Public-Private Composability.

(A “private smart contract” architecture).

Mike Connor

Aztec

Aims: Private smart contracts

- Private states
- Private function execution
- Permissionless contract deployment
- Composability: calls between contracts
- Composability: calls between private & public functions
- Intuitive transaction semantics

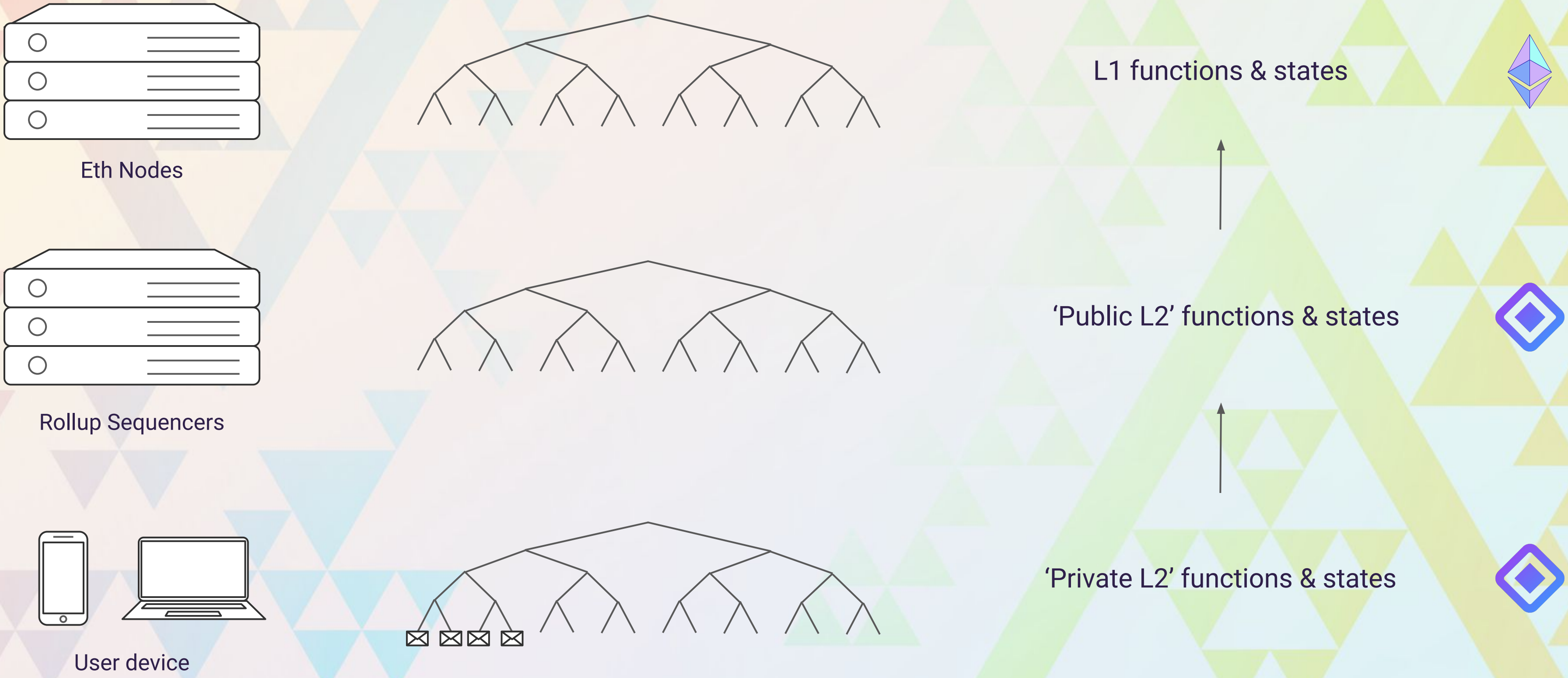
What is a smart contract?

A collection of state variables, and functions which may edit them.

```
contract ERC20 {
  mapping(address => uint256) balances;

  function transfer(address to, uint256 amount) {
    balances[msg.sender] -= amount;
    balances[to] += amount;
  }
}
```

What will an Aztec smart contract look like?



