Contract Creation

Disclaimer Disclaimer We are building Aztec as transparently as we can. The documents published here are living documents. The protocol, sandbox, language, and tools are all subject to change over time.

Please seehere for details of known Aztec protocol and Aztec Sandbox limitations.

If you would like to help us build Aztec:

- Contribute code onGitHub
- ; or
- Join inforum
- discussions.

Objectives

Answer:

- How are contracts represented in Aztec?
- How does a transaction deploy a contract?
- How do we validate contract bytecode

What is a contract?

A contract is a collection of functions that operate on shared state.

Functions can either be public (executed/proved by rollup providers with no hidden state) or private (executed/proved locally by transaction sender, can have hidden state).

Functions that write to the updatable state tree must be public; the rollup provider must perform these updates as only they know the current value of the state tree (otherwise you get race conditions).

What defines a function?

A function maps afunction signature to averification key . The vkey contains all information required for a kernel circuit to verify a proof of knowledge of the function's execution.

Function Tree

Each contract's function data is stored in a Merkle tree, where each leaf contains a hash of the following data:

1. Function Selectors
2. bool isPrivate
3. flag
4. Verification Key Hash

How are function signatures defined?

We can take a leaf from Ethereum and make them the first 4 bytes of a hash of the function definition (defined according to the contract artifact, TBD).

Contract Representation in Aztec

Aztec contains a contract Merkle tree that stores all contract data (the contract tree).

Each leaf represents a contract, and contains a hash of the following:

1. Contract Address
2. Portal Address
3. Function Tree Root
4. Constructor Hash

TheportalAddress is a unique identifier that describes an L1 portal contract that is linked to the contract. All L1 calls must be made from/sent to the portal contract.

The constructor hash is a hash of function data that defines the constructor (function signature,isPrivate flag, vkey hash).

Contract Representation on L1

L1 Portal Contract

Referring to it as an "L1 Portal" Contract is pleonastic - a portal contract canonly be deployed to L1 by its definition, so saying "Portal Contract" is fine. Each L2 contract will have its own Portal Contract deployed to L1, so that the contract may make calls to L1 (TODO link modern spec+examples).

L2 contracts can only call L1 contracts via its designated portal contract.

L2 contracts can only be calledfrom L1 via calls from the portal contract.

The goal is to simplify the domain knowledge required for Aztec devs - all comms that go across the L2-L1 boundary must go via an L2 contract's respective portal contract.

Portal contracts also act as custodians for tokens while they are bridged into their respective L2 contract.

Deploying Portal Contracts

Portal contracts are deployed prior to the deployment of the respective L2 contract.

When deploying an L2 contract, its associated portal contract address is provided as an input parameter.

Linking Portal Contracts

It is important that each L2 contract has a distinct portal contract.

It is important that all calls coming from Aztec into a portal contract can be trusted (i.e. "this call came from my linked L2 contract").

How do we prevent malicious actors spoofing calls?

E.g.

1. portal::A
2. is deployed. L2 Contract,L2::A
3. , is deployed and linked toportal::A
4. Adversary createsL2::B
5. and links toportal::A
6. Adversary sends call toL2::B
7. which then pingsportal::A

We can't solve this nullifying portal addresses when contracts are created, as this creates a race condition + griefing attack. For example and adversary sees a portal contract deployed + L2 tx in L2 tx pool to create corresponding L2 contract. They create a malicious L2 contract that links to the portal contract and post a higher gas fee, so they go first.

Potential Solutions

1: Free for all - any L2 contract can link to any L1 contract. However, when Aztec pings portal contracts, the address of the calling L2 contract is passed in as 1st parameter. Mildly annoying as portal logic has to check this 'sender' parameter to determine if call is legitimate.

2: When Aztec deploys L2 contracts, it pings portal contract with afunction Portal::linkRequest(address l2Contract) public returns (bool) . If return value isfalse or call throws, L2 deployment is unwound. This is a pain to implement + state unwinding is complex

3: When portal contract is deployed, it sends a link message to Aztec contract, that contains the deployer's address (this message gets added to the data tree). When a contract is deployed on L2, the data tree message is retrieved and a signature from the deployer address is required.

Option 1 is the simplest from an implementation point of view. Option 3 might be the safest.

Decision: Use Option 3

Deploying Contracts on L2

Data required for deployment

The following data is required to process a deployment and must also be broadcast to a data availability

solution.ContractDeploymentData contains the following:

1. Contents of the contract leaf being added to the tree
2. Contents of the constructor function
3. Contents of the leaves that define the function tree
4. ACIR opcodes for public functions
5. Address of portal contract

In addition, for public functions the protocol needs to validate the provided ACIR opcodes map to a given verification key. This is to prevent griefing sequencers/provers, by requesting they compute a public function proof but with bad ACIR opcode data; they will only notice this on generating an invalid proof.

Processing Contract Deployments

Deploying contracts is a distinct action that can only be instigated by a user.

Functions cannot deploy contracts and contracts cannot deploy contracts.

This limitation is due to the fact that contracts present a large data payload . Every "output" of a function call must be allocated as a public input to the function circuit as defined in the kernel specification (TODO: link/write).

