    empty  
    results = INITIAL_CONTRACT_CALL_RESULTS,  
}
```

Since world state persists between transactions, the latest state is injected into a new AVM context.

Given a [PublicCallRequest](#) and its parent [TxRequest](#), these above-listed "INITIAL_*" entries are defined as follows:

```
INITIAL_EXECUTION_ENVIRONMENT = ExecutionEnvironment {  
    address: PublicCallRequest.contractAddress,  
    sender: PublicCallRequest.CallContext.msgSender,  
    functionSelector: PublicCallRequest.functionSelector,  
    contractCallDepth: 0,  
    contractCallPointer: 1,  
    globals: <current block's global variables>  
    isStaticCall: PublicCallRequest.CallContext.isStaticCall,
```

Nested contract calls

See the dedicated "[Nested Contract Calls](#)" page for a detailed explanation of nested contract calls.

The nested call's execution context is derived from the caller's context and the call instruction's arguments.

The following shorthand syntax is used to refer to nested context derivation in the "[Instruction Set](#)" and other sections:

```
// instr.args are { gasOffset, addrOffset, argsOffset, retOffset,
retSize }

isStaticCall = instr.opcode == STATICCALL

nestedContext = deriveContext(context, instr.args, isStaticCall)
```

Nested context derivation is defined as follows:

```
nestedExecutionEnvironment = ExecutionEnvironment {
    address: M[addrOffset],
    sender: context.address,
    functionSelector: context.environment.functionSelector,
    transactionFee: context.environment.transactionFee,
    contractCallDepth: context.contractCallDepth + 1,
    contractCallPointer:
    context.worldStateAccessTrace.contractCalls.length + 1,
    globals: context.globals,
    isStaticCall: isStaticCall,
    calldata:
    context.memory[M[argsOffset]:M[argsOffset]+argsSize],
}
```

```
nestedContext = AvmContext {  
    environment: nestedExecutionEnvironment,  
    machineState: nestedMachineState,  
    worldState: context.worldState,  
    worldStateAccessTrace: context.worldStateAccessTrace,  
    accruedSubstate: { [], ... [], }, // all empty  
    results: {reverted: false, output: []},  
}
```

M[offset] notation is shorthand for context.machineState.memory[offset]

Execution, Gas, Halting

Execution of an AVM program, within a provided [execution context](#), includes the following steps:

1. Fetch contract bytecode and decode into a vector of [AVM instructions](#)
2. Repeat the next step until a [halt](#) is reached
3. Execute the instruction at the index specified by the context's [program counter](#)
 - Instruction execution will update the program counter

The following shorthand syntax is used to refer to this execution routine in the "[Instruction Set](#)", "[Nested execution](#)", and other sections:

```
execute(context);
```

Bytecode fetch and decode

Before execution begins, a contract's bytecode is retrieved.

```
bytecode = context.worldState.contracts[context.environment.address].bytecode;
```

As described in "[Contract Deployment](#)", contracts are not stored in a dedicated tree. A [contract class](#) is represented as an unencrypted log containing the [ContractClass](#) structure (which contains the bytecode) and a nullifier representing the class identifier. A [contract instance](#) is represented as an unencrypted log containing the [ContractInstance](#) structure and a nullifier representing the contract address.

Thus, the syntax used above for bytecode retrieval is shorthand for:

1. Perform a membership check of the contract instance address nullifier
2. Retrieve the [ContractInstance](#) from a database that tracks all such unencrypted logs

```
contractInstance = contractInstances[context.environment.address];
```

3. Perform a membership check of the contract class identifier nullifier
4. Retrieve the [ContractClass](#) and its bytecode from a database that tracks all such unencrypted logs

```
contractClass = contractClasses[contractInstance.contract_class_id];
bytecode = contractClass.packed_public_bytecode;
```

The bytecode is then decoded into a vector of `instructions`. An instruction is referenced throughout this document according to the following interface:

| Member | Description |
|-----------------------|--|
| <code>opcode</code> | The 8-bit opcode value that identifies the operation an instruction is meant to perform. |
| <code>indirect</code> | Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like <code>M[M[offset]]</code> instead of the more standard <code>M[offset]</code> . |
| <code>inTag</code> | The <code>tag/size</code> to check inputs against and/or tag the destination with. |
| <code>args</code> | Named arguments as specified for an instruction in the " Instruction Set ". As an example, <code>instr.args.a0ffset</code> refers to an instructions argument named <code>a0ffset</code> . |
| <code>execute</code> | Apply this instruction's transition function to an execution context (e.g. <code>instr.execute(context)</code>). |

Instruction execution

Once bytecode has been fetched and decoded into the `instructions` vector, instruction execution begins.

The AVM executes the instruction at the index specified by the context's program counter.

```
while (!halted) instr = instructions[machineState.pc];
instr.execute(context);
```

An instruction's execution mutates the context's state as specified in the "[Instruction Set](#)".

Program Counter and Control Flow

A context is initialized with a program counter of zero, and so instruction execution always begins with a contract's very first instruction.

The program counter specifies which instruction the AVM will execute next, and each instruction's execution updates the program counter in some way. This allows the AVM to progress to the next instruction at each step.

Most instructions simply increment the program counter by 1. This allows VM execution to flow naturally from instruction to instruction. Some instructions (`JUMP`, `JUMPI`, `INTERNALCALL`) modify the program counter based on arguments.

The `INTERNALCALL` instruction pushes `machineState.pc+1` to `machineState.internalCallStack` and then updates `pc` to the instruction's destination argument (`instr.args.loc`). The `INTERNALRETURN` instruction pops a destination from `machineState.internalCallStack` and assigns the result to `pc`.

An instruction will never assign program counter a value from memory (`machineState.memory`). A `JUMP`, `JUMPI`, or `INTERNALCALL` instruction's destination is a constant from the program bytecode. This property allows for easier static program analysis.

Gas checks and tracking

See "[Gas and Fees](#)" for a deeper dive into Aztec's gas model and for definitions of each type of gas.

Each instruction has an associated `l2GasCost` and `daGasCost`. The AVM uses these values to enforce that sufficient gas is available before executing an instruction, and to deduct the cost from the context's remaining gas. The process of checking and charging gas is referred to in other sections using the following shorthand:

```
chargeGas(context, l2GasCost, daGasCost);
```

Checking gas

Before an instruction is executed, the VM enforces that there is sufficient gas remaining via the following assertions:

```
assert machineState.l2GasLeft - instr.l2GasCost >= 0  
assert machineState.daGasLeft - instr.daGasCost >= 0
```

Many instructions (like arithmetic operations) have 0 `daGasCost`. Instructions only incur a DA cost if they modify the [world state](#) or [accrued substate](#).

Charging gas

If these assertions pass, the machine state's gas left is decreased prior to the instruction's core execution:

```
machineState.l2GasLeft -= instr.l2GasCost  
machineState.daGasLeft -= instr.daGasCost
```

If either of these assertions *fail* for an instruction, this triggers an exceptional halt. The gas left is set to 0 and execution reverts.

```
machineState.l2GasLeft = 0  
machineState.daGasLeft = 0
```

Reverting and exceptional halts are covered in more detail in the ["Halting" section](#).

Gas cost notes and examples

An instruction's gas cost is meant to reflect the computational cost of generating a proof of its correct execution. For some instructions, this computational cost changes based on inputs. Here are some examples and important notes:

- All instructions have a base cost. `JUMP` is an example of an instruction with constant gas cost. Regardless of its inputs, the instruction always incurs the same `l2GasCost` and `daGasCost`.
- The `SET` instruction operates on a different sized constant (based on its `dstTag`). Therefore, this

instruction's gas cost increases with the size of its input.

- In addition to the base cost, the cost of an instruction increases with the number of reads and writes to memory. This is affected by the total number of input and outputs: the gas cost for `AND` should be greater than that of `NOT` since it takes one more input.
- Input parameters flagged as "indirect" require an extra memory access, so these should further increase the gas cost of the instruction.
- The base cost for instructions that operate on a data range of a specified "size" scale in cost with that size, but only if they perform an operation on the data other than copying. For example, `CALLDATACOPY` copies `copySize` words from `environment.calldata` to `machineState.memory`, so its increased cost is captured by the extra memory accesses. On the other hand, `SSTORE` requires accesses to persistent storage proportional to `srcSize`, so its base cost should also increase.
- The `CALL`/`STATICCALL` instruction's gas cost is determined by its `*Gas` arguments, but any gas unused by the nested contract call's execution is refunded after its completion ([more on this later](#)).

An instruction's gas cost will roughly align with the number of rows it corresponds to in the SNARK execution trace including rows in the sub-operation table, memory table, chiplet tables, etc.

An instruction's gas cost takes into account the costs of associated downstream computations. An instruction that triggers accesses to the public data tree (`SLOAD`/`SSTORE`) incurs a cost that accounts for state access validation in later circuits (public kernel or rollup). A contract call instruction (`CALL`/`STATICCALL`) incurs a cost accounting for the nested call's complete execution as well as any work required by the public kernel circuit for this additional call.

Halting

A context's execution can end with a **normal halt** or **exceptional halt**. A halt ends execution within the current context and returns control flow to the calling context.

Normal halting

A normal halt occurs when the VM encounters an explicit halting instruction (`RETURN` or `REVERT`). Such instructions consume gas normally and optionally initialize some output data before finally halting the current context's execution.

```
machineState.l2GasLeft -= instr.l2GasCost
```

Definitions: `retOffset` and `retSize` here are arguments to the `RETURN` and `REVERT` instructions. If `retSize` is 0, the context will have no output. Otherwise, these arguments point to a region of memory to output.

`results.output` is only relevant when the caller is a contract call itself. In other words, it is only relevant for [nested contract calls](#). When an [initial contract call](#) (initiated by a public execution request) halts normally, its `results.output` is ignored.

Exceptional halting

An exceptional halt is not explicitly triggered by an instruction but instead occurs when an exceptional condition is met.

When an exceptional halt occurs, the context is flagged as consuming all of its allocated gas and is marked as `reverted` with *no output data*, and then execution within the current context ends.

```
machineState.l2GasLeft = 0  
machineState.daGasLeft = 0  
results.reverted = true  
// results.output remains empty
```

The AVM's exceptional halting conditions area listed below:

1. Insufficient gas

```
assert machineState.l2GasLeft - instr.l2GasCost >= 0  
assert machineState.daGasLeft - instr.l2GasCost >= 0
```

2. Invalid instruction encountered

```
assert instructions[machineState.pc].opcode <= MAX_AVM_OPCODE
```

3. Jump destination past end of program

```
assert instructions[machineState.pc].opcode not in {JUMP, JUMPI,  
INTERNALCALL}  
OR instr.args.loc < instructions.length
```

4. Failed memory tag check

- Defined per-instruction in the [Instruction Set](#)

5. Out of bounds memory access (max memory offset is $2^{32} - 1$)

```
for offset in instr.args.*Offset:  
    assert offset < 2^32
```

6. World state modification attempt during a static call

```
assert !environment.isStaticCall  
    OR instructions[machineState.pc].opcode not in WS_AS MODIFYING_OPS
```

Definition: `WS_AS MODIFYING_OPS` represents the list of all opcodes corresponding to instructions that modify world state or accrued substate.

7. Maximum contract call depth (1024) exceeded

```
assert environment.contractCallDepth <= 1024  
assert instructions[machineState.pc].opcode not in {CALL, STATICCALL}  
    OR environment.contractCallDepth < 1024
```

8. Maximum contract call calls per execution request (1024) exceeded

```
assert worldStateAccessTrace.contractCalls.length <= 1024  
assert instructions[machineState.pc].opcode not in {CALL, STATICCALL}  
    OR worldStateAccessTrace.contractCalls.length < 1024
```

9. Maximum internal call depth (1024) exceeded

```
assert machineState.internalCallStack.length <= 1024  
assert instructions[machineState.pc].opcode != INTERNALCALL  
    OR environment.contractCallDepth < 1024
```

10. Maximum world state accesses (1024-per-category) exceeded

```

assert worldStateAccessTrace.publicStorageReads.length <= 1024
    AND worldStateAccessTrace.publicStorageWrites.length <= 1024
    AND worldStateAccessTrace.noteHashChecks.length <= 1024
    AND worldStateAccessTrace.noteHashes.length <= 1024
    AND worldStateAccessTrace.nullifierChecks.length <= 1024
    AND worldStateAccessTrace.nullifiers.length <= 1024
    AND worldStateAccessTrace.l1ToL2MessageChecks.length <= 1024
    AND worldStateAccessTrace.archiveChecks.length <= 1024

// Storage
assert instructions[machineState.pc].opcode != SLOAD
    OR worldStateAccessTrace.publicStorageReads.length < 1024
assert instructions[machineState.pc].opcode != SSTORE
    OR worldStateAccessTrace.publicStorageWrites.length < 1024

// Note hashes
assert instructions[machineState.pc].opcode != NOTEHASHEXISTS
    OR noteHashChecks.length < 1024
assert instructions[machineState.pc].opcode != EMITNOTEHASH
    OR noteHashes.length < 1024

// Nullifiers
assert instructions[machineState.pc].opcode != NULLIFIEREXISTS
    OR nullifierChecks.length < 1024
assert instructions[machineState.pc].opcode != EMITNULLIFIER
    OR nullifiers.length < 1024

// Read L1 to L2 messages
assert instructions[machineState.pc].opcode != L1TOL2MSGEXISTS
    OR worldStateAccessTrace.l1ToL2MessagesChecks.length < 1024

```

11. Maximum accrued substate entries (per-category) exceeded

```

assert accruedSubstate.unencryptedLogs.length <= MAX_UNENCRYPTED_LOGS
    AND accruedSubstate.sentL2ToL1Messages.length <=
MAX_SENT_L2_TO_L1_MESSAGES

// Unencrypted logs
assert instructions[machineState.pc].opcode != EMITUNENCRYPTEDLOG
    OR unencryptedLogs.length < MAX_UNENCRYPTED_LOGS

// Sent L2 to L1 messages
assert instructions[machineState.pc].opcode != SENDL2TOL1MSG

```

Note that ideally the AVM should limit the *total* accrued substate entries per-category instead of the entries per-call.

Nested Contract Calls

A nested contract call occurs *during* AVM execution and is triggered by a **contract call instruction**. The AVM **instruction set** includes three contract call instructions: **CALL**, and **STATICCALL**.

A nested contract call performs the following operations:

1. Charge gas for the nested call
2. Trace the nested contract call
3. Derive the nested context from the calling context and the call instruction
4. Initiate AVM execution within the nested context until a halt is reached
5. Update the calling context after the nested call halts

Or, in pseudocode:

```
// instr.args are { gasOffset, addrOffset, argsOffset, retOffset, retSize }

isStaticCall = instr.opcode == STATICCALL;
l2GasCost = min(M[instr.args.gasOffset],
context.machineState.l2GasLeft);
daGasCost = min(M[instr.args.gasOffset + 1],
context.machineState.daGasLeft);

chargeGas(context, l2GasCost, daGasCost);
traceNestedCall(context, instr.args.addrOffset);
nestedContext = deriveContext(context, instr.args, isStaticCall);
execute(nestedContext);
updateContextAfterNestedCall(context, instr.args, nestedContext);
```

These call instructions share the same argument definitions: **gasOffset**,

`addrOffset`, `argsOffset`, `argsSize`, `retOffset`, `retSize`, and `successOffset` (defined in the [instruction set](#)). These arguments will be referred to via those keywords below, and will often be used in conjunction with the `M[offset]` syntax which is shorthand for `context.machineState.memory[offset]`.

Tracing nested contract calls

Before nested execution begins, the contract call is traced.