Eth Nodes

Rollup Sequencers

User device

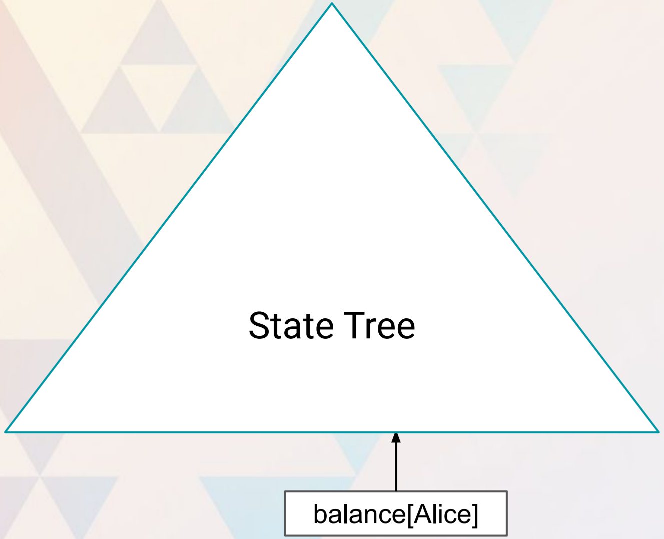
L1 functions & states

‘Public L2’ functions & states

‘Private L2’ functions & states

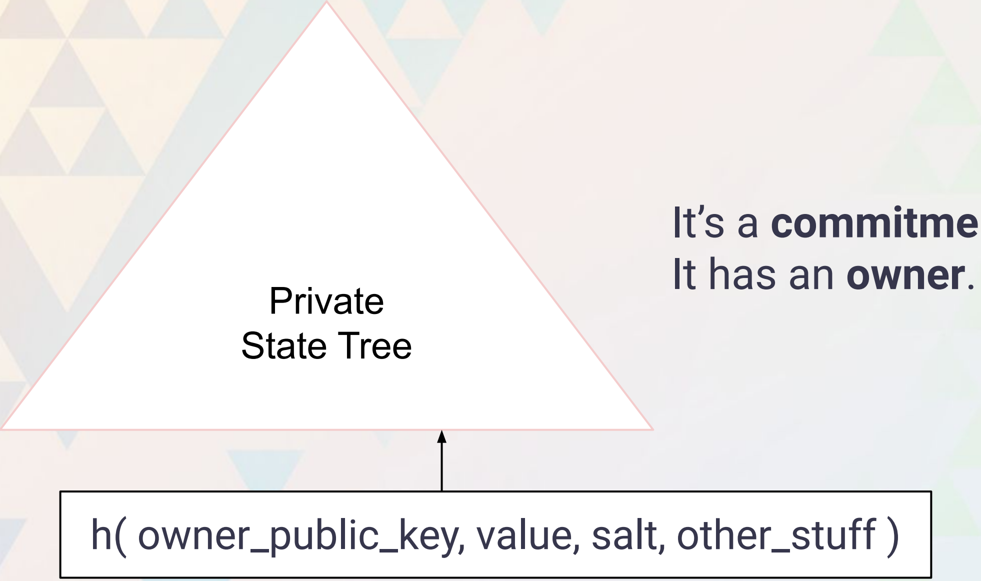
Private State

Aside: What's a public state?



It's just a value in a tree.

Aside: What's a private state?



Private State Tree

h(owner_public_key, value, salt, other_stuff)

