Circuit safety and an Introduction to Noir

Maddiaa and Maxim
Aztec

Aztec

Content

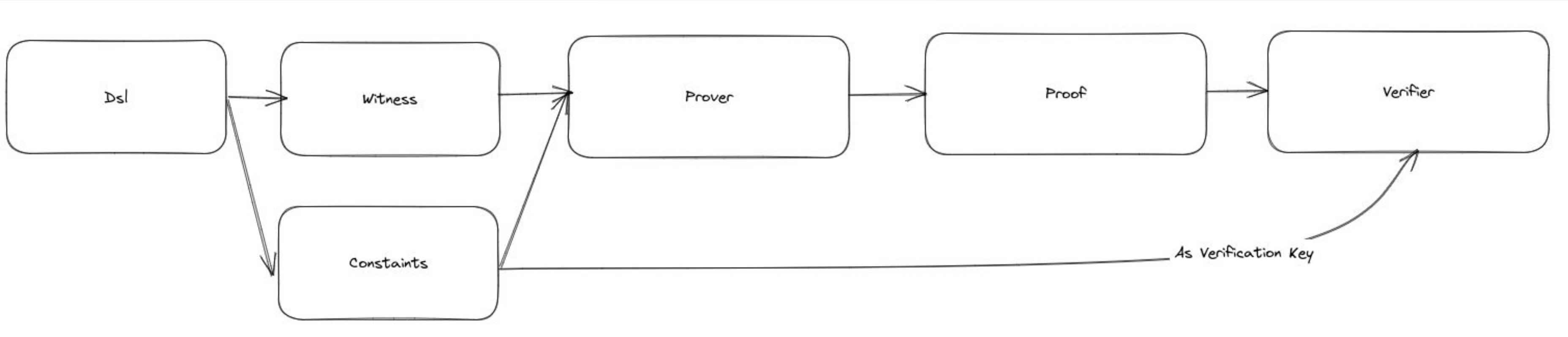
- The ZK Programming Model
- Proving System Vulnerabilities
- Common circuit bugs
- Noir
- Noir in Aztec 3
- Unconstrained Functions

We Know

- **ZK proofs are powerful**
  - Information hiding
  - Efficient verification

The zk programming model

- New programming paradigm
- Focus on constraining correctness (you are proving after all)



Aztec

Vulnerability Classes

- Proving system vulnerabilities
- Circuit construction vulnerabilities

Proving system vulnerabilities

- Remember there is a huge dependency

Types of issues:

- Frozen Heart
- Setup Issues
- Implementation bugs

Frozen Heart

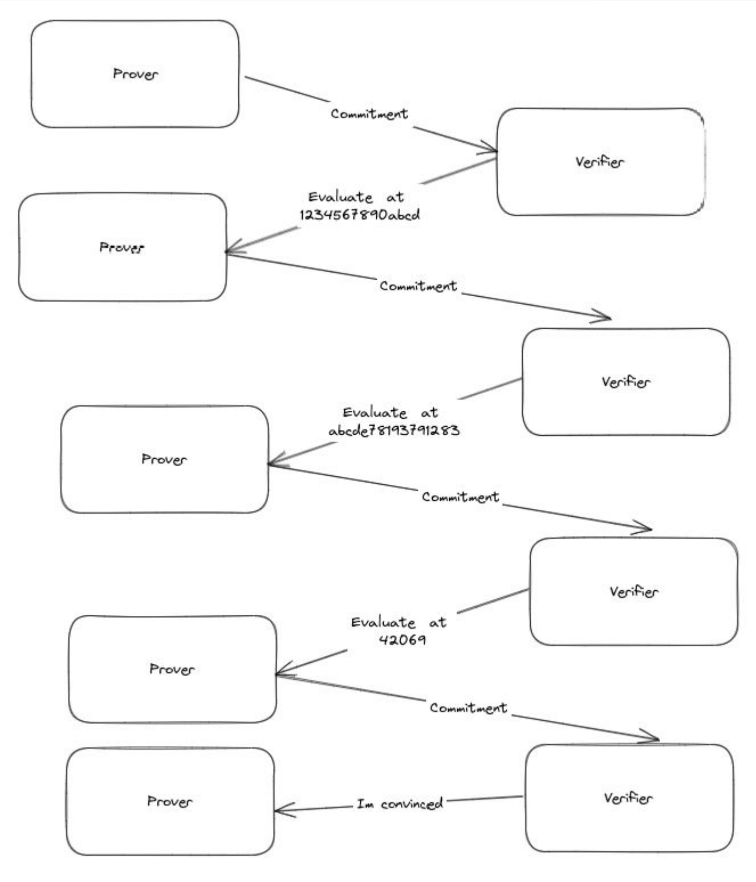
- Implementation issue in multiple plonk | bulletproof implementations
- Stems from lack of guidance over fiat shamir implementation

Frozen Heart - Fiat shamir

- Converts an interactive proof into a non interactive one.