```
traceNestedCall(context, addrOffset)
// which is shorthand for
context.worldStateAccessTrace.contractCalls.append(
    TracedContractCall {
        callPointer:
        context.worldStateAccessTrace.contractCalls.length + 1,
        address: M[addrOffset],
        counter: ++context.worldStateAccessTrace.accessCounter,
        endLifetime: 0, // The call's end-lifetime will be
        updated later if it or its caller reverts
    }
)
```

Context initialization for nested calls

The nested call's execution context is derived from the caller's context and the call instruction's arguments.

The following shorthand syntax is used to refer to nested context derivation in the ["Instruction Set"](#) and other sections:

```

// instr.args are { gasOffset, addrOffset, argsOffset, retOffset,
retSize }

isStaticCall = instr.opcode == STATICCALL

nestedContext = deriveContext(context, instr.args, isStaticCall)

```

Nested context derivation is defined as follows:

```

nestedExecutionEnvironment = ExecutionEnvironment {
    address: M[addrOffset],
    sender: context.address,
    functionSelector: context.environment.functionSelector,
    transactionFee: context.environment.transactionFee,
    contractCallDepth: context.contractCallDepth + 1,
    contractCallPointer:
        context.worldStateAccessTrace.contractCalls.length + 1,
    globals: context.globals,
    isStaticCall: isStaticCall,
    calldata:
        context.memory[M[argsOffset]:M[argsOffset]+argsSize],
}

nestedMachineState = MachineState {
    l2GasLeft: context.machineState.memory[M[gasOffset]],
    daGasLeft: context.machineState.memory[M[gasOffset+1]],
    pc = 0,
    internalCallStack = [], // initialized as empty
    memory = [0, ..., 0],   // all 2^32 entries are initialized
    to zero
}

```

```

nestedContext = AvmContext {
    environment: nestedExecutionEnvironment,

```

`M[offset]` notation is shorthand for `context.machineState.memory[offset]`

Gas cost of call instruction

A call instruction's gas cost is derived from its `gasOffset` argument. In other words, the caller "allocates" gas for a nested call via its `gasOffset` argument.

As with all instructions, gas is checked and cost is deducted *prior* to the instruction's execution.

```
chargeGas(context, 12GasCost, daGasCost);
```

The shorthand `chargeGas` is defined in "[Gas checks and tracking](#)".

As with all instructions, gas is checked and cost is deducted *prior* to the instruction's execution.

```
assert context.machineState.l2GasLeft - 12GasCost >= 0
assert context.machineState.daGasLeft - daGasCost >= 0
context.l2GasLeft -= 12GasCost
context.daGasLeft -= daGasCost
```

When the nested call halts, it may not have used up its entire gas allocation. Any unused gas is refunded to the caller as expanded on in "[Updating the calling context after nested call halts](#)".

Nested execution

Once the nested call's context is initialized, execution within that context begins.

```
execute(nestedContext);
```

Execution (and the `execution` shorthand above) is detailed in "[Execution, Gas, Halting](#)". Note that execution mutates the nested context.

Updating the calling context after nested call halts

After the nested call halts, the calling context is updated. The call's success is extracted, unused gas is refunded, output data can be copied to the caller's memory, world state and accrued substate are conditionally accepted, and the world state trace is updated. The following shorthand is used to refer to this process in the "[Instruction Set](#)":

```
updateContextAfterNestedCall(context, instr.args, nestedContext);
```

The caller checks whether the nested call succeeded, and places the answer in memory.

```
context.machineState.memory[instr.args.successOffset] =  
!nestedContext.results.reverted;
```

Any unused gas is refunded to the caller.

```
context.l2GasLeft += nestedContext.machineState.l2GasLeft;  
context.daGasLeft += nestedContext.machineState.daGasLeft;
```

If the call instruction specifies non-zero `retSize`, the caller copies any returned

output data to its memory.

```
if retSize > 0:  
    context.machineState.memory[retOffset:retOffset+retSize] =  
nestedContext.results.output
```

If the nested call succeeded, the caller accepts its world state and accrued substate modifications.

```
if !nestedContext.results.reverted:  
    context.worldState = nestedContext.worldState  
    context.accruedSubstate.append(nestedContext.accruedSubstate)
```

Accepting nested call's World State access trace

If the nested call reverted, the caller initializes the "end-lifetime" of all world state accesses made within the nested call.

```
if nestedContext.results.reverted:  
    // process all traces (this is shorthand)  
    for trace in nestedContext.worldStateAccessTrace:  
        for access in trace:  
            if access.callPointer >=  
nestedContext.environment.callPointer:  
                // don't override end-lifetime already set by a  
                // deeper nested call  
                if access.endLifetime == 0:  
                    access.endLifetime =  
nestedContext.worldStateAccessTrace.accessCounter
```

A world state access that was made in a deeper nested *reverted* context will already have its end-lifetime initialized. The caller does *not* overwrite this access' end-lifetime.

end-lifetime here as it already has a narrower lifetime.

Regardless of whether the nested call reverted, the caller accepts its updated world state access trace (with updated end-lifetimes).