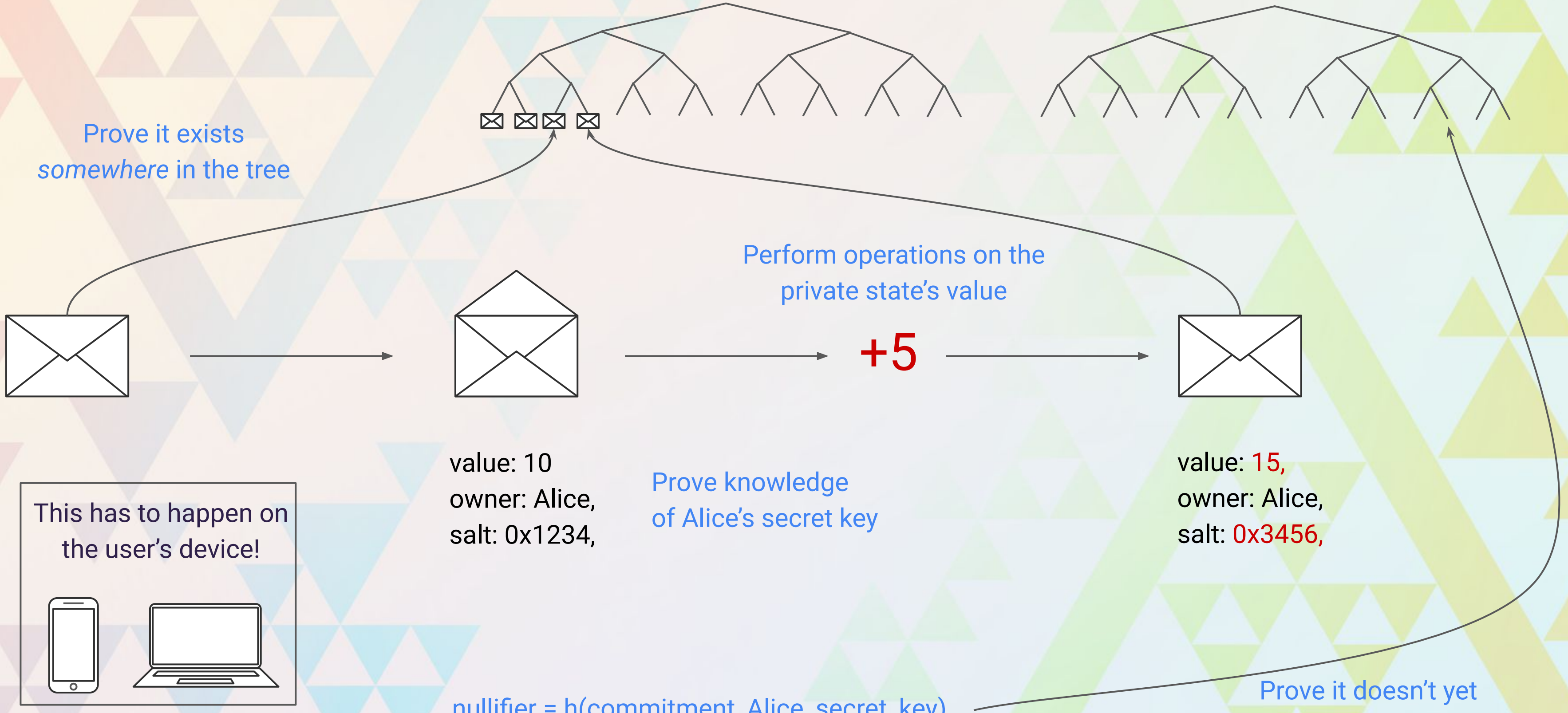
It's a **commitment** to a value in a tree.
It has an **owner**.

State - Account vs UTXO model

Public state → Account model

Private state → UTXO model

Private State change, within a Private Function



Prove it exists *somewhere* in the tree

Perform operations on the private state's value

value: 10
owner: Alice,
salt: 0x1234,

Prove knowledge of Alice's secret key

value: 15,
owner: Alice,
salt: 0x3456,

This has to happen on the user's device!

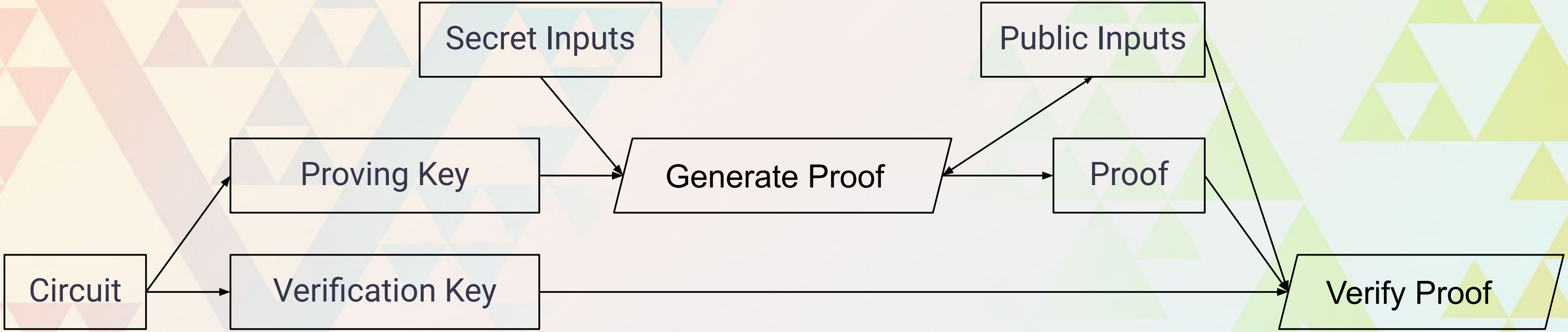
nullifier = h(commitment, Alice_secret_key)

Prove it doesn't yet exist in the tree.

