Shoutout to @spalladino for the idea to remove the contracts tree, and to adopt a “classes & instances” approach to contracts.

Shoutout to Starkware for lovely ideas relating to contract classes & instances.

This proposes quite a drastic change to contract deployment on Aztec.

One of my goals is to simplify the core protocol, so whenever I need to coin a new concept, I make sure to include the word “abstraction” in there.

See the end for pros & cons.

tl;dr

- Separate contracts into contract classes and contract instances.
- A contract class is roughly bytecode / function data / vk data.
- A contract instance establishes a contract address, and storage space for the contract.
- Classes are “declared”.
- Instances are “deployed”.
- A contract class is roughly bytecode / function data / vk data.
- A contract instance establishes a contract address, and storage space for the contract.
- Classes are “declared”.
- Instances are “deployed”.
- Remove the contracts tree.
- Piggy-back on the nullifier tree instead when declaring contract classes and deploying new instances.
- Piggy-back on the nullifier tree instead when declaring contract classes and deploying new instances.
- Contract class declaration logic and contract instance deployment logic is moved from the kernel circuit into an app.
- This enables us to piggy-back on existing nullifier and event functionality.
- It also enables a constructor function to be called by another contract.
- That is, it enables “contracts deploying other contracts”.
- That is, it enables “contracts deploying other contracts”.
- This enables us to piggy-back on existing nullifier and event functionality.
- It also enables a constructor function to be called by another contract.
- That is, it enables “contracts deploying other contracts”.
- That is, it enables “contracts deploying other contracts”.
- When declaring a class:
- A nullifier is emitted, which captures all of the data about the class.
- Data relating to the contract class is broadcast to the network via a conventional event.
- A nullifier is emitted, which captures all of the data about the class.
- Data relating to the contract class is broadcast to the network via a conventional event.
- When deploying an instance:
- A nullifier is emitted, which captures the underlying class, the constructor args, the deployer, the new contract address, (and various public keys, pending further keys discussion).
- The contract address could even be

this nullifier (pending further discussions relating to keys).

- The contract address could even be

this nullifier (pending further discussions relating to keys).

- Data relating to the new contract instance is broadcast to the network via a conventional event.
- A nullifier is emitted, which captures the underlying class, the constructor args, the deployer, the new contract address, (and various public keys, pending further keys discussion).
- The contract address could even be

this nullifier (pending further discussions relating to keys).

- The contract address could even be

this nullifier (pending further discussions relating to keys).

- Data relating to the new contract instance is broadcast to the network via a conventional event.
- When executing a private function:
- The function selector and vk are looked-up from the function tree, (which is embedded within the contract class id, which is embedded within the contract instance, which is embedded within the contract instance's nullifier (which was emitted at deployment)).
- This is just a merkle membership proof of a leaf in the nullifier tree.
- This is just a merkle membership proof of a leaf in the nullifier tree.
- The function selector and vk are looked-up from the function tree, (which is embedded within the contract class id, which is embedded within the contract instance, which is embedded within the contract instance's nullifier (which was emitted at deployment)).
- This is just a merkle membership proof of a leaf in the nullifier tree.
- This is just a merkle membership proof of a leaf in the nullifier tree.
- When executing a public function / unconstrained function:
- The bytecode is looked-up from the contract class id, which is embedded within the contract instance, which is embedded within the contract instance's nullifier (which was emitted at deployment).
- This is just a merkle membership proof of a leaf in the nullifier tree.
- This is just a merkle membership proof of a leaf in the nullifier tree.
- The bytecode is looked-up from the contract class id, which is embedded within the contract instance, which is embedded within the contract instance's nullifier (which was emitted at deployment).
- This is just a merkle membership proof of a leaf in the nullifier tree.
- This is just a merkle membership proof of a leaf in the nullifier tree.

tl;dr tl;dr

"Let's put lots of contract deployment logic in an app contract, to remove stuff from the kernel circuit. Let's emit contract data as nullifiers and events. We get 'contracts deploying other contracts' functionality.".

Contract class

A contract_class_id

would be derived in a way which looks something like this:

[

image

1920×645 83.8 KB

](https://europe1.discourse-cdn.com/business20/uploads/aztec/original/1X/3c487450426ea00630cc88200f9bf765f2072b8d.jpeg)

Note: this diagram is illustrative, but even in writing this post, I've realised it's missing some things, or needs to be rearranged.

I.O.U. some updated diagrams.