```
context.worldStateAccessTrace =  
nestedContext.worldStateAccessTrace;
```

Instruction Set

This page lists all of the instructions supported by the Aztec Virtual Machine (AVM).

The following notes are relevant to the table and sections below:

- `M[offset]` notation is shorthand for `context.machineState.memory[offset]`
- `S[slot]` notation is shorthand for an access to the specified slot in the current contract's public storage (`context.worldState.publicStorage`) after the slot has been siloed by the contract address (`hash(context.environment.address, slot)`)
- Any instruction whose description does not mention a program counter change simply increments it: `context.machineState.pc++`
- All instructions update `context.machineState.*GasLeft` as detailed in "Gas limits and tracking"
- Any instruction can lead to an exceptional halt as specified in "Exceptional halting"
- The term `hash` used in expressions below represents a Poseidon hash operation.
- Type structures used in world state tracing operations are defined in "Type Definitions"

Instructions Table

Click on an instruction name to jump to its section.

| Opcode | Name | Summary | Expression |
|--------|----------------------|-----------------------------------|---|
| 0x00 | ADD | Addition (a + b) | $M[\text{dstOffset}] = M[\text{aOffset}] + M[\text{bOffset}] \bmod 2^k$ |
| 0x01 | SUB | Subtraction (a - b) | $M[\text{dstOffset}] = M[\text{aOffset}] - M[\text{bOffset}] \bmod 2^k$ |
| 0x02 | MUL | Multiplication (a * b) | $M[\text{dstOffset}] = M[\text{aOffset}] * M[\text{bOffset}] \bmod 2^k$ |
| 0x03 | DIV | Unsigned integer division (a / b) | $M[\text{dstOffset}] = M[\text{aOffset}] / M[\text{bOffset}]$ |
| 0x04 | FDIV | Field division (a / b) | $M[\text{dstOffset}] = M[\text{aOffset}] / M[\text{bOffset}]$ |
| 0x05 | EQ | Equality check (a == b) | $M[\text{dstOffset}] = M[\text{aOffset}] == M[\text{bOffset}] ? 1 : 0$ |
| 0x06 | LT | Less-than check (a < b) | $M[\text{dstOffset}] = M[\text{aOffset}] < M[\text{bOffset}] ? 1 : 0$ |

| | | | |
|------|------------------|---|--|
| 0x07 | LTE | Less-than-or-equals check
(a <= b) | <code>M[dstOffset] = M[aOffset] <= M[bOffset] ? 1 : 0</code> |
| 0x08 | AND | Bitwise AND
(a & b) | <code>M[dstOffset] = M[aOffset] AND M[bOffset]</code> |
| 0x09 | OR | Bitwise OR (a b) | <code>M[dstOffset] = M[aOffset] OR M[bOffset]</code> |
| 0x0a | XOR | Bitwise XOR
(a ^ b) | <code>M[dstOffset] = M[aOffset] XOR M[bOffset]</code> |
| 0x0b | NOT | Bitwise NOT
(inversion) | <code>M[dstOffset] = NOT M[aOffset]</code> |
| 0x0c | SHL | Bitwise leftward shift
(a << b) | <code>M[dstOffset] = M[aOffset] << M[bOffset]</code> |
| 0x0d | SHR | Bitwise rightward shift (a >> b) | <code>M[dstOffset] = M[aOffset] >> M[bOffset]</code> |
| 0x0e | CAST | Type cast | <code>M[dstOffset] = cast<dstTag>(M[aOffset])</code> |
| 0x0f | ADDRESS | Get the address of the currently executing I2 contract | <code>M[dstOffset] = context.environment.address</code> |
| 0x10 | SENDER | Get the address of the sender (caller of the current context) | <code>M[dstOffset] = context.environment.sender</code> |
| 0x11 | FUNCTIONSELECTOR | Get the function selector of the contract function | <code>M[dstOffset] = context.environment.functionSelector</code> |

| | | | |
|------|-----------------------------|--|---|
| | | being executed | |
| 0x12 | <code>TRANSACTIONFEE</code> | Get the computed transaction fee during teardown phase, zero otherwise | <pre>M[dstOffset] = context.environment.transactionFee</pre> |
| 0x13 | <code>CHAINID</code> | Get this rollup's L1 chain ID | <pre>M[dstOffset] = context.environment.globals.chainId</pre> |
| 0x14 | <code>VERSION</code> | Get this rollup's L2 version ID | <pre>M[dstOffset] = context.environment.globals.version</pre> |
| 0x15 | <code>BLOCKNUMBER</code> | Get this L2 block's number | <pre>M[dstOffset] = context.environment.globals.blocknumber</pre> |
| 0x16 | <code>TIMESTAMP</code> | Get this L2 block's timestamp | <pre>M[dstOffset] = context.environment.globals.timestamp</pre> |
| 0x17 | <code>FEEPERL2GAS</code> | Get the fee to be paid per "L2 gas" - constant for entire transaction | <pre>M[dstOffset] = context.environment.globals.feePerL2Gas</pre> |
| 0x18 | <code>FEEPERDAGAS</code> | Get the fee to be paid per "DA gas" - constant for entire transaction | <pre>M[dstOffset] = context.environment.globals.feePerDaGas</pre> |
| 0x19 | <code>CALLDATACOPY</code> | Copy calldata into memory | <pre>M[dstOffset:dstOffset+copySize] = context.environment.calldata[cdOffset:cdOffset+copySize]</pre> |

| | | | |
|------|----------------|--|---|
| 0x1a | L2GASLEFT | Remaining "L2 gas" for this call (after this instruction) | <code>M[dstOffset] = context.MachineState.l2GasLeft</code> |
| 0x1b | DAGASLEFT | Remaining "DA gas" for this call (after this instruction) | <code>M[dstOffset] = context.machineState.daGasLeft</code> |
| 0x1c | JUMP | Jump to a location in the bytecode | <code>context.machineState.pc = loc</code> |
| 0x1d | JUMPI | Conditionally jump to a location in the bytecode | <code>context.machineState.pc = M[condOffset] > 0 ? loc : context.machineState.pc</code> |
| 0x1e | INTERNALCALL | Make an internal call. Push the current PC to the internal call stack and jump to the target location. | <code>context.machineState.internalCallStack.push(context.machineState.pc)
context.machineState.pc = loc</code> |
| 0x1f | INTERNALRETURN | Return from an internal call. Pop from the internal call stack and jump to the popped location. | <code>context.machineState.pc =
context.machineState.internalCallStack.pop()</code> |
| 0x20 | SET | Set a memory word from a constant in the bytecode | <code>M[dstOffset] = const</code> |

| | | | |
|------|-----------------------------|---|---|
| 0x21 | <code>MOV</code> | Move a word from source memory location to destination | $M[\text{dstOffset}] = M[\text{srcOffset}]$ |
| 0x22 | <code>CMOV</code> | | $M[\text{dstOffset}] = M[\text{condOffset}] > 0 ? M[\text{aOffset}] : M[\text{bOffset}]$ |
| 0x23 | <code>SLOAD</code> | Load a word from this contract's persistent public storage. Zero is loaded for unwritten slots. | $M[\text{dstOffset}] = S[M[\text{slotOffset}]]$ |
| 0x24 | <code>SSTORE</code> | Write a word to this contract's persistent public storage | $S[M[\text{slotOffset}]] = M[\text{srcOffset}]$ |
| 0x25 | <code>NOTEHASHEXISTS</code> | Check whether a note hash exists in the note hash tree (as of the start of the current block) | <pre>exists = context.worldState.noteHashes.<i>has</i>({ leafIndex: M[leafIndexOffset] leaf: hash(context.environment.address, M[noteHashOffset]), }) M[existsOffset] = exists</pre> |
| 0x26 | <code>EMITNOTEHASH</code> | Emit a new note hash to be inserted into the note hash tree | <pre>context.worldState.noteHashes.<i>append</i>(hash(context.environment.address, M[noteHashOffset]))</pre> |
| 0x27 | <code>NULIFIEREXISTS</code> | Check whether a nullifier exists in the nullifier tree (including | <pre>exists = pendingNullifiers.<i>has</i>(M[addressOffset], M[nullifierOffset]) context.worldState.nullifiers.<i>has</i>(hash(M[addressOffset], M[nullifierOffset])) M[existsOffset] = exists</pre> |

| | | | |
|------|----------------------------------|--|---|
| | | nullifiers from earlier in the current transaction or from earlier in the current block) | |
| 0x28 | <code>EMITNULLIFIER</code> | Emit a new nullifier to be inserted into the nullifier tree | <pre>context.worldState.nullifiers.append(hash(context.environment.address, M=nullifierOffset)))</pre> |
| 0x29 | <code>L1TOL2MSGEXISTS</code> | Check if a message exists in the L1-to-L2 message tree | <pre>exists = context.worldState.l1ToL2Messages.has({ leafIndex: M[msgLeafIndexOffset], leaf: M[msgHashOffset] }) M[existsOffset] = exists</pre> |
| 0x2a | <code>GETCONTRACTINSTANCE</code> | Copies contract instance data to memory | <pre>M[dstOffset:dstOffset+CONTRACT_INSTANCE_SIZE+1] = [instance_found_in_address, instance.salt ?? 0, instance.deployer ?? 0, instance.contractClassId ?? 0, instance.initializationHash ?? 0, instance.portalContractAddress ?? 0, instance.publicKeysHash ?? 0,]</pre> |
| 0x2b | <code>EMITUNENCRYPTEDLOG</code> | Emit an unencrypted log | <pre>context.accruedSubstate.unencryptedLogs.append(UnencryptedLog { address: context.environment.address, log: M[logOffset:logOffset+M[logSizeOffset]], })</pre> |
| 0x2c | <code>SENDL2TOL1MSG</code> | Send an L2-to-L1 message | <pre>context.accruedSubstate.sentL2ToL1Messages.append(SentL2ToL1Message { address: context.environment.address, recipient: M[recipientOffset], message: M[contentOffset] })</pre> |

| | | | |
|------|--|--|--|
| 0x2d | | Call into another contract | <pre>// instr.args are { gasOffset, addrOffset, argsOffset, retOffset, retSize } chargeGas(context, l2GasCost=M[instr.args.gasOffset], daGasCost=M[instr.args.gasOffset+1]) traceNestedCall(context, instr.args.addrOffset) nestedContext = deriveContext(context, instr.args, isStaticCall=false) execute(nestedContext) updateContextAfterNestedCall(context, instr.args, nestedContext)</pre> |
| 0x2e | | Call into another contract, disallowing World State and Accrued Substate modifications | <pre>// instr.args are { gasOffset, addrOffset, argsOffset, retOffset, retSize } chargeGas(context, l2GasCost=M[instr.args.gasOffset], daGasCost=M[instr.args.gasOffset+1]) traceNestedCall(context, instr.args.addrOffset) nestedContext = deriveContext(context, instr.args, isStaticCall=true) execute(nestedContext) updateContextAfterNestedCall(context, instr.args, nestedContext)</pre> |
| 0x2f | | Halt execution within this context (without revert), optionally returning some data | <pre>context.contractCallResults.output = M[retOffset:retOffset+retSize] halt</pre> |
| 0x30 | | | <pre>context.contractCallResults.output = M[retOffset:retOffset+retSize] context.contractCallResults.reverted = true halt</pre> |
| 0x31 | | Convert a word to an array of limbs in little-endian radix form | TBD: Storage of limbs and if T[dstOffset] is constrained to U8 |

Instructions

ADD

Addition ($a + b$)

See in table.

- **Opcode:** 0x00
- **Category:** Compute - Arithmetic
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[offset]]$ instead of the more standard $M[offset]$.
 - **inTag:** The **tag/size** to check inputs against and tag the destination with.
- **Args:**
 - **aOffset:** memory offset of the operation's left input
 - **bOffset:** memory offset of the operation's right input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** $M[dstOffset] = M[aOffset] + M[bOffset] \bmod 2^k$
- **Details:** Wraps on overflow
- **Tag checks:** $T[aOffset] == T[bOffset] == inTag$
- **Tag updates:** $T[dstOffset] = inTag$
- **Bit-size:** 128

| opcode | reserved | flags | aOffset | bOffset | dstOffset |
|--------|----------|----------------|---------|---------|-----------|
| ADD | XXX | indirect inTag | | | |

SUB

Subtraction ($a - b$)

See in table.

- **Opcode:** 0x01
- **Category:** Compute - Arithmetic
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[offset]]$ instead of the more standard $M[offset]$.
 - **inTag:** The **tag/size** to check inputs against and tag the destination with.
- **Args:**
 - **aOffset:** memory offset of the operation's left input
 - **bOffset:** memory offset of the operation's right input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** $M[dstOffset] = M[aOffset] - M[bOffset] \bmod 2^k$
- **Details:** Wraps on undeflow
- **Tag checks:** $T[aOffset] == T[bOffset] == inTag$
- **Tag updates:** $T[dstOffset] = inTag$
- **Bit-size:** 128

| opcode | reserved | flags | aOffset | bOffset | dstOffset |
|--------|----------|----------------|---------|---------|-----------|
| SUB | XXX | indirect inTag | | | |

MUL

Multiplication ($a * b$)

See in table.

- **Opcode:** 0x02
- **Category:** Compute - Arithmetic
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[offset]]$ instead of the more standard $M[offset]$.
 - **inTag:** The **tag/size** to check inputs against and tag the destination with.
- **Args:**
 - **aOffset:** memory offset of the operation's left input
 - **bOffset:** memory offset of the operation's right input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** $M[dstOffset] = M[aOffset] * M[bOffset] \bmod 2^k$
- **Details:** Wraps on overflow
- **Tag checks:** $T[aOffset] == T[bOffset] == inTag$
- **Tag updates:** $T[dstOffset] = inTag$
- **Bit-size:** 128

| opcode | reserved | flags | aOffset | bOffset | dstOffset |
|--------|----------|----------------|---------|---------|-----------|
| MUL | XXX | indirect inTag | | | |

DIV

Unsigned integer division (a / b)

See in table.

- **Opcode:** 0x03
- **Category:** Compute - Arithmetic
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[offset]]$ instead of the more standard $M[offset]$.
 - **inTag:** The **tag/size** to check inputs against and tag the destination with.
- **Args:**
 - **aOffset:** memory offset of the operation's left input
 - **bOffset:** memory offset of the operation's right input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** $M[dstOffset] = M[aOffset] / M[bOffset]$
- **Details:** If the input is a field, it will be interpreted as an integer
- **Tag checks:** $T[aOffset] == T[bOffset] == inTag$
- **Tag updates:** $T[dstOffset] = inTag$
- **Bit-size:** 128

| opcode | reserved | flags | aOffset | bOffset | dstOffset |
|--------|----------|----------------|---------|---------|-----------|
| DIV | XXX | indirect inTag | | | |

FDIV

Field division (a / b)

See in table.

- **Opcode:** 0x04
- **Category:** Compute - Arithmetic
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **aOffset:** memory offset of the operation's left input
 - **bOffset:** memory offset of the operation's right input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = M[aOffset] / M[bOffset]`
- **Tag checks:** `T[aOffset] == T[bOffset] == field`
- **Tag updates:** `T[dstOffset] = field`
- **Bit-size:** 120

EQ

Equality check ($a == b$)

[See in table.](#)

- **Opcode:** 0x05
- **Category:** Compute - Comparators
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
 - **inTag:** The [tag/size](#) to check inputs against and tag the destination with.
- **Args:**
 - **aOffset:** memory offset of the operation's left input
 - **bOffset:** memory offset of the operation's right input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = M[aOffset] == M[bOffset] ? 1 : 0`
- **Tag checks:** `T[aOffset] == T[bOffset] == inTag`
- **Tag updates:** `T[dstOffset] = u8`
- **Bit-size:** 128

| opcode | reserved | indirect | Flags | aOffset | args | bOffset | dstOffset |
|--------|----------|----------|-------|---------|------|---------|-----------|
| EQ | XXX | | inTag | | | | |

LT

Less-than check ($a < b$)

[See in table.](#)

- **Opcode:** 0x06
- **Category:** Compute - Comparators
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
 - **inTag:** The [tag/size](#) to check inputs against and tag the destination with.
- **Args:**

- **aOffset**: memory offset of the operation's left input
- **bOffset**: memory offset of the operation's right input
- **dstOffset**: memory offset specifying where to store operation's result
- **Expression:** $M[\text{dstOffset}] = M[\text{aOffset}] < M[\text{bOffset}] ? 1 : 0$
- **Tag checks:** $T[\text{aOffset}] == T[\text{bOffset}] == \text{inTag}$
- **Tag updates:** $T[\text{dstOffset}] = \text{u8}$
- **Bit-size:** 128

| opcode | reserved | flags | args | dstOffset | | |
|--------|----------|----------|-------|-----------|---------|-----------|
| LT | XXX | indirect | inTag | aOffset | bOffset | dstOffset |

LTE

Less-than-or-equals check ($a \leq b$)

See in table.

- **Opcode:** 0x07
- **Category:** Compute - Comparators
- **Flags:**
 - **indirect**: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[\text{offset}]]$ instead of the more standard $M[\text{offset}]$.
 - **inTag**: The **tag/size** to check inputs against and tag the destination with.
- **Args:**
 - **aOffset**: memory offset of the operation's left input
 - **bOffset**: memory offset of the operation's right input
 - **dstOffset**: memory offset specifying where to store operation's result
- **Expression:** $M[\text{dstOffset}] = M[\text{aOffset}] \leq M[\text{bOffset}] ? 1 : 0$
- **Tag checks:** $T[\text{aOffset}] == T[\text{bOffset}] == \text{inTag}$
- **Tag updates:** $T[\text{dstOffset}] = \text{u8}$
- **Bit-size:** 128

| opcode | reserved | flags | args | dstOffset | | |
|--------|----------|----------|-------|-----------|---------|-----------|
| LTE | XXX | indirect | inTag | aOffset | bOffset | dstOffset |

AND

Bitwise AND ($a \& b$)

See in table.

- **Opcode:** 0x08
- **Category:** Compute - Bitwise
- **Flags:**
 - **indirect**: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[\text{offset}]]$ instead of the more standard $M[\text{offset}]$.
 - **inTag**: The **tag/size** to check inputs against and tag the destination with. **field** type is NOT supported for this instruction.
- **Args:**
 - **aOffset**: memory offset of the operation's left input
 - **bOffset**: memory offset of the operation's right input

- dstOffset: memory offset specifying where to store operation's result
- Expression: $M[\text{dstOffset}] = M[\text{aOffset}] \text{ AND } M[\text{bOffset}]$
- Tag checks: $T[\text{aOffset}] == T[\text{bOffset}] == \text{inTag}$
- Tag updates: $T[\text{dstOffset}] = \text{inTag}$
- Bit-size: 128

| opcode | reserved | indirect | flags | aOffset | bOffset | args | dstOffset |
|--------|----------|----------|-------|---------|---------|------|-----------|
| AND | XXX | | inTag | | | | |

OR

Bitwise OR ($a \mid b$)

[See in table.](#)

- Opcode: 0x09
- Category: Compute - Bitwise
- Flags:
 - indirect: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[\text{offset}]]$ instead of the more standard $M[\text{offset}]$.
 - inTag: The [tag/size](#) to check inputs against and tag the destination with. [field](#) type is NOT supported for this instruction.
- Args:
 - aOffset: memory offset of the operation's left input
 - bOffset: memory offset of the operation's right input
 - dstOffset: memory offset specifying where to store operation's result
- Expression: $M[\text{dstOffset}] = M[\text{aOffset}] \text{ OR } M[\text{bOffset}]$
- Tag checks: $T[\text{aOffset}] == T[\text{bOffset}] == \text{inTag}$
- Tag updates: $T[\text{dstOffset}] = \text{inTag}$
- Bit-size: 128

| opcode | reserved | indirect | flags | aOffset | bOffset | args | dstOffset |
|--------|----------|----------|-------|---------|---------|------|-----------|
| OR | XXX | | inTag | | | | |

XOR

Bitwise XOR ($a \wedge b$)

[See in table.](#)

- Opcode: 0x0a
- Category: Compute - Bitwise
- Flags:
 - indirect: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[\text{offset}]]$ instead of the more standard $M[\text{offset}]$.
 - inTag: The [tag/size](#) to check inputs against and tag the destination with. [field](#) type is NOT supported for this instruction.
- Args:
 - aOffset: memory offset of the operation's left input
 - bOffset: memory offset of the operation's right input
 - dstOffset: memory offset specifying where to store operation's result
- Expression: $M[\text{dstOffset}] = M[\text{aOffset}] \text{ XOR } M[\text{bOffset}]$

- **Tag checks:** $T[aOffset] == T[bOffset] == \text{inTag}$
- **Tag updates:** $T[\text{dstOffset}] = \text{inTag}$
- **Bit-size:** 128

| opcode | reserved | indirect | flags | aOffset | bOffset | args | dstOffset |
|--------|----------|----------|-------|---------|---------|------|-----------|
| XOR | XXX | | inTag | | | | |

NOT

Bitwise NOT (inversion)

See in table.

- **Opcode:** 0x0b
- **Category:** Compute - Bitwise
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[offset]]$ instead of the more standard $M[offset]$.
 - **inTag:** The [tag/size](#) to check inputs against and tag the destination with. [field](#) type is NOT supported for this instruction.
- **Args:**
 - **aOffset:** memory offset of the operation's input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** $M[\text{dstOffset}] = \text{NOT } M[\text{aOffset}]$
- **Tag checks:** $T[aOffset] == \text{inTag}$
- **Tag updates:** $T[\text{dstOffset}] = \text{inTag}$
- **Bit-size:** 96

| opcode | reserved | indirect | flags | aOffset | args | dstOffset |
|--------|----------|----------|-------|---------|------|-----------|
| NOT | XXX | | inTag | | | |

SHL

Bitwise leftward shift ($a << b$)

See in table.

- **Opcode:** 0x0c
- **Category:** Compute - Bitwise
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[offset]]$ instead of the more standard $M[offset]$.
 - **inTag:** The [tag/size](#) to check inputs against and tag the destination with. [field](#) type is NOT supported for this instruction.
- **Args:**
 - **aOffset:** memory offset of the operation's left input
 - **bOffset:** memory offset of the operation's right input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** $M[\text{dstOffset}] = M[\text{aOffset}] << M[\text{bOffset}]$
- **Tag checks:** $T[aOffset] == \text{inTag}, T[bOffset] == \text{u8}$
- **Tag updates:** $T[\text{dstOffset}] = \text{inTag}$

- Bit-size: 128

| opcode | reserved | indirect | inTag | aOffset | args | bOffset | dstOffset |
|--------|----------|----------|-------|---------|------|---------|-----------|
| SHL | XXX | | | | | | |

SHR

Bitwise rightward shift ($a \gg b$)

See in table.

- **Opcode:** 0x0d
- **Category:** Compute - Bitwise
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
 - **inTag:** The `tag/size` to check inputs against and tag the destination with. `field` type is NOT supported for this instruction.
- **Args:**
 - **aOffset:** memory offset of the operation's left input
 - **bOffset:** memory offset of the operation's right input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = M[aOffset] >> M[bOffset]`
- **Tag checks:** `T[aOffset] == inTag, T[bOffset] == u8`
- **Tag updates:** `T[dstOffset] = inTag`
- Bit-size: 128

| opcode | reserved | indirect | inTag | aOffset | args | bOffset | dstOffset |
|--------|----------|----------|-------|---------|------|---------|-----------|
| SHR | XXX | | | | | | |

CAST

Type cast

See in table.

- **Opcode:** 0x0e
- **Category:** Type Conversions
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
 - **dstTag:** The `tag/size` to tag the destination with but not to check inputs against.
- **Args:**
 - **aOffset:** memory offset of word to cast
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = cast<dstTag>(M[aOffset])`
- **Details:** Cast a word in memory based on the `dstTag` specified in the bytecode. Truncates (`M[dstOffset] = M[aOffset] mod 2^dstsize`) when casting to a smaller type, left-zero-pads when casting to a larger type. See [here](#) for more details.
- **Tag updates:** `T[dstOffset] = dstTag`
- Bit-size: 96

| opcode | reserved | flags | args |
|--------|----------|-----------------|-------------------|
| CAST | XXX | indirect dstTag | aOffset dstOffset |

ADDRESS

Get the address of the currently executing I2 contract

See in table.

- **Opcode:** 0x0f
- **Category:** Execution Environment
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - `dstOffset`: memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.environment.address`
- **Tag updates:** `T[dstOffset] = field`
- **Bit-size:** 56

| opcode | reserved | flags | args |
|---------|----------|----------|-----------|
| ADDRESS | XXX | indirect | dstOffset |

SENDER

Get the address of the sender (caller of the current context)

See in table.

- **Opcode:** 0x10
- **Category:** Execution Environment
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - `dstOffset`: memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.environment.sender`
- **Tag updates:** `T[dstOffset] = field`
- **Bit-size:** 56

| opcode | reserved | flags | args |
|--------|----------|----------|-----------|
| SENDER | XXX | indirect | dstOffset |

FUNCTIONSELECTOR

Get the function selector of the contract function being executed

See in table.

- **Opcode:** 0x11
- **Category:** Execution Environment
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.environment.functionSelector`
- **Tag updates:** `T[dstOffset] = u32`
- **Bit-size:** 56

TRANSACTIONFEE

Get the computed transaction fee during teardown phase, zero otherwise

See in table.

- **Opcode:** 0x12
- **Category:** Execution Environment
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.environment.transactionFee`
- **Tag updates:** `T[dstOffset] = field`
- **Bit-size:** 56

CHAINID

Get this rollup's L1 chain ID

See in table.

- **Opcode:** 0x13
- **Category:** Execution Environment - Globals
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.environment.globals.chainId`
- **Tag updates:** `T[dstOffset] = field`
- **Bit-size:** 56

| opcode | reserved | flags | args |
|---------|----------|----------|-----------|
| CHAINID | XXX | indirect | dstOffset |

VERSION

Get this rollup's L2 version ID

[See in table.](#)

- **Opcode:** 0x14
- **Category:** Execution Environment - Globals
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - `dstOffset`: memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.environment.globals.version`
- **Tag updates:** `T[dstOffset] = field`
- **Bit-size:** 56

| opcode | reserved | flags | args |
|---------|----------|----------|-----------|
| VERSION | XXX | indirect | dstOffset |

BLOCKNUMBER

Get this L2 block's number

[See in table.](#)

- **Opcode:** 0x15
- **Category:** Execution Environment - Globals
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - `dstOffset`: memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.environment.globals.blocknumber`
- **Tag updates:** `T[dstOffset] = field`
- **Bit-size:** 56

| opcode | reserved | flags | args |
|-------------|----------|----------|-----------|
| BLOCKNUMBER | XXX | indirect | dstOffset |

TIMESTAMP

Get this L2 block's timestamp

See in table.

- **Opcode:** 0x16
- **Category:** Execution Environment - Globals
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - `dstOffset`: memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.environment.globals.timestamp`
- **Tag updates:** `T[dstOffset] = u64`
- **Bit-size:** 56

| 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 |
|-----------|----------|----------|----|------|-----------|----|----|
| opcode | reserved | flags | | args | | | |
| TIMESTAMP | XXX | indirect | | | dstOffset | | |

FEEPERL2GAS

Get the fee to be paid per "L2 gas" - constant for entire transaction

See in table.

- **Opcode:** 0x17
- **Category:** Execution Environment - Globals - Gas
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - `dstOffset`: memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.environment.globals.feePerL2Gas`
- **Tag updates:** `T[dstOffset] = field`
- **Bit-size:** 56

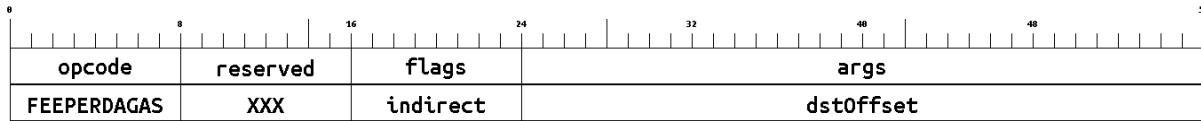
| 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 |
|-------------|----------|----------|----|-----------|----|----|----|
| opcode | reserved | flags | | args | | | |
| FEEPERL2GAS | XXX | indirect | | dstOffset | | | |

FEEPERDAGS

Get the fee to be paid per "DA gas" - constant for entire transaction

See in table.

- **Opcode:** 0x18
 - **Category:** Execution Environment - Globals - Gas
 - **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
 - **Args:**
 - **dstOffset:** memory offset specifying where to store operation's result
 - **Expression:** `M[dstOffset] = context.environment.globals.feePerDaGas`
 - **Tag updates:** `T[dstOffset] = field`
 - **Bit-size:** 56

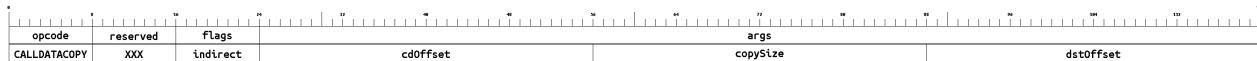


CALLDATACOPY

Copy calldata into memory

See in table.

- **Opcode:** 0x19
 - **Category:** Execution Environment - Calldata
 - **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
 - **Args:**
 - **cdOffset:** offset into calldata to copy from
 - **copySize:** number of words to copy
 - **dstOffset:** memory offset specifying where to copy the first word to
 - **Expression:** `M[dstOffset:dstOffset+copySize] = context.environment.calldata[cdOffset:cdOffset+copySize]`
 - **Details:** Calldata is read-only and cannot be directly operated on by other instructions. This instruction moves words from calldata into memory so they can be operated on normally.
 - **Tag updates:** `T[dstOffset:dstOffset+copySize] = field`
 - **Bit-size:** 120



L2GASLEFT

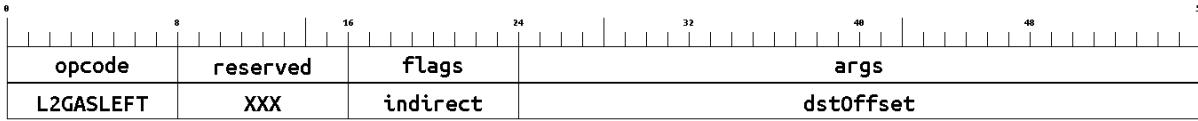
Remaining "L2 gas" for this call (after this instruction)

See in table.

- **Opcode:** 0x1a
 - **Category:** Machine State - Gas
 - **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect

offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.

- **Args:**
 - `dstOffset`: memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.MachineState.l2GasLeft`
- **Tag updates:** `T[dstOffset] = u32`
- **Bit-size:** 56

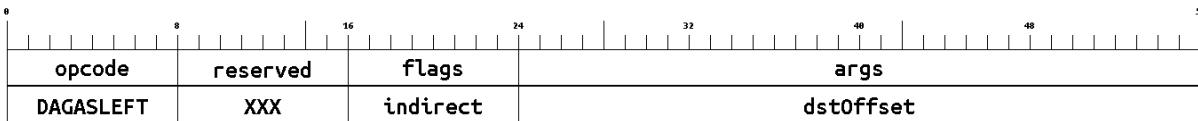


DAGASLEFT

Remaining "DA gas" for this call (after this instruction)

[See in table.](#)

- **Opcode:** 0x1b
- **Category:** Machine State - Gas
- **Flags:**
 - `indirect`: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - `dstOffset`: memory offset specifying where to store operation's result
- **Expression:** `M[dstOffset] = context.machineState.daGasLeft`
- **Tag updates:** `T[dstOffset] = u32`
- **Bit-size:** 56



JUMP

Jump to a location in the bytecode

[See in table.](#)

- **Opcode:** 0x1c
- **Category:** Machine State - Control Flow
- **Args:**
 - `loc`: target location to jump to
- **Expression:** `context.machineState.pc = loc`
- **Details:** Target location is an immediate value (a constant in the bytecode).
- **Bit-size:** 48

| 8 | 8 | 16 | 24 | 32 | 40 | 48 |
|--------|----------|----|------|-----|----|----|
| opcode | reserved | | args | | | |
| JUMP | XXX | | | loc | | |

JUMPI

Conditionally jump to a location in the bytecode

See in table.

- **Opcode:** 0x1d
- **Category:** Machine State - Control Flow
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **loc:** target location conditionally jump to
 - **condOffset:** memory offset of the operations 'conditional' input
- **Expression:** `context.machineState.pc = M[condOffset] > 0 ? loc : context.machineState.pc`
- **Details:** Target location is an immediate value (a constant in the bytecode). `T[condOffset]` is not checked because the greater-than-zero suboperation is the same regardless of type.
- **Bit-size:** 88

| 8 | 8 | 16 | 24 | 32 | 40 | 48 |
|--------|----------|----------|-----|------|------------|----|
| opcode | reserved | flags | | args | | |
| JUMPI | XXX | indirect | loc | | condOffset | |

INTERNALCALL

Make an internal call. Push the current PC to the internal call stack and jump to the target location.

See in table.

- **Opcode:** 0x1e
- **Category:** Machine State - Control Flow
- **Args:**
 - **loc:** target location to jump/call to
- **Expression:**