Section 1

Private Functions

L2 functions are zk-SNARK circuits

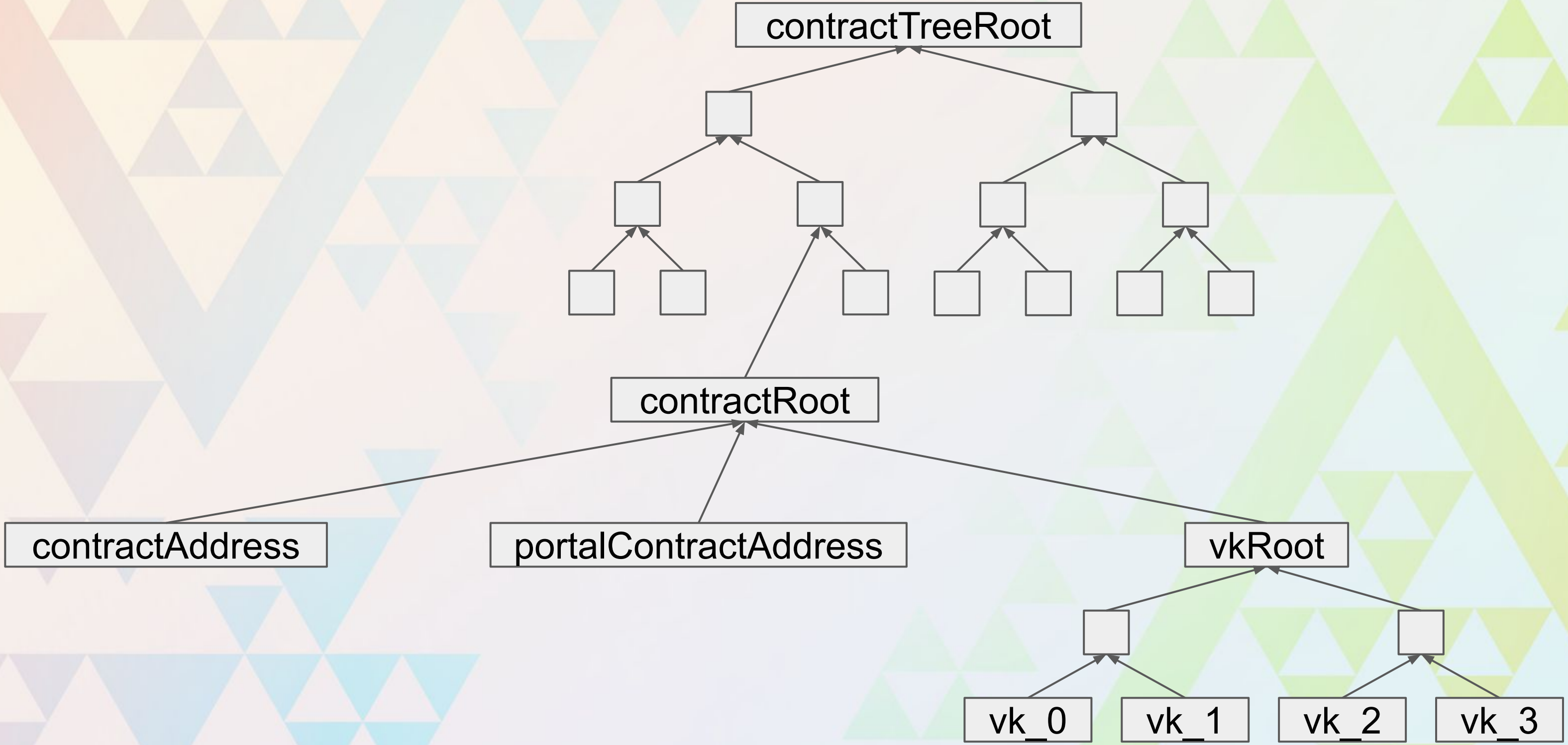


Notice:

- The Verification Key can be a unique ID for the circuit.
- h(Verification Key) *succinctly* represents the circuit.

*This will all be handled by Noir

Contract tree



Modifying private state in Zexe

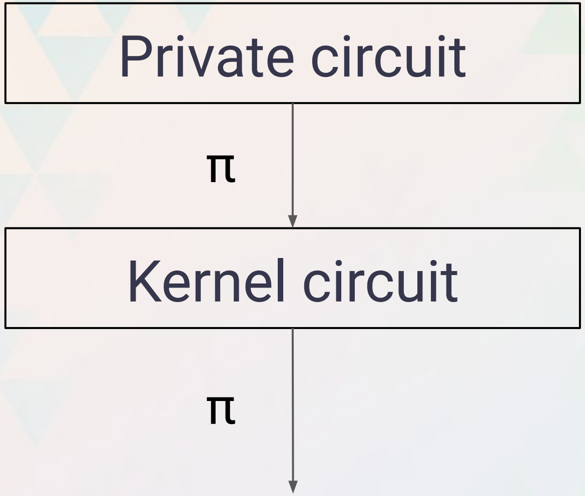
Each can be represented by h(verification_key)

$$h\left(\text{owner\_pk},\ \text{payload},\ \begin{array}{l}\text{birth\_predicate,}\\ \text{death\_predicate,}\end{array}\ \text{nonce},\ \text{salt}\right)$$