A contract class id contains the following (most of which are self-explanatory):

- class_version
- declarer_address
- artifact_hash
- a hash of the abi/artifact which is spat out by noir.
- Must include:
- The version of nargo that's been used to generate the bytecode.
- Bytecode (including that of unconstrained functions).
- The version of nargo that's been used to generate the bytecode.
- Bytecode (including that of unconstrained functions).
- Must include:
- The version of nargo that's been used to generate the bytecode.
- Bytecode (including that of unconstrained functions).
- The version of nargo that's been used to generate the bytecode.
- Bytecode (including that of unconstrained functions).
- Missing from the diagram:
- Information about the repo / commit hash / tag / version / type of the proving system used to generate the VKs.
- Information about the repo / commit hash / tag / version / type of the proving system used to generate the VKs.
- Constructor info:
- constructor_function_selector
- constructor_vk_hash
- Suggestion: move this info to instead be the 0-th leaf of the function tree.
- constructor_function_selector
- constructor_vk_hash
- Suggestion: move this info to instead be the 0-th leaf of the function tree.
- Private function info is encoded in a function tree:
- (Recall, private functions are standalone circuits, because we can't afford a private vm).
- Each leaf contains info about a private function:
- function_selector
- booleans relating to the nature of the function
- vk_hash
- function_salt
- to salt a function's bytecode.
- Missing from the diagram: a hash of the acir PLUS unconstrained bytecode.
- function_selector

- booleans relating to the nature of the function
- vk_hash
- function_salt
- to salt a function’s bytecode.
- Missing from the diagram: a hash of the acir PLUS unconstrained bytecode.
- (Recall, private functions are standalone circuits, because we can’t afford a private vm).
- Each leaf contains info about a private function:
- function_selector
- booleans relating to the nature of the function
- vk_hash
- function_salt
- to salt a function’s bytecode.
- Missing from the diagram: a hash of the acir PLUS unconstrained bytecode.
- function_selector
- booleans relating to the nature of the function
- vk_hash
- function_salt
- to salt a function’s bytecode.
- Missing from the diagram: a hash of the acir PLUS unconstrained bytecode.
- Public function info is encoded, but it depends on a few things which haven’t-yet been specced:
- If selectors aren’t enshrined, then we just need a hash of the public bytecode to be included.
- If selectors are enshrined, public functions can be encoded in a similar way to private functions in the function tree.
- Missing from the diagram: a hash of the avm bytecode PLUS unconstrained bytecode.
- If selectors aren’t enshrined, then we just need a hash of the public bytecode to be included.
- If selectors are enshrined, public functions can be encoded in a similar way to private functions in the function tree.
- Missing from the diagram: a hash of the avm bytecode PLUS unconstrained bytecode.
- Unconstrained functions’ info can likely be included the same way as public function info.
- portal_contract_bytecode_hash
- an L2 contract is developed with exact corresponding portal contract bytecode in mind.

Public function encoding is a big question mark for my brain, whilst it’s all being figured out by the public vm team.

Contract Instance

A contract_address

would be derived in a way which looks something like this:

[

image

2880×724 190 KB

](https://europe1.discourse-

cdn.com/business20/uploads/aztec/original/1X/434a8ea9b1916a749e930c45ccbddc15b94a40a3.png)

Note: this diagram is illustrative.

A contract instance (and contract_address

) contains the following (most of which are self-explanatory):

- deployer_address
- deployment_salt
- contract_class_id
- portal_contract_address
- the actually-deployed portal contract address. A check needs to happen to ensure the bytecode of this L1 contract matches the portal_contract_bytecode_hash

contained within the contract_class_id

.

- constructor_args_hash
- Info about the public keys for this contract.

Why classes & instances?

- It reduces contract deployment costs, in cases where a class has already been deployed.
- It's a neater separation of concerns.
- See also: Contract classes, upgrades, and default accounts
- See also: Implementing contract upgrades

Why remove the contract tree?

- We already have a tree (the nullifier tree) which can contain all this info.
- It's one less tree to manage within the circuits and within an aztec node.

What's wrong with our current approach to contract deployment?

- Lots of (usually unused) contract deployment logic is baked into the initial private kernel circuit.
- A contract cannot deploy another contract (which is particularly strange in an account-abstraction world where users are meant to be represented by an account contract).

Contract deployment abstraction

Instead of baking most contract deployment logic into the Initial Private Kernel Circuit, we have a standalone smart contract (an app) to contain:

- contract class declaration logic;
- avm opcode commitment validation logic;
- contract instance deployment logic.

The kernel circuits (and other core circuits) contain less (no?) contract-deployment related logic. But they would still contain function lookup logic at the time of execution.

Contract class declaration logic:

The logic would live in a smart contract:

Basically:

- Emit the contract_class_id

as a nullifier, and its underlying data as an unencrypted event. (Events are designed to be an arbitrary length and submitted to L1).

Pseudocode:

fn declare_new_contract_class(contract_class_data: ContractClassData,) { // This data might not align with the diagram above. // Don't worry about it. It's all illustrative. const { class_version, declarer_address, artifact_hash, constructor_function_selector, constructor_vk_hash, function_tree_root, all_function_leaf_hashes, all_public_function_leaf_preimages, optional_non_public_function_leaf_preimages, all_public_bytecode, optional_non_public_bytecode, portal_contract_bytecode_hash, } = contract_class_data;

```
assert(class_version == 1); // An example of some hard-coded check that can be done.
                            // In a world of contract deployment abstraction, the class_version
                            // could maybe even be the deploying contract's address...?
assert(declarer_address == context.this_address);

// Compute the contract_class_id:
const contract_class_id = hash(
    class_version,
    // declarer_address, // Oooh, this isn't needed, because the kernel siloes every nullifier!
    artifact_hash,
    constructor_function_selector,
    constructor_vk_hash,
    function_tree_root,
    portal_contract_bytecode_hash,
);

// ---

// Deploy the class, as a nullifier:
context.push_new_nullifier(contract_class_id);

// Emit contract bytecode (etc) as an event:
context.emit_unencrypted_event(
    "New Contract Class",
    contract_class_data, // loads of data: it all gets sha256-hashed behind the scenes.
)

// ---

// We could call a bytecode commitment circuit as follows...
const bytecode_commitment_contract_address = 0x...;
const bytecode_commitment_function_selector = 0x12345678;

let i = 0;
for (public_function_leaf_preimage in all_public_function_leaf_preimages) {
    // Pass the purported avm_opcodes_commitment, and the bytecode.
    // Validate that the commitment is correct.
    const result = context.call(
        bytecode_commitment_contract_address,
        bytecode_commitment_function_selector,
        public_function_leaf_preimage.avm_opcodes_commitment,
        all_public_bytecode[i];
    );
    assert(result == true);
    ++i;
}

}
```

Contract instance deployment logic:

The logic would live in a smart contract:

Basically:

- Emit the contract_address

as a nullifier, and its underlying data as an unencrypted event. (Events are designed to be an arbitrary length and submitted to L1).

Pseudocode:

fn deploy_new_contract_instance(contract_instance_data: ContractInstanceData,) { // This data might not align with the diagram above. // Don't worry about it. It's all illustrative. const { deployment_salt, constructor_args, contract_class_id,

```
portal_contract_address, portal_contract_bytecode_hash, keys_hash, } = contract_instance_data;

const deployer_address = context.this_address;
const constructor_args_hash = hash(constructor_args);

// Make an L1->L2 call (before executing this function)
// to validate that the bytecode of the L1 contract has actually
// been deployed at the purported portal_contract_address,
// using the portal_contract_bytecode_hash contained within
// the contract_class_id.
// Consumption of this L1->L2 message read is NOT SHOWN HERE.

const contract_deployment_hash = hash(
    deployer_address,
    deployment_salt,
    constructor_args_hash,
    contract_class_id,
    portal_contract_address,
);

// ---

const contract_address = hash(contract_deployment_hash, keys_hash);

// Deploy the new contract address, as a nullifier:
// TODO: use the siloed nullifier _as_ the contract address, instead?
context.push_new_nullifier(contract_address);

// Usually the kernel circuit will check that a function exists
// in some already-deployed function, before executing it.
// Here the check would need to be modified to look at
// pending nullifiers (to find the newly-emitted contract address nullifier.
// Perhaps a `constructor_call` function is needed? Although I hope not.
context.call(contract_class_id, contract_address, constructor_function_selector, constructor_args);

context.emit_unencrypted_event(
    "New contract instance",
    deployer_address,
    deployment_salt,
    constructor_args,
    contract_class_id,
    portal_contract_address,
);

}
```

A bootstrapping problem

If the code for deploying a smart contract lives in a smart contract, how can we deploy that smart contract?

I guess we'd have to make a version of this 'deployment' smart contract as a special precompile, built into the genesis block.

Pros & Cons

Pros:

- It's cool.
- Less contract deployment logic in core protocol circuits. (I think… this would need to be validated).
- Removes the contract tree, so less code to maintain and audit.
- It enables contracts to call other contracts.
- It might enable contract deployment logic to be more-easily updated in future (although this is debatable, because at the time of execution, the kernel would still need to support both old and new execution paths).

Cons

- Overloads nullifiers and events.
- It's basically still "core protocol" in that we'll likely need to ensure a version is deployed in the genesis block. (See the bootstrapping section above).
- If there's a bug in one of these smart contracts, it would be hard to deploy a new, replacement 'contract deployment' contract (because it would only be deployable with this buggy smart contract). (That is, unless this contract has some

special ‘precompile’ status…).

- There might be more… hopefully this thread can unearth them…

Full contract abstraction, if you’re an absolute mad lad

The above proposal for “contract deployment abstraction” says “Let’s put lots of contract deployment logic in an app contract, to remove stuff from the kernel circuit. Let’s emit contract data as nullifiers and events”.

But the structure

of the nullifiers would still need to follow a rigid, enshrined structure. That’s because when executing

a function, the kernel circuit would need to know how to look the function up, against a nullifier in the tree.

So although the deployment process is “abstracted”, it isn’t really. It’s just moved from the kernel. In fact it might be that one canonical smart contract would need to be deployed to perform the deployment logic, and that would suffice for the whole network’s deployments.

A “full contract abstraction” approach would be to have the kernel “make a call” (whatever that means) to the smart contract which deployed

the contract in the first place to say “Please validate that this function exists in this contract which you deployed… I don’t know how to read the preimage of its nullifier (because it’s fully abstracted), but you do. Let me know if it’s a valid function of this contract, and I’ll proceed with verifying it.”

Pretty crazy, right?

I’m not advocating for it. I’m not even sure if it’s possible (or if the King of the Hill problem would rear its ugly head again). But it was a fun thought.