```
context.machineState.internalCallStack.push(context.machineState.pc)
context.machineState.pc = loc
```

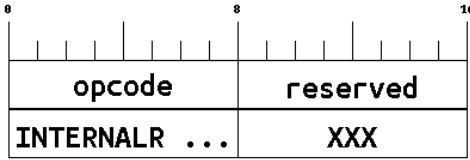
- **Details:** Target location is an immediate value (a constant in the bytecode).
- **Bit-size:** 48

INTERNALRETURN

Return from an internal call. Pop from the internal call stack and jump to the popped location.

See in table.

- **Opcode:** 0x1
- **Category:** Machine State - Control Flow
- **Expression:** `context.machineState.pc = context.machineState.internalCallStack.pop()`
- **Bit-size:** 16



SET

Set a memory word from a constant in the bytecode

See in table.

- **Opcode:** 0x20
- **Category:** Machine State - Memory
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
 - **inTag:** The `type/size` to check inputs against and tag the destination with. `field` type is NOT supported for SET.
- **Args:**
 - **const:** an N-bit constant value from the bytecode to store in memory (any type except `field`)
 - **dstOffset:** memory offset specifying where to store the constant
- **Expression:** `M[dstOffset] = const`
- **Details:** Set memory word at `dstOffset` to `const`'s immediate value. `const`'s bit-size (N) can be 8, 16, 32, 64, or 128 based on `inTag`. It cannot be 256 (`field` type)!
- **Tag updates:** `T[dstOffset] = inTag`
- **Bit-size:** 64+N



MOV

Move a word from source memory location to destination

See in table.

- **Opcode:** 0x21
- **Category:** Machine State - Memory
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **srcOffset:** memory offset of word to move

- **dstOffset:** memory offset specifying where to store that word
- **Expression:** $M[\text{dstOffset}] = M[\text{srcOffset}]$
- **Tag updates:** $T[\text{dstOffset}] = T[\text{srcOffset}]$
- **Bit-size:** 88

| opcode | reserved | flags | args | | | |
|--------|----------|----------|-----------|--|--|-----------|
| MOV | XXX | indirect | srcOffset | | | dstOffset |

CMOV

Move a word (conditionally chosen) from one memory location to another ($d \leftarrow \text{cond} > 0 ? a : b$)

See in table.

- **Opcode:** 0x22
- **Category:** Machine State - Memory
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[\text{offset}]]$ instead of the more standard $M[\text{offset}]$.
- **Args:**
 - **aOffset:** memory offset of word 'a' to conditionally move
 - **bOffset:** memory offset of word 'b' to conditionally move
 - **condOffset:** memory offset of the operations 'conditional' input
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:** $M[\text{dstOffset}] = M[\text{condOffset}] > 0 ? M[\text{aOffset}] : M[\text{bOffset}]$
- **Details:** One of two source memory locations is chosen based on the condition. $T[\text{condOffset}]$ is not checked because the greater-than-zero suboperation is the same regardless of type.
- **Tag updates:** $T[\text{dstOffset}] = M[\text{condOffset}] > 0 ? T[\text{aOffset}] : T[\text{bOffset}]$
- **Bit-size:** 152

| opcode | reserved | flags | args | | | |
|--------|----------|----------|---------|---------|------------|-----------|
| CMOV | XXX | Indirect | aOffset | bOffset | condOffset | dstOffset |

SLOAD

Load a word from this contract's persistent public storage. Zero is loaded for unwritten slots.

See in table.

- **Opcode:** 0x23
- **Category:** World State - Public Storage
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like $M[M[\text{offset}]]$ instead of the more standard $M[\text{offset}]$.
- **Args:**
 - **slotOffset:** memory offset of the storage slot to load from
 - **dstOffset:** memory offset specifying where to store operation's result
- **Expression:**