Instead…

$$h\left(\text{contract\_address},\ h\left(\text{owner\_pk},\ \overbrace{\begin{array}{l}\text{storage\_slot,}\\ \text{value,}\\ \text{is\_dummy,}\\ \text{creator\_pk,}\\ \text{memo,}\end{array}}^{\text{“payload”}}\ ,\ \text{nonce},\ \text{salt}\right)\right)$$

Hiding the function being executed with Zexe



Private circuit

π

Kernel circuit

π

No one learns which function was executed.

Functions calling functions

(...calling functions calling functions calling functions...)

Example

A decentralized exchange.

- Alice pings exchange contract to swap tokens A<>B
- Exchange contract pings contract A to transfer tokens to Exchange
- Exchange contract pings contract B to transfer tokens to Alice

Extremely basic transactions require nested function calls

Function calls

- contract_address
- 4 bytes of keccak hash of function signature
- arguments

- contract_address
- vk_index
- h(public inputs)

Function calls

```
import Contract2;

contract Contract1 {

  private uint x;

  function1(uint a, uint b, uint c) {
    d = Contract2.function2_1(a, b);
    x += d;
    Contract2.function2_2(c, x);
  }
}
```

```
import Contract3;

contract Contract2 {

  private uint y_1;
  uint y_2;

  function2_1(uint a, uint b) {
    d = Contract3.function3_1(a, b);
    y_1 += d;
    function2_3(a);
    return d;
  }

  function2_2(uint c, uint x) {
    return c * c;
  }

  public function2_3(uint a) {
    y_2 += a;
    Contract3.function3_2();
  }
}
```

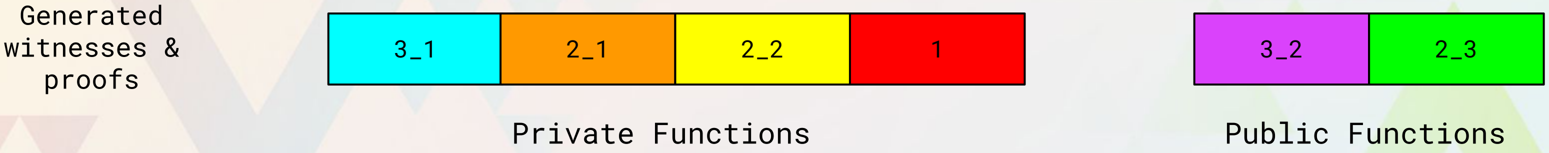
```
contract Contract3 {

  uint z;

  function3_1(uint a, uint b) {
    return a * b;
  }

  public function3_2() {
    z++;
  }
}
```

Function calls



Generated witnesses & proofs

3_1 2_1 2_2 1

Private Functions

3_2 2_3

Public Functions

6 distinct proofs are generated (one for each function).

How do we prove they relate to one-another?

Call stacks!

functions which call other functions	private_call_stack	public_call_stack
1	2_2 2_1	
2_1	3_1	2_3
2_3		3_2

```
import Contract2;

contract Contract1 {

  private uint x;

  function1(uint a, uint b, uint c) {
    d = Contract2.function2_1(a, b);
    x += d;
    Contract2.function2_2(c, x);
  }
}
```

```
import Contract3;

contract Contract2 {

  private uint y_1;
  uint y_2;

  function2_1(uint a, uint b) {
    d = Contract3.function3_1(a, b);
    y_1 += d;
    function2_3(a);
    return d;
  }

  function2_2(uint c, uint x) {
    return c * c;
  }

  public function2_3(uint a) {
    y_2 += a;
    Contract3.function3_2();
  }
}
```