Large contract deployments would require the specification allocates many public inputs to represent the ACIR bytecode of the deployed contract (e.g. 24kb contract = 768 public inputs). Each function circuit must have a fixed number of public inputs as defined by the kernel spec, which makes it impossible to efficiently output the large amounts of data required to deploy contracts.

Bytecode validation proofs

In the VM model, the only part of a verification key that depends on the contract's ACIR opcodes is a BN254 commitment to the opcodes.

It is relatively straightforward to create a grumpkin circuit that will do the following:

1. Take in ACIR bytecode as public inputs
2. Take the BN254 commitment as a public input
3. Perform native ecc multi-exponentiation to reproduce the commitment, validate it matches the provided input

This proof has a large number of public inputs due to the need to pass in ACIR code.

This proof is not processed by user-generated kernel circuits because of this; it will be processed/verified as part of the rollup circuit.

L2 Contract Address

The contract address is calculated by the contract deployer, deterministically, as:

- contractAddress = hash(deployerAddress, salt, functionTreeRoot, constructorHash)

The EVM's CREATE2 doescontractAddress = hash(0xff, deployerAddress, salt, keccak(bytecode)) - we've taken this as inspiration. * deployerAddress * is included to prevent frontrunning of deployment requests, but it does reveal who is deploying the contract. To remain private a user would have to use a burner address, or deploy a contractthrough * a private contract which can deploy contracts. * **?** Why does CREATE2 include a deployerAddress? * **!** So that contracts can deploy contracts to deterministic addresses. The original goal was to enable pre-funding contracts before they were deployed. Not v. relevant for us though * salt * gives the deployer some 'choice' over the eventual contract address; they can loop through salts until they find an address they like. * functionTreeRoot * is like the bytecode without constructors or constructor arguments. This allows people to validate the functions of the contract. * constructorHash = hash(privateConstructorPublicInputsHash, publicConstructorPublicInputsHash, privateConstructorVKHash, publicConstructorVKHash) * - this allows people to validate the initial states of the contract. (Note: this is similar to how thebytecode * in create2 includes an encoding of the constructor arguments).

To prevent duplicate contract addresses existing, a 'nullifier' is submitted when each new contract is deployed.newContractAddressNullifier = hash(newContractAddress) .

In order to link acontractAddress , with a leafIndex in thecontractTree , we reserve the storageSlot0 of each contract's public data tree storage to store that leafIndex.

ThecontractAddress is stored within the contract's leaf of thecontractTree , so that the private kernel circuit may validate contract address <--> vk relationships.

Aside: It would have been neat for a contract's address to be the leaf index of its mini Merkle tree (i.e. the root of thevkTree) in thecontractTree . However, the leaf index is only known at the time the rollup provider runs the 'contract deployment kernel snark', whereas we need the contract address to be known earlier, at the time the deployer generates the private constructor proof. Without a contract address, the private constructor would not be able to: call other functions (and pass them a valid callContext); or silo newly-created commitments.

Deployment overview

When the user has finished creating their contract deployment proofs, the expected output is proofs of knowledge over 2 circuits:

1. A private kernel snark proof where the private call stack is empty
2. A set of contract validation proofs: one per deployed contract

A sequencer can validate the legitimacy of the tx, by extracting the acir commitment to each contract from the kernel snark public inputs, validate they match the inputs to the validation proofs and validate the validation proofs are correct.

We cannot use gas metering to allow for valid kernel proofs but invalid validation proofs. This is because there is nothing stopping a malicious prover from substituting valid proofs for invalid ones (thus preventing the contract from being deployed but taking the gas fee from the kernel proof). Contract validation proofs are checked in the rollup circuit and we must assert that if contracts are being deployed, the rollup circuit is only valid if correct validation proofs are supplied.

On further thought wecould allow this, if in the kernel circuit, avalidation proof hash is provided. This ensures the tx sender fixes the validation proofs being used in the rollup circuit, but if the proofs are incorrect the rollup circuit can still be computed and fees paid. Open Q: is it worth doing this?

Kernel Circuit Logic

We modify the private function call stack to also supportcontract deployments .

Each call stack object contains the following additional data parameters:

1. bool isContractDeployment
2. Option contractDataHash
3. Option vkRoot
4. Option constructorHash

If the function call is a deployment and the contract has a constructor, the constructor function data fills the regular function call stack parameters.

When processing a private fn call, usually the contract address + fn selector is used to recover a verification key object from the contract tree.

IfisContractDeployment == true this check isbypassed . Instead:

- The contract address is derived and is used as the call context for the constructor function.
- The{ contractDataHash, vkRoot, constructorHash, contractAddress }
- are pushed onto adeployedContracts[]
- dynamic array.

ThedeployedContracts array has a maximum sizeMAX_NUM_DEPLOYED_CONTRACTS . It is anoutput of the private kernel snark.

Excluding these steps, constructor function call is executed identically to a regular private function.

Under this design, if a public constructor is desired, one must create a private constructor function that then calls a public function. This could be abstracted away by the aztec-nargo compiler. Q: why can't deployments be part of the public fn callstack?