Frozen Heart - Fiat shamir

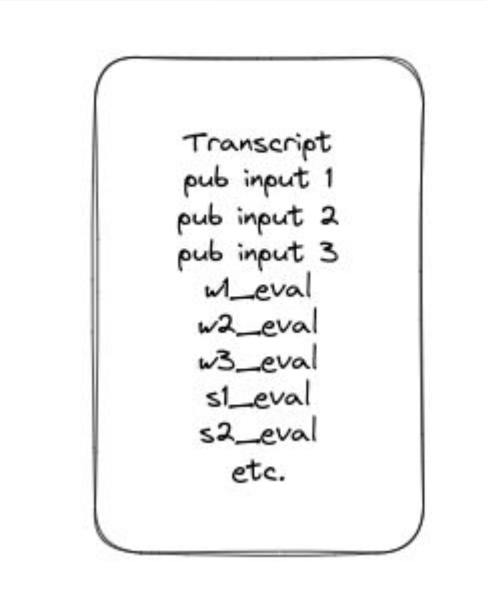
- Interactive proofs look like this:
- The verifier will choose random values for the prover to evaluate
- Multi-round format

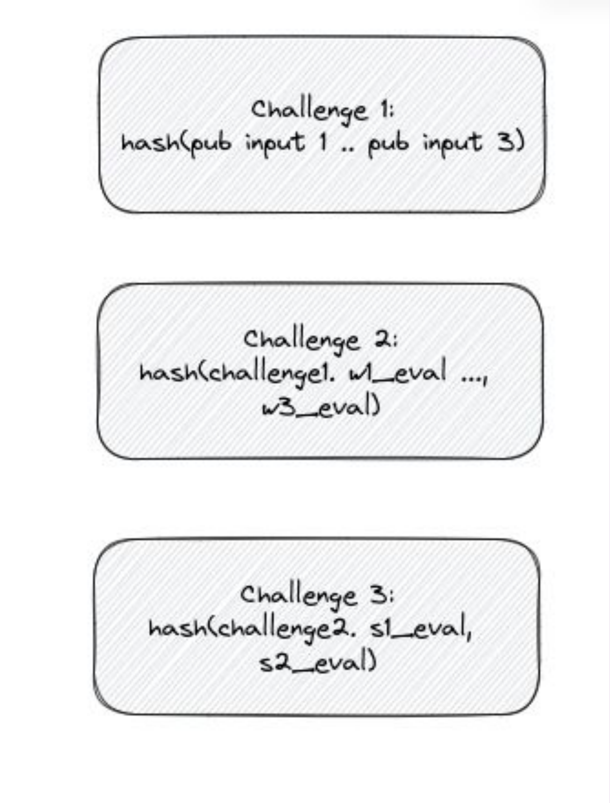


Aztec

Frozen Heart - Fiat shamir

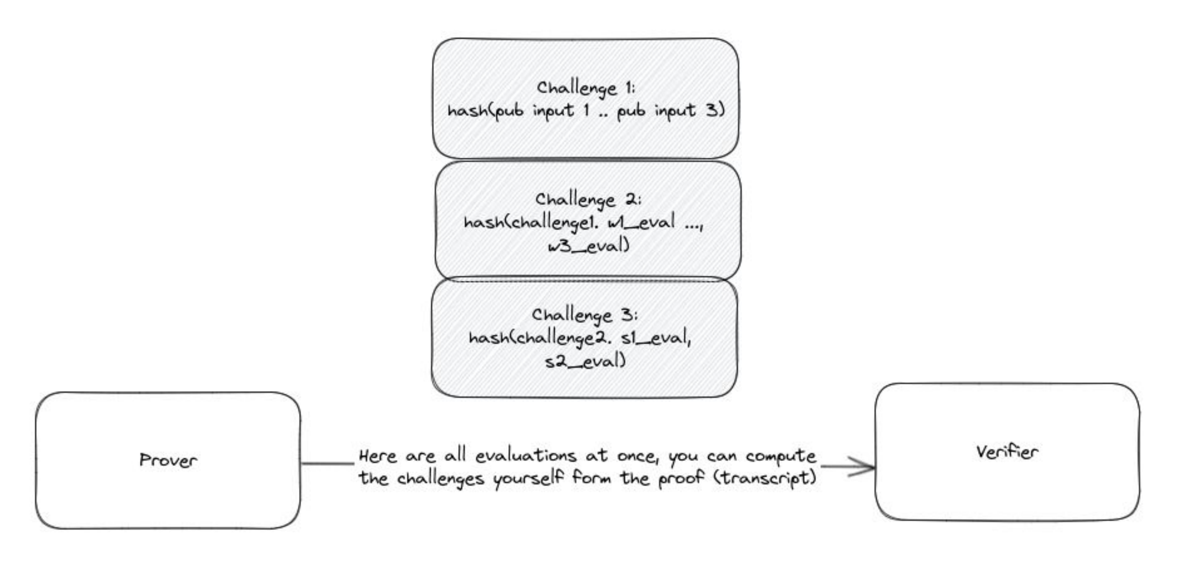
- A transcript is the generated proof data.