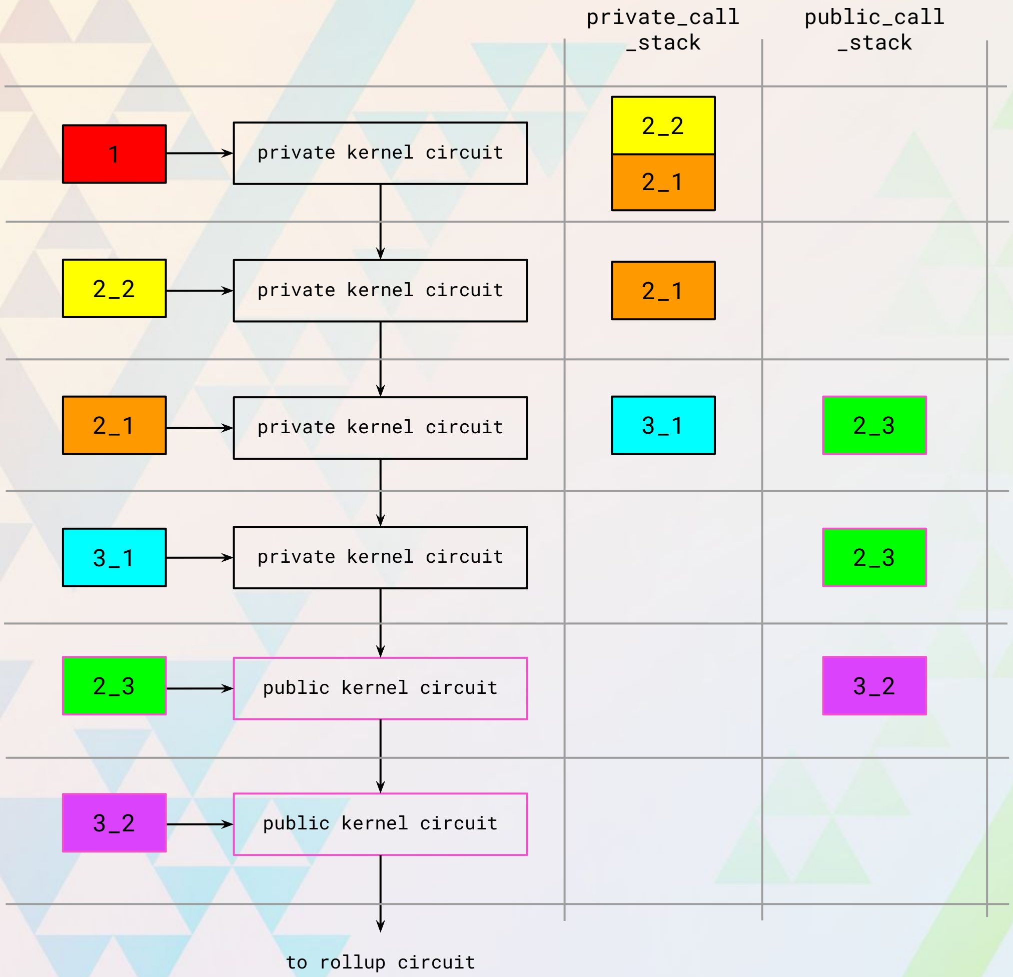
```
contract Contract3 {

  private uint z;

  function3_1(uint a, uint b) {
    return a * b;
  }

  public function3_2() {
    z++;
  }
}
```

Kernel Recursion



function	private_call_stack	public_call_stack
1	2_2 2_1	
2_1	3_1	2_3
2_3		3_2

Public Inputs ABIs,...

Private Circuits

```
custom_inputs: [],
custom_outputs: [],
emitted_data: [],
output_commitments: [],
input_nullifiers: [],
old_private_data_tree_root,
private_call_stack: [],
public_call_stack: [],
contract_deployment_call_stack: [],
l1_call_stack: [],
callback_stack: [{
    success_callback,
    failure_callback,
}],
executed_as_callback: {
    l1_result_hash,
    l1_results_tree_leaf_index,
},
bools,
```

Public Circuits

```
custom_inputs: [],
custom_outputs: [],
emitted_data: [],
state_transitions: [],
state_reads: [],
old_private_data_tree_root,

public_call_stack: [],
contract_deployment_call_stack: [],
l1_call_stack: [],
callback_stack: [{
    success_callback,
    failure_callback,
}],
executed_callback: {
    l1_result_hash,
    l1_results_tree_leaf_index,
},
bools,
prover_address,
```