```
M[dstOffset] = S[M[slotOffset]]
```

- Details:

```
// Expression is shorthand for
leafIndex = hash(context.environment.address, M[slotOffset])
exists = context.worldState.publicStorage.has(leafIndex) // exists == previously-written
if exists:
    value = context.worldState.publicStorage.get(leafIndex: leafIndex)
else:
    value = 0
M[dstOffset] = value
```

- World State access tracing:

```
context.worldStateAccessTrace.publicStorageReads.append(
    TracedStorageRead {
        callPointer: context.environment.callPointer,
        slot: M[slotOffset],
        exists: exists, // defined above
        value: value, // defined above
        counter: ++context.worldStateAccessTrace.accessCounter,
    }
)
```

- Triggers downstream circuit operations: Storage slot siloing (hash with contract address), public data tree membership check
- Tag updates: T[dstOffset] = field
- Bit-size: 88

| opcode | reserved | flags | slotOffset | args | dstOffset |
|--------|----------|----------|------------|------|-----------|
| SLOAD | XXX | indirect | | | |

SSTORE

Write a word to this contract's persistent public storage

See in table.

- Opcode: 0x24
- Category: World State - Public Storage
- Flags:
 - indirect: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like M[M[offset]] instead of the more standard M[offset].
- Args:
 - srcOffset: memory offset of the word to store
 - slotOffset: memory offset containing the storage slot to store to
- Expression:

```
S[M[slotOffset]] = M[srcOffset]
```

- Details:

```
// Expression is shorthand for
context.worldState.publicStorage.set({
  leafIndex: hash(context.environment.address, M[slotOffset]),
  leaf: M[srcOffset],
})
```

- World State access tracing:

```
context.worldStateAccessTrace.publicStorageWrites.append(
  TracedStorageWrite {
    callPointer: context.environment.callPointer,
    slot: M[slotOffset],
    value: M[srcOffset],
    counter: ++context.worldStateAccessTrace.accessCounter,
  }
)
```

- Triggers downstream circuit operations: Storage slot siloing (hash with contract address), public data tree update
- Bit-size: 88

| opcode | reserved | flags | args | slotOffset |
|--------|----------|----------|-----------|------------|
| SSTORE | XXX | indirect | srcOffset | |

NOTEHASHEXISTS

Check whether a note hash exists in the note hash tree (as of the start of the current block)

See in table.

- Opcode: 0x25
- Category: World State - Notes & Nullifiers
- Flags:
 - **indirect**: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- Args:
 - **noteHashOffset**: memory offset of the note hash
 - **leafIndexOffset**: memory offset of the leaf index
 - **existsOffset**: memory offset specifying where to store operation's result (whether the note hash leaf exists)
- Expression:

```
exists = context.worldState.noteHashes.has({
  leafIndex: M[leafIndexOffset]
  leaf: hash(context.environment.address, M[noteHashOffset]),
})
M[existsOffset] = exists
```

- World State access tracing:

```
context.worldStateAccessTrace.noteHashChecks.append(
  TracedNoteHashCheck {
    callPointer: context.environment.callPointer,
    leafIndex: M[leafIndexOffset]
    noteHash: M[noteHashOffset],
  }
)
```

- Triggers downstream circuit operations: Note hash siloing (hash with storage contract address), note hash tree membership check
- Tag updates: `T[existsOffset] = u8`
- Bit-size: 120

EMITNOTEHASH

Emit a new note hash to be inserted into the note hash tree

[See in table.](#)

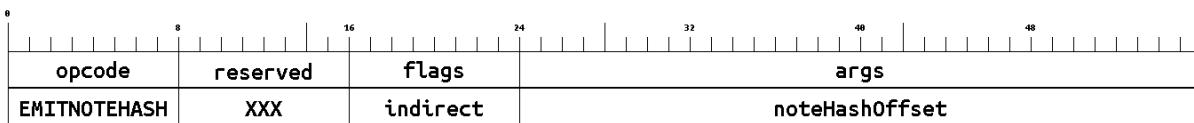
- **Opcode:** 0x26
- **Category:** World State - Notes & Nullifiers
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - `noteHashOffset`: memory offset of the note hash
- **Expression:**

```
context.worldState.noteHashes.append(
    hash(context.environment.address, M[noteHashOffset])
)
```

- **World State access tracing:**

```
context.worldStateAccessTrace.noteHashes.append(
    TracedNoteHash {
        callPointer: context.environment.callPointer,
        noteHash: M[noteHashOffset], // unsiloed note hash
        counter: ++context.worldStateAccessTrace.accessCounter,
    }
)
```

- Triggers downstream circuit operations: Note hash siloing (hash with contract address), note hash tree insertion.
- Bit-size: 56



NULIFIEREXISTS

Check whether a nullifier exists in the nullifier tree (including nullifiers from earlier in the current transaction or from earlier in the current block)

[See in table.](#)

- **Opcode:** 0x27
- **Category:** World State - Notes & Nullifiers
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.

- **Args:**
 - **nullifierOffset**: memory offset of the unsiloed nullifier
 - **addressOffset**: memory offset of the storage address
 - **existsOffset**: memory offset specifying where to store operation's result (whether the nullifier exists)
- **Expression:**

```
exists = pendingNullifiers.has(M[addressOffset], M=nullifierOffset) || context.worldState.nullifiers.has(
    hash(M[addressOffset], M=nullifierOffset))
)
M[existsOffset] = exists
```

- **World State access tracing:**

```
context.worldStateAccessTrace.nullifierChecks.append(
    TracedNullifierCheck {
        callPointer: context.environment.callPointer,
        nullifier: M=nullifierOffset,
        address: M[addressOffset],
        exists: exists, // defined above
        counter: ++context.worldStateAccessTrace.accessCounter,
    }
)
```

- **Triggers downstream circuit operations:** Nullifier siloing (hash with storage contract address), nullifier tree membership check
- **Tag updates:** T[existsOffset] = u8
- **Bit-size:** 120

EMITNULLIFIER

Emit a new nullifier to be inserted into the nullifier tree

See in table.

- **Opcode:** 0x28
- **Category:** World State - Notes & Nullifiers
- **Flags:**
 - **indirect**: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like M[M[offset]] instead of the more standard M[offset].
- **Args:**
 - **nullifierOffset**: memory offset of nullifier
- **Expression:**

```
context.worldState.nullifiers.append(
    hash(context.environment.address, M=nullifierOffset)
)
```

- **World State access tracing:**

```
context.worldStateAccessTrace.nullifiers.append(
    TracedNullifier {
        callPointer: context.environment.callPointer,
        nullifier: M=nullifierOffset, // unsiloed nullifier
        counter: ++context.worldStateAccessTrace.accessCounter,
    }
)
```

- Triggers downstream circuit operations: Nullifier siloing (hash with contract address), nullifier tree non-membership-check and insertion.
- Bit-size: 56



L1TOL2MSGEXISTS

Check if a message exists in the L1-to-L2 message tree

See in table.

- **Opcode:** 0x29
- **Category:** World State - Messaging
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **msgHashOffset:** memory offset of the message hash
 - **msgLeafIndexOffset:** memory offset of the message's leaf index in the L1-to-L2 message tree
 - **existsOffset:** memory offset specifying where to store operation's result (whether the message exists in the L1-to-L2 message tree)
- **Expression:**

```
exists = context.worldState.l1ToL2Messages.has({
    leafIndex: M[msgLeafIndexOffset], leaf: M[msgHashOffset]
})
M[existsOffset] = exists
```

- **World State access tracing:**

```
context.worldStateAccessTrace.l1ToL2MessagesChecks.append(  
    L1ToL2Message {  
        callPointer: context.environment.callPointer,  
        leafIndex: M[msgLeafIndexOffset],  
        msgHash: M[msgHashOffset],  
        exists: exists, // defined above  
    }  
)
```

- Triggers downstream circuit operations: L1-to-L2 message tree membership check
- **Tag updates:**

```
T[existsOffset] = u8,
```

- **Bit-size:** 120

GETCONTRACTINSTANCE

Copies contract instance data to memory

See in table.

- **Opcode:** 0x2a
- **Category:** Other
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **addressOffset:** memory offset of the contract instance address
 - **dstOffset:** location to write the contract instance information to
- **Expression:**

```
M[dstOffset:dstOffset+CONTRACT_INSTANCE_SIZE+1] = [
    instance_found_in_address,
    instance.salt ?? 0,
    instance.deployer ?? 0,
    instance.contractClassId ?? 0,
    instance.initializationHash ?? 0,
    instance.portalContractAddress ?? 0,
    instance.publicKeysHash ?? 0,
]
```

- **Additional AVM circuit checks:** TO-DO
- **Triggers downstream circuit operations:** TO-DO
- **Tag updates:** `T[dstOffset:dstOffset+CONTRACT_INSTANCE_SIZE+1] = field`
- **Bit-size:** 88

EMITUNENCRYPTEDLOG

Emit an unencrypted log

See in table.

- **Opcode:** 0x2b
- **Category:** Accrued Substate - Logging
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **logOffset:** memory offset of the data to log
 - **logSizeOffset:** memory offset to number of words to log
- **Expression:**

```
context.accruedSubstate.unencryptedLogs.append(
    UnencryptedLog {
        address: context.environment.address,
        log: M[logOffset:logOffset+M[logSizeOffset]],
    }
)
```

- **Bit-size:** 88

| opcode | reserved | flags | args | | | | | | | | | | | | |
|---------------|----------|----------|-----------|--|--|--|--|--|--|--|--|--|--|--|---------|
| EMITUNENC ... | XXX | indirect | logOffset | | | | | | | | | | | | logSize |

SENDL2TOL1MSG

Send an L2-to-L1 message

See in table.

- **Opcode:** 0x2c
- **Category:** Accrued Substate - Messaging
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **recipientOffset:** memory offset of the message recipient
 - **contentOffset:** memory offset of the message content
- **Expression:**

```
context.acquiredSubstate.sentL2ToL1Messages.append(
    SentL2ToL1Message {
        address: context.environment.address,
        recipient: M[recipientOffset],
        message: M[contentOffset]
    }
)
```

- **Bit-size:** 88

| opcode | reserved | flags | args | | | | | | | | | | | | |
|---------------|----------|----------|-----------|--|--|--|--|--|--|--|--|--|--|--|---------|
| SENDL2TOL1MSG | XXX | indirect | msgOffset | | | | | | | | | | | | msgSize |

CALL

Call into another contract

See in table.

- **Opcode:** 0x2d
- **Category:** Control Flow - Contract Calls
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **gasOffset:** offset to two words containing `{l2GasLeft, daGasLeft}`: amount of gas to provide to the callee
 - **addrOffset:** address of the contract to call
 - **argsOffset:** memory offset to args (will become the callee's calldata)
 - **argsSizeOffset:** memory offset for the number of words to pass via callee's calldata
 - **retOffset:** destination memory offset specifying where to store the data returned from the callee
 - **retSize:** number of words to copy from data returned by callee

- **successOffset**: destination memory offset specifying where to store the call's success (0: failure, 1: success)
- **Expression**:

```
// instr.args are { gasOffset, addrOffset, argsOffset, retOffset, retSize }
chargeGas(context,
    l2GasCost=M[instr.args.gasOffset],
    daGasCost=M[instr.args.gasOffset+1])
traceNestedCall(context, instr.args.addrOffset)
nestedContext = deriveContext(context, instr.args, isStaticCall=false)
execute(nestedContext)
updateContextAfterNestedCall(context, instr.args, nestedContext)
```

- **Details**: Creates a new (nested) execution context and triggers execution within that context. Execution proceeds in the nested context until it reaches a halt at which point execution resumes in the current/calling context. A non-existent contract or one with no code will return success. "Nested contract calls" provides a full explanation of this instruction along with the shorthand used in the expression above. The explanation includes details on charging gas for nested calls, nested context derivation, world state tracing, and updating the parent context after the nested call halts.
- **Tag checks**: T[gasOffset] == T[gasOffset+1] == T[gasOffset+2] == u32
- **Tag updates**:

```
T[successOffset] = u8
T[retOffset:retOffset+retSize] = field
```

- **Bit-size**: 248

| opcode | reserved | Flags | gasOffset | addrOffset | argsOffset | argsSize | retOffset | retSize | successOffset |
|--------|----------|----------|-----------|------------|------------|----------|-----------|---------|---------------|
| CALL | XXX | Indirect | | | | | | | |

STATICCALL

Call into another contract, disallowing World State and Accrued Substate modifications

See in table.

- **Opcode**: 0x2e
- **Category**: Control Flow - Contract Calls
- **Flags**:
 - **indirect**: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like M[M[offset]] instead of the more standard M[offset].
- **Args**:
 - **gasOffset**: offset to two words containing {l2GasLeft, daGasLeft}: amount of gas to provide to the callee
 - **addrOffset**: address of the contract to call
 - **argsOffset**: memory offset to args (will become the callee's calldata)
 - **argsSizeOffset**: memory offset for the number of words to pass via callee's calldata
 - **retOffset**: destination memory offset specifying where to store the data returned from the callee
 - **retSize**: number of words to copy from data returned by callee
 - **successOffset**: destination memory offset specifying where to store the call's success (0: failure, 1: success)
- **Expression**:

```
// instr.args are { gasOffset, addrOffset, argsOffset, retOffset, retSize }
chargeGas(context,
    l2GasCost=M[instr.args.gasOffset],
    daGasCost=M[instr.args.gasOffset+1])
traceNestedCall(context, instr.args.addrOffset)
```

- **Details:** Same as `CALL`, but disallows World State and Accrued Substate modifications. "[Nested contract calls](#)" provides a full explanation of this instruction along with the shorthand used in the expression above. The explanation includes details on charging gas for nested calls, nested context derivation, world state tracing, and updating the parent context after the nested call halts.
- **Tag checks:** `T[gasOffset] == T[gasOffset+1] == T[gasOffset+2] == u32`
- **Tag updates:**

```
T[successOffset] = u8
T[retOffset:retOffset+retSize] = field
```

- **Bit-size:** 248

| opcode | reserved | flags | gasOffset | addrOffset | argOffset | args | retOffset | retSize | successOffset |
|------------|----------|----------|-----------|------------|-----------|------|-----------|---------|---------------|
| STATICCALL | XXX | indirect | | | | | | | |

RETURN

Halt execution within this context (without revert), optionally returning some data

See in table.

- **Opcode:** 0x2f
- **Category:** Control Flow - Contract Calls
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **retOffset:** memory offset of first word to return
 - **retSize:** number of words to return
- **Expression:**

```
context.contractCallResults.output = M[retOffset:retOffset+retSize]
halt
```

- **Details:** Return control flow to the calling context/contract. Caller will accept World State and Accrued Substate modifications. See "[Halting](#)" to learn more. See "[Nested contract calls](#)" to see how the caller updates its context after the nested call halts.
- **Bit-size:** 88

| opcode | reserved | flags | args | retSize |
|--------|----------|----------|-----------|---------|
| RETURN | XXX | indirect | retOffset | retSize |

REVERT

Halt execution within this context as `reverted`, optionally returning some data

See in table.

- **Opcode:** 0x30
- **Category:** Control Flow - Contract Calls
- **Flags:**
 - **indirect:** Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **retOffset:** memory offset of first word to return

- **retSize**: number of words to return

- **Expression:**

```
context.contractCallResults.output = M[retOffset:retOffset+retSize]
context.contractCallResults.reverted = true
halt
```

- **Details:** Return control flow to the calling context/contract. Caller will reject World State and Accrued Substate modifications. See "[Halting](#)" to learn more. See "[Nested contract calls](#)" to see how the caller updates its context after the nested call halts.

- **Bit-size:** 88

| opcode | reserved | flags | args |
|--------|----------|----------|------------------------|
| REVERT | XXX | indirect | retOffset retSize |

TORADIXLE

Convert a word to an array of limbs in little-endian radix form

[See in table.](#)

- **Opcode:** 0x31
- **Category:** Conversions
- **Flags:**
 - **indirect**: Toggles whether each memory-offset argument is an indirect offset. Rightmost bit corresponds to 0th offset arg, etc. Indirect offsets result in memory accesses like `M[M[offset]]` instead of the more standard `M[offset]`.
- **Args:**
 - **srcOffset**: memory offset of word to convert.
 - **dstOffset**: memory offset specifying where the first limb of the radix-conversion result is stored.
 - **radix**: the maximum bit-size of each limb.
 - **numLimbs**: the number of limbs the word will be converted into.
- **Expression:** TBD: Storage of limbs and if `T[dstOffset]` is constrained to U8
- **Details:** The limbs will be stored in a contiguous memory block starting at `dstOffset`.
- **Tag checks:** `T[srcOffset] == field`
- **Bit-size:** 152

AVM Circuit

The AVM circuit's purpose is to prove execution of a sequence of instructions for a public execution request. Regardless of whether execution succeeds or reverts, the circuit always generates a valid proof of execution.

AVM Circuit

The AVM circuit's purpose is to prove execution of a sequence of instructions for a public execution request. Regardless of whether execution succeeds or reverts, the circuit always generates a valid proof of execution.

Introduction

Circuit Architecture Outline

The circuit is comprised of the following components:

- **Bytecode Table:** includes bytecode for all calls, indexed by call pointer and program counter.
- **Instruction Controller:** fetches an instruction from the Bytecode Table. Decodes the instructions into sub-operations to be forwarded to other modules.
- **Intermediate Registers:** for staging sub-operation inputs and outputs.
- **Control Flow Unit:** maintains program counter and call pointer. Processes control-flow sub-operations (program counter increments, internal call stack operations, contract call operations).
- **Gas Controller:** tracks remaining gas for current call. Processes gas tracking sub-operations.
- **Memory Controller:** processes memory sub-operations to load and store data between memory and intermediate registers.
- **Storage Controller:** processes storage sub-operations to load and store data between storage and intermediate registers.
- **Side-effect Accumulator:** processes side-effect sub-operations by pushing to a side-effect vector.

- **Chiplets:** perform compute operations on intermediate registers. Some chiplets include the ALU, Conditional Unit, Type Converter, and Crypto/Hash Gadgets.
- **Circuit I/O:** data structures used to ingest circuit inputs and emit outputs.

Bytecode Table

To review, the AVM circuit's primary purpose is to prove execution of the proper sequence of instructions given a contract call's bytecode and inputs. The circuit will prove correct execution of any nested contract calls as well. Each nested call will have its own bytecode and inputs, but will be processed within the same circuit.

Prior to the VM circuit's execution, a vector is assembled to contain the bytecode for all of a request's contract calls (initial and nested). If a request's execution contains contract calls to contracts A, B, C, and D (in that order), the VM circuit's bytecode vector will contain A's bytecode, followed by B's, C's, and finally D's. Each one will be zero-padded to some constant length `MAX_PUBLIC_INSTRUCTIONS_PER_CONTRACT`.

Each entry in the bytecode vector will be paired with a call pointer and program counter. This **Bytecode Table** maps a call pointer and program counter to an instruction, and is used by the Instruction Controller to fetch instructions.

Note: "call pointer" is expanded on in a later section.

Each contract's public bytecode is committed to during contract deployment. As part of the AVM circuit verification algorithm, the bytecode vector (as a concatenation of all relevant contract bytecodes) is verified against the corresponding bytecode commitments. This is expanded on in "[Bytecode Validation Circuit](#)". While the AVM circuit enforces that the correct instructions are executed according to its bytecode table, the verifier checks that bytecode table against the previously validated bytecode commitments.

Instruction Controller

The Instruction Controller's responsibilities include instruction fetching and decoding.

Instruction fetching

The Instruction Controller's **instruction fetch** mechanism makes use of the bytecode table to determine which instruction to execute based on the call pointer and program counter. Each instruction fetch corresponds to a circuit lookup to enforce that the correct instruction is processed for a given contract and program counter.

The combination of the instruction fetch circuitry, the bytecode table, and the "[Bytecode Validation Circuit](#)" ensure that VM circuit processes the proper sequence of instructions.

Instruction decoding and sub-operations

An instruction (its opcode, flags, and arguments) represents some high-level VM operation. For example, an `ADD` instruction says "add two items from memory and store the result in memory". The Instruction Controller **instruction decode** mechanism decodes instructions into sub-operations. While an instruction likely requires many circuit components, a **sub-operation** is a smaller task that can be fed to just one VM circuit component for processing. By decoding an instruction into sub-operations, the VM circuit translates high-level instructions into smaller achievable tasks. To continue with the `ADD` example, it would translate "add two items from memory and store the result in memory" to "load an item from memory, load another item from memory, add them, and store the result to memory."

A pre-computed/hardcoded **sub-operations table** maps instruction opcodes to sub-operations. This provides the Instruction Controller with everything it needs to decode

an instruction.

The Instruction Controller forwards sub-operations according to the following categorizations:

- Control flow sub-operations are forwarded to the Control Flow Unit
- Gas tracking sub-operations are forwarded to the Gas Controller
- Memory sub-operations are forwarded to the Memory Controller
- Storage sub-operations are forwarded to the Storage Controller
- Side-effect sub-operations are forwarded to the Side-effect Controller
- A chiplet sub-operation is forwarded to the proper chiplet

TODO: table of all sub-operations by category (copy from hackmd with updates)

Note: for simple instructions (like `ADD`), the instruction can be fetched and all sub-operations can be processed in a single clock cycle. Since the VM circuit has limited resources, some complex instructions (like `CALLDATACOPY`) involve too many sub-operations to be processed in one clock cycle. A "clock cycle" in the AVM circuit represents the smallest subdivision of time during which some parallel operations can be performed. A clock cycle corresponds to a row in the circuit's operations trace. Simple instructions correspond to only a single row in this trace, but complex instructions span multiple rows. A `CLK` column tracks the clock cycle for each row and its set of sub-operations.

Decoding example

The `ADD` instruction is decoded into two `LOAD` memory sub-operations, an `ADD` ALU (chiplet) sub-operation, and a `STORE` memory sub-operation.

Take the following `ADD` instruction as an example: `ADD<u32> a0ffset b0ffset dst0ffset`. Assuming this instruction is executed as part of contract call with pointer `C`, it is decoded into the following sub-operations:

```

// Load word from call's memory into register Ia (index 0)
LOAD 0 aOffset // Ia = M<C>[aOffset]
// Load word from call's memory into register Ib (index 1)
LOAD 1 bOffset // Ib = M<C>[aOffset]
// Use the ALU chiplet in ADD<32> mode to add registers Ia and Ib
// Place the results in Ic
ADD<u32> // Ic = ALU_ADD<u32>(Ia, Ib)
// Store results of addition from register Ic (index 2) to memory
STORE 2 dstOffset

```

Note: the **ADD** instruction is an example of a "simple" instruction that can be fully processed in a single clock cycle. All four of the above-listed sub-operations happen in one clock cycle and therefore take up only a single row in the circuit's operations trace.

Intermediate Registers

User code (AVM bytecode) has no concept of "registers", and so instructions often operate directly on user memory. Sub-operations on the other hand operate on intermediate registers. The only circuit component that has direct access to memory is the Memory Controller (further explained later), and therefore only memory sub-operations access memory. All other sub-operations operate on **intermediate registers** which serve as a staging ground between memory and the various processing components of the VM circuit.

Three intermediate registers exist: I_a , I_b , and I_c .

Refer to "[AVM State Model](#)" for more details on the absence of "external registers" in the AVM.

Control Flow Unit

Processes updates to the program counter and call pointer to ensure that execution proceeds properly from one instruction to the next.

Program Counter

The Control Flow Unit's **program counter** is an index into the bytecode of the current call's contract. For most instructions, the Control Flow Unit will simply increment the program counter. Certain instructions (like `JUMP`) decode into control flow sub-operations (like `PCSTORE`). The Control Flow Unit processes such instructions to update the program counter.

Call Pointer

A **contract call pointer** uniquely identifies a contract call among all contract calls processed by the current circuit. The Control Flow Unit tracks the currently active call pointer and the next available one. When a nested contract call is encountered, it assigns it the next available call pointer (`callPointer = nextCallPointer++`) and increments that next pointer value. It then sets the program counter to 0.

A request's initial contract call is assigned call pointer of 1. The first nested contract call encountered during execution is assigned call pointer of 2. The Control Flow Unit assigns call pointers based on execution order.

There is certain information that must be tracked by the VM circuit on a per-call basis. For example, each call will correspond to the execution of a different contract's bytecode, and each call will access call-specific memory. As a per-call unique identifier, the contract call pointer enables bytecode and memory lookups, among other things, on a per-call basis.

"Input" and "output" call pointers

It is important to note that the initial contract call's pointer is 1, not 0. The zero call pointer is a special case known as the "input" call pointer.

As expanded on later, the VM circuit memory table has a separate section for each call pointer. The memory table section for the input call pointer is reserved for the initial call's `calldata`. This will be expanded on later.

Internal Call Stack

TODO

Nested contract calls

TODO

Initializing nested call context

TODO

Snapshotting and restoring context

TODO

Memory Controller

The VM circuit's **Memory Controller** processes loads and stores between intermediate registers and memory.

Memory Sub-operations

When decoded, instructions that operate on memory map to some Memory Controller sub-operations. A memory read maps to a `LOAD` sub-operation which loads a word from memory into an intermediate register. The memory offset for this sub-operation is generally specified by an instruction argument. Similarly, a memory write maps to a `STORE` sub-operation which stores a word from an intermediate register to memory.

User Memory

This table tracks all memory `Read` or `Write` operations. As introduced in the "[Memory State Model](#)", a memory cell is indexed by a 32-bit unsigned integer (`u32`), i.e., the memory capacity is of 2^{32} words. Each word is associated with a tag defining its type (`uninitialized`, `u8`, `u16`, `u32`, `u64`, `u128`, `field`). At the beginning of a new call, each memory cell is of type `uninitialized` and has value 0.

The main property enforcement of this table concerns read/write consistency of every memory cell. This must ensure:

- Each initial read on a memory cell must have value 0 (`uninitialized` is compatible with any other type).
- Each read on a memory cell must have the same value and the same tag as those set by the last write on this memory cell.

In addition, this table ensures that the instruction tag corresponding to a memory operation is the same as the memory cell tag. The instruction tag is passed to the memory controller and added to the pertaining row(s) of this table. Note that this is common for an instruction to generate several memory operations and thus several rows in this table.

The user memory table essentially consists of the following columns:

- `CALL_PTR`: call pointer uniquely identifying the contract call
- `CLK`: clock value of the memory operation
- `ADDR`: address (type `u32`) pertaining to the memory operation
- `VAL`: value which is read (resp. written) from (resp. to) the memory address
- `TAG`: tag associated to this memory address
- `IN_TAG`: tag of the pertaining instruction
- `RW`: boolean indicating whether memory operation is read or write
- `TAG_ERR`: boolean set to true if there is a mismatch between `TAG` and `IN_TAG`

To facilitate consistency check, the rows are sorted by `CALL_PTR` then by `ADDR` and then by `CLK` in ascending (arrow of time) order. Any (non-initial) read operation row is constrained to have the same `VAL` and `TAG` than the previous row. A write operation does not need to be constrained.

The tag consistency check can be performed within every row (order of rows does not matter).

Note that `CLK` also plays the role of a foreign key to point to the corresponding sub-operation. This is crucial to enforce consistency of copied values between the sub-operations and memory table.

Calldata

TODO

Initial call's calldata

Any lookup into calldata from a request's initial contract call must retrieve a value matching the `calldata` public inputs column. To enforce this, an equivalence check

is applied between the `calldata` column and the memory trace for user memory accesses that use "input call pointer".

Storage Controller

TODO

Side-effect Accumulator

TODO

Chiplets

A chiplet is essentially a sub-circuit for performing specialized sub-operations. A chiplet is defined as a dedicated table (a set of columns and relations) in the AVM circuit that is activated when the relevant sub-operation is used. The main rationale behind the use of chiplets is to offload specialized computations to a region of the circuit *outside the main operations trace and instruction controller* where the computations might require many rows and/or additional dedicated columns. In addition, this approach offers strong modularity for the operations implemented as chiplets.

The interaction between a chiplet and the instruction controller follows the following pattern:

1. The **inputs** of a chiplet sub-operation are loaded to the respective intermediate registers (usually I_a, I_b).
2. The dedicated chiplet fetches/copies the content of the intermediate registers from the **operations trace** in its own table and executes the operation.

3. The output of the operation is copied back to the **operations trace** in a specific register (usually I_c). This register is usually involved in a write memory sub-operation.

In addition to the mentioned inputs and output, some other relevant information such as the instruction tag might be copied as well to the chiplet.

In the circuit, the transmission of the input/output between the **chiplet trace** and the **operations trace** are performed through lookup or permutation constraints, i.e., they ensure that all relevant intermediate registers have the same values between both traces. The unique key of this mapping is `CLK` which is basically used as a "DB foreign key" from the **chiplet trace** pointing to corresponding entry in the **operations trace**.

The planned chiplets for the AVM are:

- **ALU:** Arithmetic and bitwise operations such as addition, multiplication, XOR, etc...
- **Type Converter:** Dedicated to casting words between different types and/or type constraints.
- **Gadgets:** Relevant cryptographic operations or other computationally intensive operations. There will likely be multiple chiplets of this category, including `Poseidon2Permutation`, `Keccakf1600`, and `ECADD`.

Circuit I/O

How do "Public Inputs" work in the AVM circuit?

ZK circuit proof systems generally define some mechanism for "public inputs" for which witness values must be communicated in full to a verifier. The AVM proof system defines its own mechanism for public inputs in which it flags certain trace columns as "public input columns". Any public input columns must be communicated

in full to a verifier.

AVM public inputs structure

The VM circuit's I/O (`AvmPublicInputs`) is defined below:

```
AvmSessionInputs {  
    // Initializes Execution Environment  
    feePerL2Gas: field,  
    feePerDaGas: field,  
    globals: PublicGlobalVariables,  
    address: AztecAddress,  
    sender: AztecAddress,  
    contractCallDepth: field,  
    isStaticCall: boolean,  
    transactionFee: field,  
    // Initializes Machine State  
    l2GasLeft: field,  
    daGasLeft: field,  
}  
AvmSessionResults {  
    l2GasLeft: field,  
    daGasLeft: field,  
    reverted: boolean,  
}  
AvmSessionPublicInputs {  
    sessionInputs: AvmSessionInputs,  
    calldata: [field; MAX_CALLDATA_LENGTH],  
    worldStateAccessTrace: WorldStateAccessTrace,  
    accruedSubstate: AccruedSubstate,  
    sessionResults: AvmSessionResults,  
}
```

The `WorldStateAccessTrace` and `AccruedSubstate` types are defined in "["State"](#)". Their vectors are assigned constant/maximum lengths when used as

circuit inputs.

AVM public input columns

The `AvmPublicInputs` structure is represented in the VM trace via the following public input columns:

1. `sessionInputs` has a dedicated column and is used to initialize the initial call's `AvmContext.ExecutionEnvironment` and `AvmContext.MachineState`.
2. `calldata` occupies its own public input column as it is handled differently from the rest of the `ExecutionEnvironment`. It is used to initialize the initial call's `AvmContext.ExecutionEnvironment.calldata`.
 - Equivalence is enforced between this `calldata` column and the "input call pointer"'s memory. Through this mechanism, the initial call's `calldata` is placed in a region memory that can be referenced via the `CALLDATACOPY` instruction from within the initial call.
3. `worldStateAccessTrace` is a trace of all world state accesses. Each of its component vectors has a dedicated set of public input columns (a sub-table). An instruction that reads or writes world state must match a trace entry. The [trace type definition in the "State" section] lists, for each trace vector, the instruction that populate its entries.
4. `accruedSubstate` contains the final `AccruedSubstate`.
 - This includes the accrued substate of all *unreverted* sub-contexts.
 - Reverted substate is not present in the Circuit I/O as it does not require further validation/processing by downstream circuits.
5. `sessionResults` has a dedicated column and represents the core "results" of the AVM session processed by this circuit (remaining gas, reverted).

VM Control Flow + High-level architecture

This document breaks down the VM into internal components and defines how these components interact with one another.

This can be considered an intermediate abstraction layer between the high-level VM definition and the explicit circuit architecture. The intention is for this abstraction to be analogous to how CPU chips are broken down into discrete components.

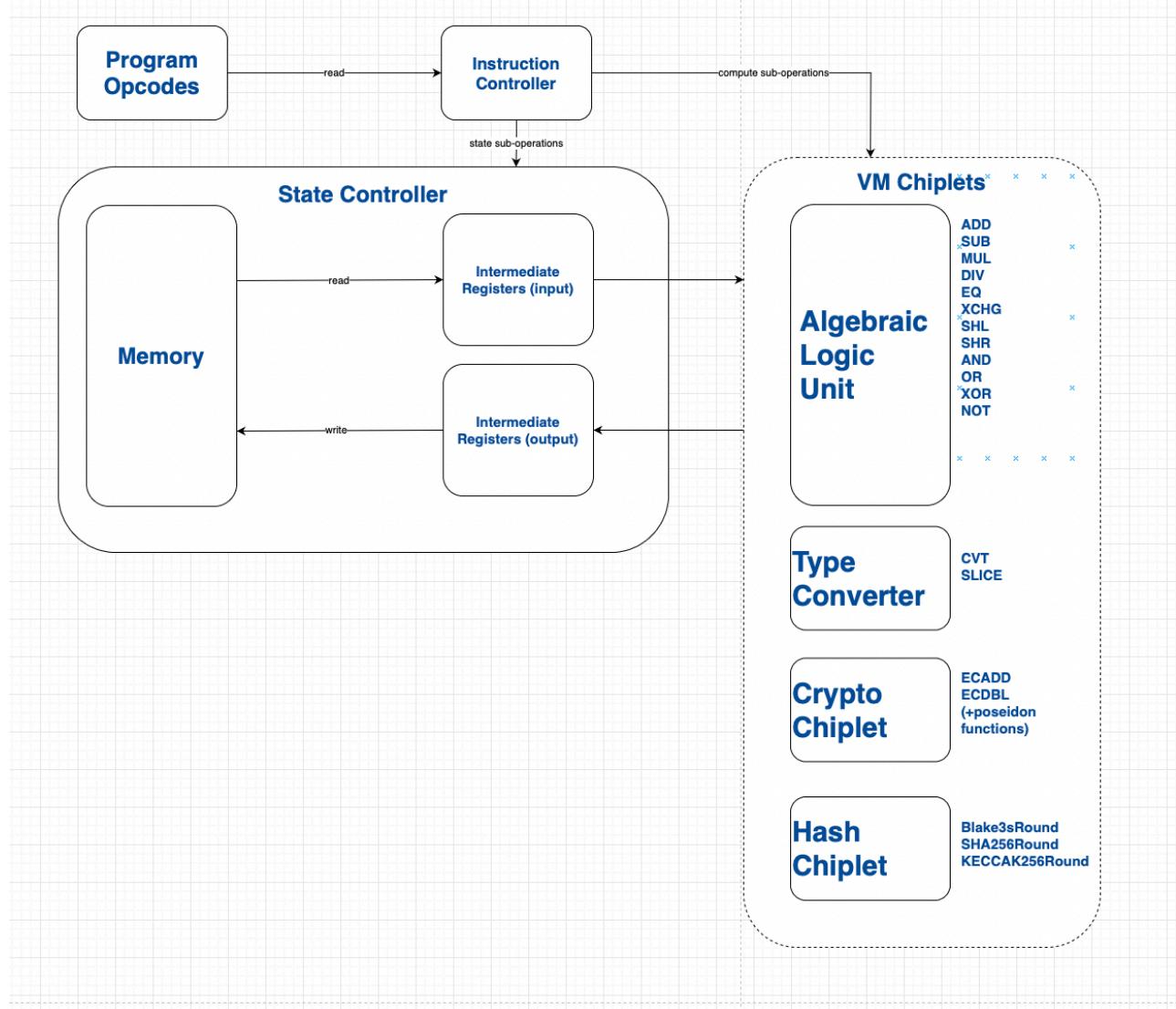
Sub-operations

Notation note: I use the term "clock cycle" in a way that is analogous to "row in the execution trace".

We wish to define a set of sub-operations our VM can execute. Multiple sub-operations can be executed per clock cycle. Each instruction in the instruction set exposed by the VM is composed of 1 or more sub-operations.

The intention is for sub-operations to be implementable as independent VM circuit relations.

Control flow



Notation note: whenever the VM "sends a signal" to one or more VM components, this is analogous to defining a boolean column in the execution trace that toggles on/off specific functionality

- The instruction controller uses a program counter to read the current opcode to be executed, and send signals to downstream components to execute the opcode
- The state controller reads data from memory into the intermediate registers,

using the opcode parameter values

- The VM "algebraic logic unit" executes the opcode given the intermediate register values, and writes output into the intermediate registers
- The state controller writes the output from an intermediate register into memory

Chiplets

This borrows the chiplet nomenclature from the Miden VM - these components encapsulate functionality that can be effectively defined via an independent sub-circuit (analog in real world = specific part of a CPU chip die)

Chiplets can be developed iteratively i.e. first draft of the AVM only needs a barebones algebraic logic unit.

We want apply the following design heuristic to chiplets: they are *loosely coupled* to other AVM components. It should be possible to remove a chiplet and the AVM opcodes it implements, without requiring any upstream changes to the AVM architecture/implementation

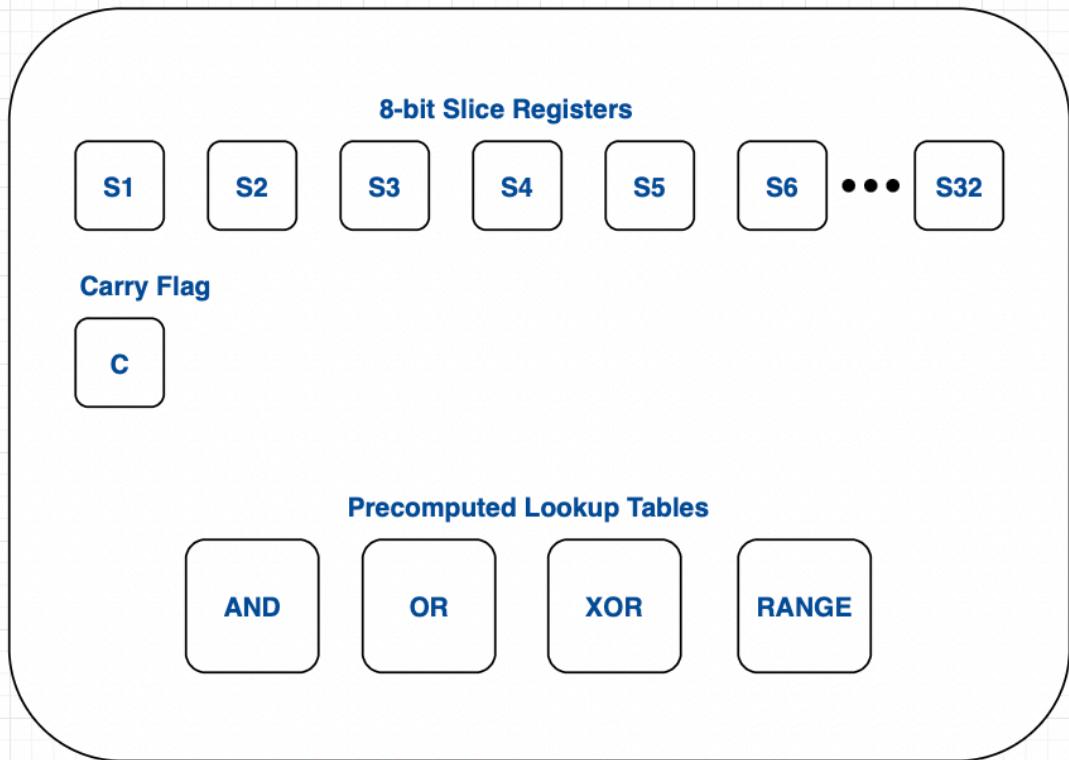
Algebraic Logic Unit

The algebraic logic unit performs operations analogous to an arithmetic unit in a CPU.

This component of the VM circuit evaluates both base-2 arithmetic operations and prime-field operation. It takes its input/output from the intermediate registers in the state controller.

The following block diagram maps out an draft of the internal components of the "ALU"

Algebraic Logic Unit internal structure



Notes:

For logic operations (e.g. AND/OR) we use lookup tables. The max. size of each lookup table cannot grow too large as the Prover pays a constant cost linear with the size of the lookup table.

To this end we use lookup tables for logic operations that take *8-bit input operands* for a total table size of 2^{16} .

i.e. the table contains the output for every possible 8-bit combination of 2 input operands.

Slice registers

We need to slice our inputs into 8-bit chunks for logic operations, in order to index the lookup tables.

As a simplification, we can say that *any* operation that requires range-constraints will split the input operands into 8-bit slices, as we can then apply consistent range-checking logic.

Carry flag

Used to test for overflows. If we want the high-level instruction set to have "add with carry" we need to expose the carry flag to the state controller.

Example operation: 32-bit ADD(a, b, c)

Assume we start with a in intermediate register $R1$, b in intermediate register $R2$, and c in intermediate register $R3$

1. Store the first 32 bits of $a + b$ in slice registers $s1, s2, s3, s4$, with the carry bit in carry
2. Validate $a + b = s_1 + 2^8 s_2 + 2^{16} s_3 + 2^{24} s_4 + 2^{32} \text{carry}$
3. Validate $c = s_1 + 2^8 s_2 + 2^{16} s_3 + 2^{24} s_4$

Bytecode Validation Circuit

Goal: Validate that a polynomial commitment to AVM program opcodes maps to the bytecode representation of an AVM program of maximum size n .

Definitions - Curves and Fields

The bytecode validation circuit is implemented over the BN254 elliptic curve with group elements defined via \mathbb{G}_{bn254} .

The field \mathbb{F} represents the finite field whose characteristic equals the number of points on the BN254 curve.

Bytecode representation

Each opcode in the AVM can be described by an integer in the range $[0, \dots, 256^{31}]$ (i.e. 31 bytes of data). All opcodes excluding SET require much less data than 31 bytes.

In the AVM circuit architecture, multiple columns are used to define a VM operation. These columns describe the following quantities:

1. the opcode to be executed (1 byte of data)
2. three parameter columns that define either literal values or memory indexes
3. three "flag" columns that define metadata associated with each parameter (e.g. whether the parameter a should be interpreted as a literal value or a memory index, or an indirect memory index)

To both minimize the amount of information used to *define* a given AVM program, the AVM possesses an additional column that describes the *packed* opcode. i.e. the integer concatenation of all 6 of the above column values. We define this column via

the vector of field elements $\mathbf{op} \in \mathbb{F}^n$ (where n is the number of opcodes in the program). \mathbf{op} is defined as the *column representation* of an AVM program.

Packed bytecode representation

When *broadcasting* the data for AVM programs, we desire an encoding that minimizes the raw number of bytes broadcast, we call this the *packed representation* of the program.

The number of bytes required to represent an element in \mathbf{op} in the AVM can be derived from the value of the 1st byte (e.g ADD requires 7 bytes of data - the ADD opcode (1 byte) and three memory indices (each of size 2 bytes)).

See (ref: TODO!) for a table that describes the amount of data required for each opcode.

Each field element in a BN254 circuit can represent 31 bytes of bytecode data. The packed representation of an AVM program $\mathbf{b} \in \mathbb{F}^n$ is defined as the concatenation of \mathbf{op} into 31-byte chunks, represented as field elements.

There exists a mapping function g that, given the packed representation \mathbf{b} , will produce the column representation \mathbf{op} .

$$g(\mathbf{b}) = \mathbf{op}$$

A full description of g is provided [further down in this document](#).

Committed representation

The committed representation of an AVM program is an elliptic curve polynomial commitment $[P] \in \mathbb{G}_{bn254}$, created via the KZG polynomial commitment scheme

(ref).

$[P]$ is a commitment to $P(X) \in \mathbb{F}[X]^n$ where $P(X) = \sum_{i=0}^{n-1} op_i X^i$

Bytecode validation logic

Given inputs $\mathbf{b} \in \mathbb{F}^n$ and $[P] \in \mathbb{G}_{bn254}$, we must validate that $[P] = \text{commit}_{KZG}(g(\mathbf{b}))$.

This requires the following *high level* steps:

1. For all $i \in [0, \dots, n - 1]$, validate that $b_i < 256^{31} - 1$
2. Compute $\mathbf{op} = g(\mathbf{b})$
3. Perform a *polynomial consistency check* between \mathbf{op} and $[P]$

Polynomial Consistency Check

The most straightforward way of validating \mathbf{op} , $[P]$ would be to directly construct $[P]$ from \mathbf{op} . We do not do this, as this would require a large multiscalar multiplication over the BN254 curve. This could only be performed efficiently over a Grumpkin SNARK circuit, which would add downstream complexity to the Aztec architecture (currently the only Grumpkin proofs being accumulated are elliptic-curve-virtual-machine circuits). The rollup circuit architecture already supports efficient recursive aggregation of BN254 proofs - the desire is for the bytecode validation circuit to be a canonical Honk SNARK over the BN254 field.

To perform a polynomial consistency check between \mathbf{op} and $[P]$, we perform the following:

1. Generate a challenge $z \in \mathbb{F}$ by computing the Poseidon hash of $H(op_0, \dots, op_{n-1}, [P])$
2. Compute $\sum_{i=0}^{n-1} op_i z^i = r \in \mathbb{F}$

- Validate via a KZG opening proof that $[P]$ commits to a polynomial $P(X)$ such that $P(z) = r$

In the same manner that Honk pairings can be deferred via aggregating pairing inputs into an accumulator, the pairing required to validate the KZG opening proof can also be deferred.

Evaluating the polynomial consistency check within a circuit

The direct computation of $r = \sum_{i=0}^{n-1} op_i z^i$ is trivial as the field is native to a BN254 SNARK circuit, and will require approx. 2 constraints per opcode.

Validating a KZG opening proof will require approx. 3 non-native elliptic curve scalar multiplications, which will have a cost of approx. 30,000 constraints if using `stdlib::biggroup` from the PLONK standard library.

The major cost of the consistency check is the Poseidon hash of the packed bytecode vector b and the commitment $[P]$ - this will incur approx. 22 constraints per element in b

Definition of mapping function g

The following is a pseudocode description of g , which can efficiently be described in a Honk circuit (i.e. no branches).

We define a function `slice(element, idx, length)`. `element` is a field element interpreted as a length-31 byte array. `slice` computes the byte array `element[idx] : element[idx + length]`, converts into a field element and returns it.

We define a size-256 lookup table `c` that maps an avm instruction byte to the byte

length required to represent its respective opcode.