Frozen Heart - Fiat shamir

- How to generate the challenges is known by both parties ahead of time
- Only one hop is required



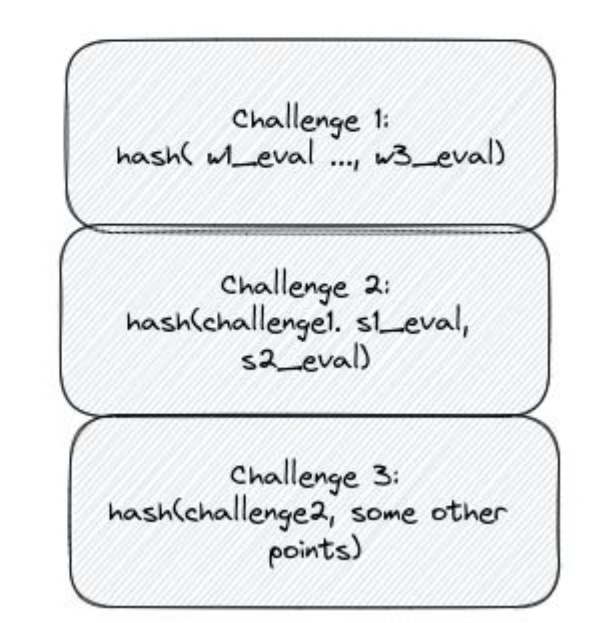
Aztec

Frozen Heart - The issue

- The plonk paper just assumed people would create secure constructions.
- Some implementations did not hash the entire transcript

Frozen Heart - The issue

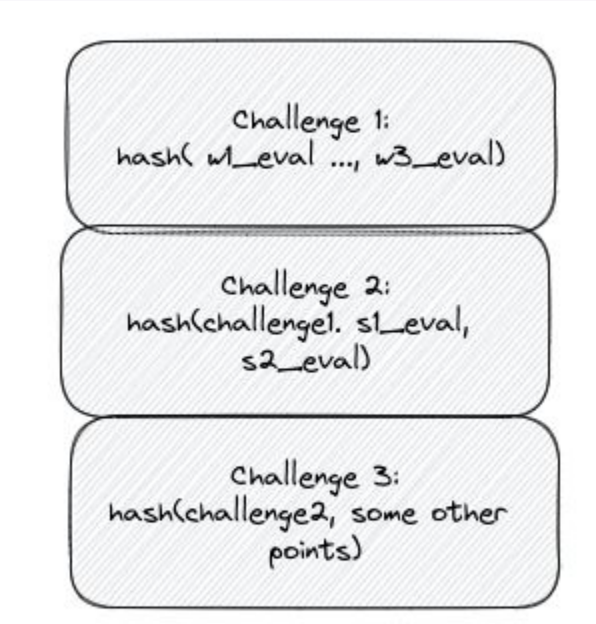
- The plonk paper just assumed people would create secure constructions.
- Some implementations did not hash the entire transcript
- Imagine this setup



Aztec

Frozen Heart - The issue

- In this construction you could technically forge proofs with any public inputs, as they are not included in the transcript.