A: When the sequencer receives a private kernel proof, they must be able to determine:

1. How many contracts are being deployed
2. Whether each deployment comes with a valid deployment validation proof (seeRollupCircuitLogic
3.)

Q: why not have deployments use a separate 'deployment stack'?

A: Function execution logic isexpensive . Ideally we only evaluate it once per Kernel circuit.

i.e. we don't implement "process constructor fn if it exists"and "process private fn if it exists"

Instead the constructor function is part of the private fn stack and we perform conditional logic to see if the fn is a regular private fn call or the constructor fn from a contract deployment.

Public Kernel Logic

The public kernel snark propagates thedeployedContracts[] array as a circuit output

Rollup Circuit Logic

For each kernel proof, iterate overdeployedContracts[] . For each entry:

Validation

(TODO: this doesn't quite work - we need to do the ACIR validation step for every public function in a contract!)

(replacedeployedContracts[] withdeployedContract anddeployedPublicFunctions[])

(TODO: flesh out contract validation - make the entire process (incl. looping overdeployedPublicFunction[]) a single circuit).

1. UnwrapcontractDataHash
2. , extractcontractData.verificationKey
3. validatecontractData.verificationKey.vkRoot == vkRoot
4. extractverificationKey.acirCommitment
5. Validate a circuit validation proof that takes in a sequence of ACIR opcodes (provided by rollup prover) and theacirCommitment
6. (and validates both are correct)
7. EmitcontractData
8. andAcirOpcodes
9. as public inputs
10. extractverificationKey.constructor
11. and validate the hash of the verification key matchescontractData.constructorVkHash

Creation

The contract data payload is generated from the above (including contract address) and added into the contract tree.

TheportalId is derived by the rollup circuit (i.e. rollup trackssmallestUnusedPortalId , wherecontract.portalId == smallestUnusedPortalId andsmallestUnusedPortalId += 1).

TODO: replaceportalId withportalAddress (n.b. needs to be unique!)

Rollup Contract Logic

The rollup contract extracts the deployed contracts from public inputs.

For each new contract, theportalId is mapped to the L1 portal contract address. (e.g.mapping(uint32 => address) portalContracts)

Distributing L2 contract data

ACIR opcodes for public functions must be avoidable on a data availability solution, as their knowledge is required by rollup providers to create valid public function proofs.

This is not required for private functions, it is left to developers to ensure their contract data is distributed and disseminated (e.g. github).

Under the current spec it is not possible to create contracts with fully hidden bytecode, unless these contracts have no public functions.

(a future update could enable completely hidden contracts by having a distinct contract public state tree (vs the current monolithic updatable state tree). The contract's state tree root would be part of the append-only data tree. This way its "public" function proofs could be executed by 3rd parties instead of the rollup provers, without creating race conditions)

(following pasted from github)

Contract Deployment

A contract is a collection of functions and state variables. Each 'function' is expressed as a circuit, and each circuit can be represented by its verification key. I.e. each verification key represents a callable function in the smart contract.

The set of functions of a contract is represented as a mini Merkle tree of verification keys - avkTree - with the root node - vkRoot - being a leaf in thecontractTree .

Deployment topics:

- Constructor functions (to populate initial state variables).
- Specifying an L2 contract address
- Distributing L2 contract data
- Linking to an L1 Portal Contract

These topics are reflected in the layout of the contract deployment artifact:

publicInputs

{ // Constructor functions privateConstructorPublicInputsHash , publicConstructorPublicInputsHash , privateConstructorVKHash , publicConstructorVKHash ,

// L2 contract address (create2-like) contractAddress , vkRoot ,

// Distributing L2 contract data circuitDataKeccakHash ,

// Linking to an L1 Portal Contract portalContractAddress , } ; Note: the distribution of L2 data on-chain is optional and can be done by submitting a compression of the ACIR representation of a circuit as calldata.

Constructor functions

Constructor functions can be called when deploying a contract, to populate the contract with some initial state variables. A private constructor can be called to populate private states, and a public constructor can be called to populate public sates. TODO: we might be able to get away with a single constructor.

The contract deployment kernel circuit verifies the constructors' executions.

Private constructor

A private constructor is only needed if we'd like private states to exist at the beginning of our contract's life. Since private states must be privateto a person (i.e.owned by a person), such states would only be created at deployment if we want thedeployer to own something privately.

An example might be if someone was creating a completely private cryptocurrency directly on aztec's L2, and they minted the total supply at once for themselves (to distribute themselves). Then initially, the deployer of the contract might be in control of a single 'value note' representing the entire initial supply. They could then distribute value notes thereafter by running a private circuit for 'transferring value' multiple times. As an extension of this example, the deployer might wish bake-into the private constructor the token distribution logic. Then with the private constructor, they could distribute value immediately and privately to a group of people by creating up-to 64 commitments (actual value TBD).

Public constructor

Adds initial public state variables to the public data tree.

Further reading

To see how to deploy a contract in practice, check out the dapp development tutorial . Edit this page

Previous Smart Contracts Next Private - Public execution