```
g(b) {
    let i := 0; // index into bytecode array `b`
    let j := 0; // byte offset of current bytecode element
    let op := []; // vector of opcode values we need to populate
    for k in [0, n]:
    {
        let f := b[i];
        let instruction_byte := f.slice(j, 1);
        let opcode_length := c[instruction_byte];
        let bytes_remaining_in_f := 30 - j;
        let op_split := opcode_length > bytes_remaining_in_f;
        let bytes_from_f := op_split ? bytes_remaining_in_f :
        opcode_length;
        let op_hi := f.slice(j, bytes_from_f);

        let f' := b[i+1];
        let bytes_from_f' := opcode_length - bytes_from_f;
        let op_lo := f'.slice(0, bytes_in_f');

        op[k] := op_lo + (op_hi << (bytes_in_f' * 8));
        i := i + op_split;
        j := op_split ? bytes_in_f' : j + opcode_length;
    }
    return op;
}
```

Pseudocode definition of `slice` function constraints:

We define `pow(x)` to be a size-31 lookup table that maps an input $x \in [0, \dots, 31]$ into the value 2^{8x}

We require the Prover has computed witness field elements `f_lo`, `f_hi`, `result` that satisfy the following constraints:

```

slice(f, index, length)
{
    assert(f_hi < pow(index));
    assert(f_lo < pow(31 - index - length));
    assert(result < pow(length));
    assert(f == f_lo + result * pow(31 - index - length) + f_hi *
pow(31 - index));
    return result;
}

```

Evaluating `g` within a Honk circuit

The `g` function requires the contents of b be present via a lookup table. We can achieve this by instantiating elements of b via the ROM abstraction present in the Plonk standard library (table initialisation costs 2 constraints per element, table reads cost 2 constraints per element)

We can instantiate tables `c`, `pow` as lookup tables via the same mechanism.

The `slice` function requires 3 variable-length range checks. In Honk circuits we only can support fixed-length range checks.

The following pseudocode defines how a variable-length range check can be composed of fixed-length range checks. Here we assume we have previously constrained all inputs to be less than $2^{248} - 1$

```

less_than(a, b) {
    // this block is not constrained and defines witness
    gneeration
    let a_lo := a & (2^{124} - 1)
    let b_lo := b & (2^{124} - 1)

```

Each `slice` call requires three `less_than` calls, and each iteration of `g` requires 3 `slice` calls. In total this produces 36 size-124 range checks per iteration of `g`. Each size-124 range check requires approx. 5 constraints, producing 180 constraints of range checks per opcode processed.

A rough estimate of the total constraints per opcode processed by the `g` function would be 200 constraints per opcode.

Bytecode Validation Circuit Summary

The bytecode validation circuit takes, as public inputs, the packed bytecode array $\mathbf{b} \in \mathbb{F}$ and the bytecode commitment $[P] \in \mathbb{G}_{bn254}$ (represented via field elements).

The circuit evaluates the following:

1. For all $i \in [0, \dots, n - 1]$, validate that $b_i < 256^{31} - 1$
2. Compute $\mathbf{op} = g(\mathbf{b})$
3. Perform a *polynomial consistency check* between \mathbf{op} and $[P]$

Summary of main circuit costs

The polynomial consistency check requires a Poseidon hash that includes the packed bytecode array b . This requires approx. 22 Honk constraints per 31 bytes of bytecode.

The `g` function will cost approx. 200 constraints per opcode.

For a given length n , the approx. number of constraints required will be approx $222n$.

A circuit of size 2^{21} (2 million constraints) will be able to process a program containing approximately $n = 9,400$ steps. In contrast, a Solidity program can contain a maximum of 24kb of bytecode.

Note: unless the efficiency of the validation circuit can be improved by a factor of ~4×, it will not be possible to construct bytecode validation proofs client-side in a web browser. Delegating proof construction to a 3rd party would be acceptable in this context because the 3rd party is untrusted and no secret information is leaked.

Type Definitions

This section lists type definitions relevant to AVM State and Circuit I/O.

TracedContractCall

| Field | Type | Description |
|--------------------------|--------------------|---|
| <code>callPointer</code> | <code>field</code> | The call pointer assigned to this call. |
| <code>address</code> | <code>field</code> | The called contract address. |
| <code>counter</code> | <code>field</code> | When did this occur relative to other world state accesses. |
| <code>endLifetime</code> | <code>field</code> | End lifetime of a call. Final <code>accessCounter</code> for reverted calls, <code>endLifetime</code> of parent for successful calls. Successful initial/top-level calls have infinite (max-value) <code>endLifetime</code> . |

TracedL1ToL2MessageCheck

| Field | Type | Description |
|--------------------------|--------------------|---|
| <code>callPointer</code> | <code>field</code> | Associates this item with a <code>TracedContractCall</code> entry in <code>worldStateAccessTrace.contractCalls</code> |
| <code>leafIndex</code> | <code>field</code> | |

| Field | Type | Description |
|--------------------------|--------------------|---|
| <code>msgHash</code> | <code>field</code> | The message hash which is also the tree leaf value. |
| <code>exists</code> | <code>field</code> | |
| <code>endLifetime</code> | <code>field</code> | Equivalent to <code>endLifetime</code> of the containing contract call. |

TracedStorageRead

| Field | Type | Description |
|--------------------------|--------------------|---|
| <code>callPointer</code> | <code>field</code> | Associates this item with a <code>TracedContractCall</code> entry in <code>worldStateAccessTrace.contractCalls</code> |
| <code>slot</code> | <code>field</code> | |
| <code>exists</code> | <code>field</code> | Whether this slot has ever been previously written |
| <code>value</code> | <code>field</code> | |
| <code>counter</code> | <code>field</code> | |
| <code>endLifetime</code> | <code>field</code> | Equivalent to <code>endLifetime</code> of the containing contract call. The last <code>counter</code> at which this read/write should be considered to "exist" if this call or a parent reverted. |

TracedStorageWrite

| Field | Type | Description |
|-------------|-------|---|
| callPointer | field | Associates this item with a TracedContractCall entry in worldStateAccessTrace.contractCalls |
| slot | field | |
| value | field | |
| counter | field | |
| endLifetime | field | Equivalent to endLifetime of the containing contract call. The last counter at which this read/write should be considered to "exist" if this call or a parent reverted. |

TracedNoteHashCheck

| Field | Type | Description |
|-------------|-------|---|
| callPointer | field | Associates this item with a TracedContractCall entry in worldStateAccessTrace.contractCalls |
| leafIndex | field | |
| noteHash | field | unsiloed |

| Field | Type | Description |
|--------------------------|--------------------|---|
| <code>exists</code> | <code>field</code> | |
| <code>counter</code> | <code>field</code> | |
| <code>endLifetime</code> | <code>field</code> | Equivalent to <code>endLifetime</code> of the containing contract call. |

TracedNoteHash

| Field | Type | Description |
|--------------------------|--------------------|---|
| <code>callPointer</code> | <code>field</code> | Associates this item with a <code>TracedContractCall</code> entry in <code>worldStateAccessTrace.contractCalls</code> |
| <code>noteHash</code> | <code>field</code> | |
| <code>counter</code> | <code>field</code> | |
| <code>endLifetime</code> | <code>field</code> | Equivalent to <code>endLifetime</code> of the containing contract call. The last <code>counter</code> at which this object should be considered to "exist" if this call or a parent reverted. |

Note: `value` here is not siloed by contract address nor is it made unique with a nonce. Note hashes are siloed and made unique by the public kernel.

TracedNullifierCheck

| Field | Type | Description |
|-------------|-------|---|
| callPointer | field | Associates this item with a TracedContractCall entry in worldStateAccessTrace.contractCalls |
| nullifier | field | unsiloed |
| exists | field | |
| counter | field | |
| endLifetime | field | Equivalent to endLifetime of the containing contract call. |

TracedNullifier

| Field | Type | Description |
|-------------|-------|---|
| callPointer | field | Associates this item with a TracedContractCall entry in worldStateAccessTrace.contractCalls |
| nullifier | field | |
| counter | field | |
| endLifetime | field | Equivalent to endLifetime of the containing contract |

| Field | Type | Description |
|-------|------|---|
| | | call. The last <code>counter</code> at which this object should be considered to "exist" if this call or a parent reverted. |

TracedArchiveLeafCheck

| Field | Type | Description |
|------------------------|--------------------|-------------|
| <code>leafIndex</code> | <code>field</code> | |
| <code>leaf</code> | <code>field</code> | |

UnencryptedLog

| Field | Type | Description |
|----------------------|--|--|
| <code>address</code> | <code>AztecAddress</code> | Contract address that emitted the log. |
| <code>log</code> | <code>[field;
MAX_UNENCRYPTED_LOG_LENGTH]</code> | |

SentL2ToL1Message

| Field | Type | Description |
|----------------------|---------------------------|---|
| <code>address</code> | <code>AztecAddress</code> | L2 contract address that emitted the message. |

| Field | Type | Description |
|-----------|------------|---|
| recipient | EthAddress | L1 contract address to send the message to. |
| content | field | Message content. |