Aztec

Frozen Heart - Overview

- Implementation issue
- It allows proof forgery, but is very hard to attack in practice. You'll still have to do a lot of hashing to create a forgery

Q&A Pause
(I’m not a cryptographer though so no hard questions)

Common Circuit vulnerabilities

Types of issues:

- Under-constrained circuits
- Mismatching bit lengths
- Double spend nullifier issues

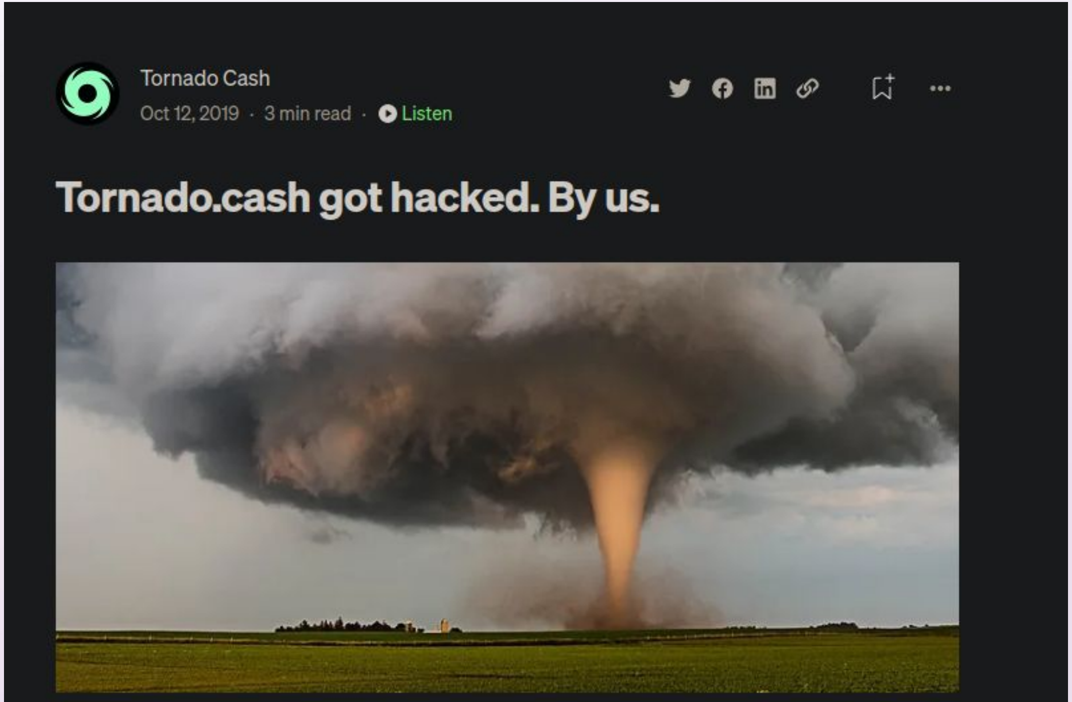
Smart contract circuit related issues

- Smart contract nullifier issues

Under-constrained Circuits

Underconstrained circuits - an example

- Tornado cash



Aztec

Underconstrained circuits - what dis?

- The constructed circuit does not have the correct rules to fulfill its intended purpose

Underconstrained circuits - An illustration

- Me saying just add a constraint doesn't actually help much
- Why does not adding a constraint cause bugs
- A quick detour into plonk land
- (dont quote me on any of this I am no cryptogtapher)