Private Kernel ABI

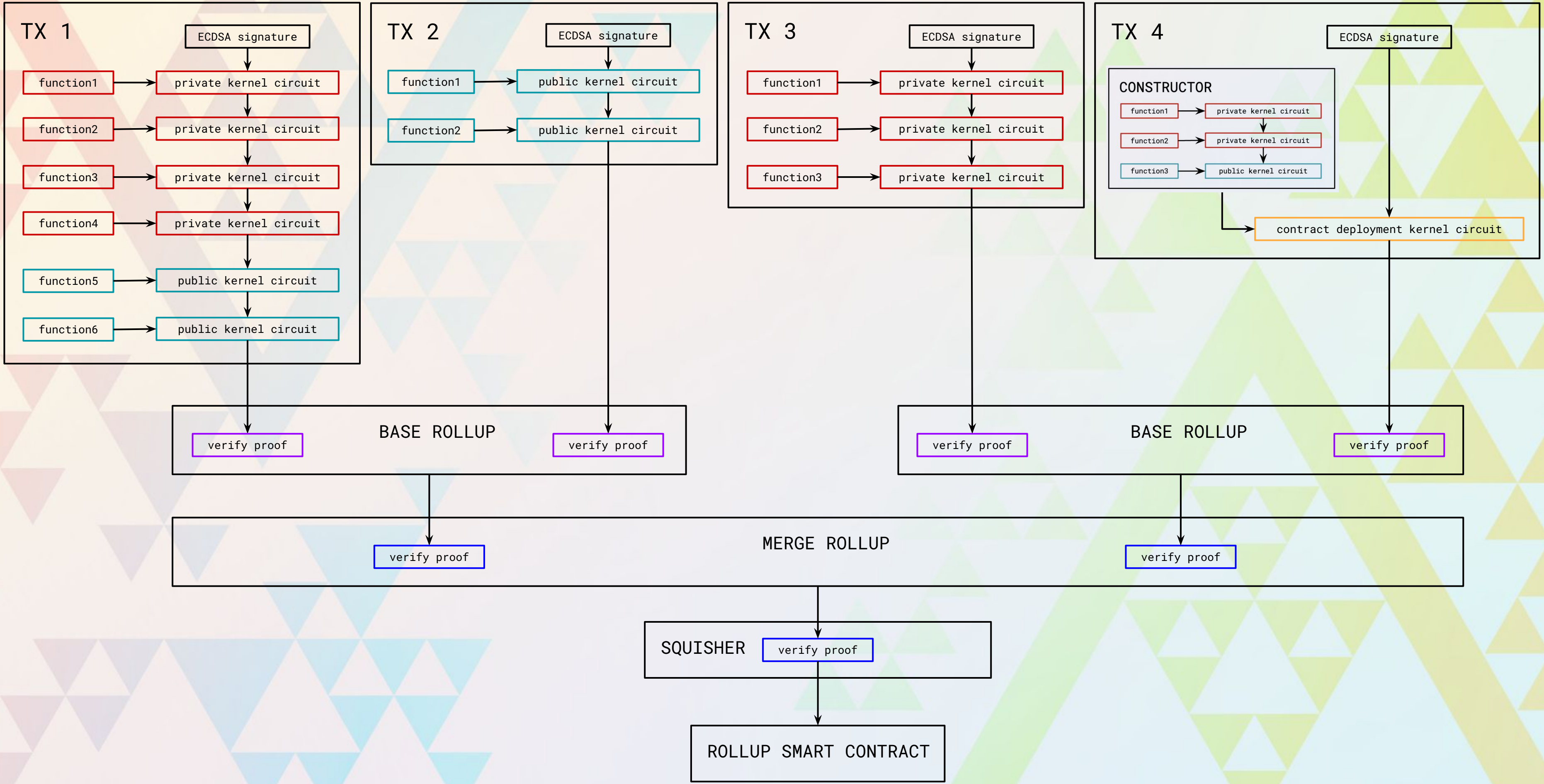
Private Inputs

```
signature,
start: {
    aggregated_proof,
    private_call_count,
    private_call_stack: [],
    public_call_stack: [],
    contract_deployment_call_stack: [],
    l1_call_stack: [],
    callback_stack: [],
    optionally_revealed_data: [],
    output_commitments: [],
    input_nullifiers: [],
},
previous_kernel: {
    proof,
    public_inputs,
    vk,
    vk_index,
    vk_path,
},
private_call: {
    function_signature,
    public_inputs,
    call_context,
    proof,
    vk,
    vk_index,
    vk_path,
    portal_contract_address,
    contract_leaf_index,
    contract_path,
    privately_executed_as_callback: {},
}
```