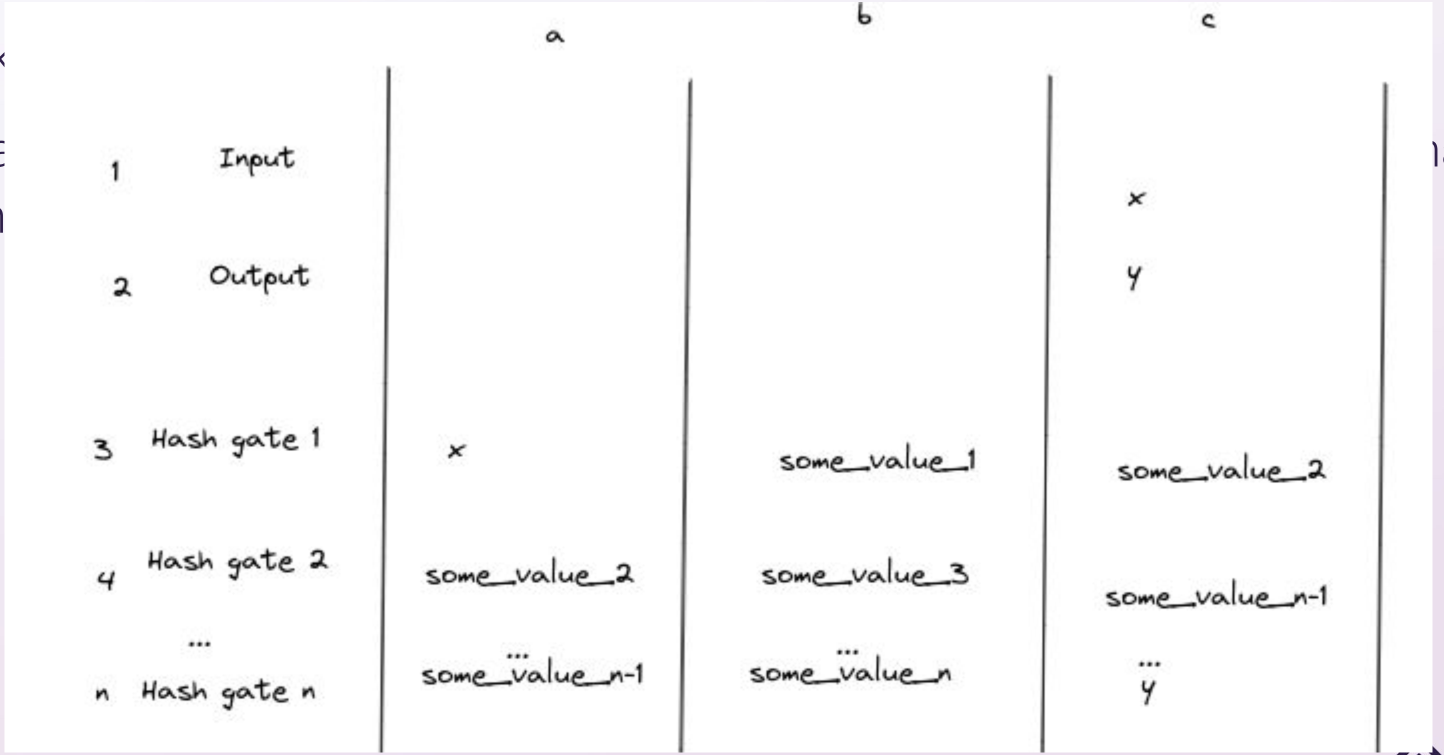
Aztec

Underconstrained circuits - An illustration

- Take a simple hash function H(x) => y
- In a plonkish circuit we will have a trace with a preimage, a trace of the hash function and an output.

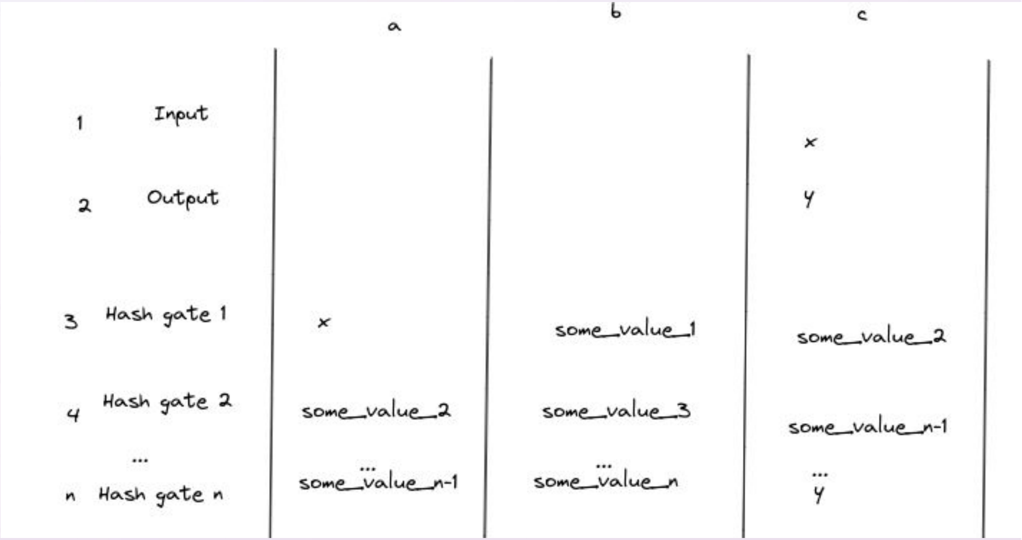
Underconstrained circuits - An (very rough) illustration