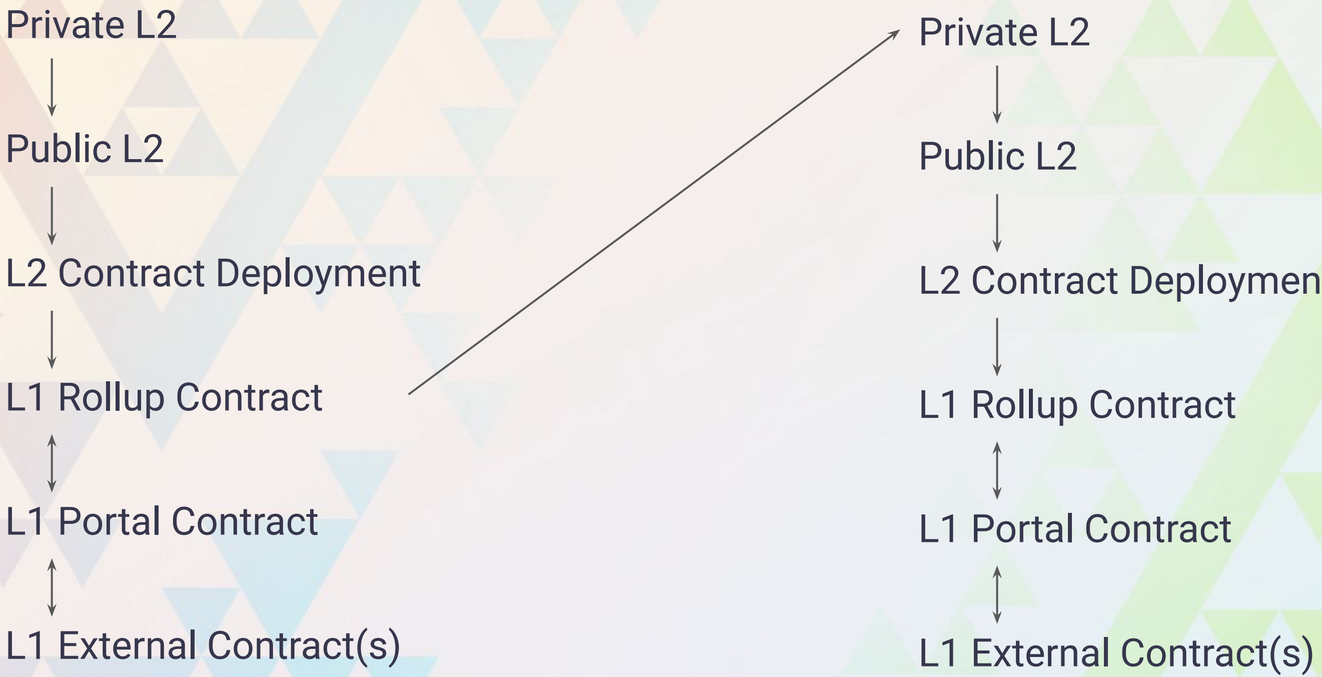
Public Inputs

```
end: {
    aggregated_proof,
    private_call_count,
    private_call_stack: [],
    public_call_stack: [],
    contract_deployment_call_stack: [],
    l1_call_stack: [],
    callback_stack: [],
    optionally_revealed_data: [],
    output_commitments: [],
    input_nullifiers: [],
},
constants: {
    old_tree_roots: {
        private_data_tree,
        contract_tree,
        l1_results_tree,
        private_kernel_vk_tree,
    },
    is_constructor_recursion,
    is_callback_recursion,
    executed_as_callback: {},
},
globals: {},
bools
```

Rollup topology



Calls



Private L2

Public L2

L2 Contract Deployment

L1 Rollup Contract

L1 Portal Contract

L1 External Contract(s)

Private L2

Public L2

L2 Contract Deployment

L1 Rollup Contract

L1 Portal Contract

L1 External Contract(s)

What does the kernel circuit do?

It makes sure txs follow the rules.

- Verifies msg_sender's signature
- Pops the next call (app proof) off the call stack
- Verifies the app proof
- Verifies the previous kernel proof
- Validates consistency between previous kernel's accumulated end data, and this kernel's start data
- Ensures the function (app proof) being verified belongs to the purported contract
- Ensures the contract exists
- For L1 calls, checks the purported portal contract's address corresponds to this contract
- Pushes new function calls to call stacks
- Checks delegatecall / staticcall contexts
- 'Silos' new commitments & nullifiers
- Optionally reveals data to public L2 / L1:
    - Fees, Events, Calls, Deployment data
- Checks that executed callbacks refer to valid L1 Result leaves
- (And some other stuff)

We recurse through the call stacks until they're empty.

Thank you!

Mike Connor

Aztec

@mike_connor
@aztecnetwork