- Tak
- In a … hash fun



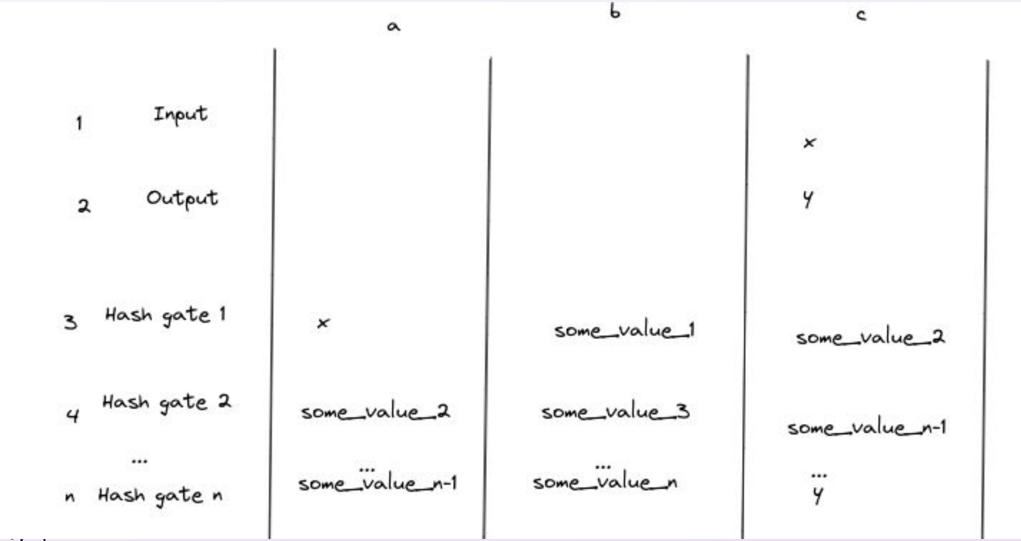
Underconstrained circuits - An (very rough) illustration

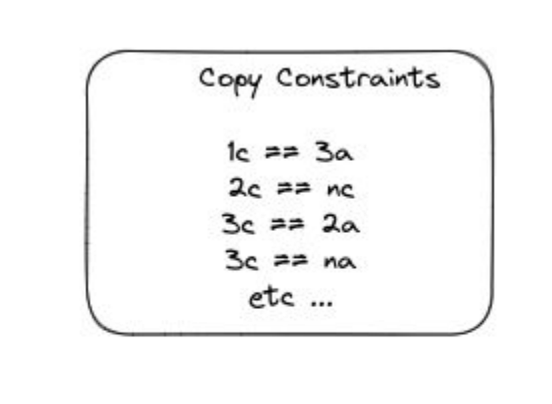
- Take a simple hash function H(x) => y
- input x
- input y
- Some hash trace that takes me from x->y
- I MUST specifically constrain that y_in = y_out of the hash function
- If not can create valid proofs



Underconstrained circuits - An (very rough) illustration

- If we do not constrain 2c === nc, we can technically put anything in 2c and the verifier be convinced!





Aztec

Underconstrained circuits - an example

- The bug was within circomlibs standard library for a MiMC hash
- The value was assigned but not actually constrained

```
outs[0] = S[nInputs - 1].xL_out;
```

Underconstrained circuits - The fix

- Literally adding a constraint

outs[0] <== S[nInputs - 1].xL_out;

```
2  circuits/mimcsponge.circom                                                Viewed

@@ -21,7 +21,7 @@ template MiMCSponge(nInputs, nRounds, nOutputs) {
      }                                          }
    }                                          }

-   outs[0] = S[nInputs - 1].xL_out;         +   outs[0] <== S[nInputs - 1].xL_out;

    for (var i = 0; i < nOutputs - 1; i++) {   for (var i = 0; i < nOutputs - 1; i++) {
      S[nInputs + i] = MiMCFeistel(nRounds);     S[nInputs + i] = MiMCFeistel(nRounds);
```


Aztec

Underconstrained circuits - The fix

- Literally adding a constraint

P.s. noir fixes this

outs[0] <== S[nInputs - 1].xL_out;

```diff
circuits/mimcsponge.circom
@@ -21,7 +21,7 @@ template MiMCSponge(nInputs, nRounds, nOutputs) {
     }
   }

-  outs[0] = S[nInputs - 1].xL_out;
+  outs[0] <== S[nInputs - 1].xL_out;

   for (var i = 0; i < nOutputs - 1; i++) {
     S[nInputs + i] = MiMCFeistel(nRounds);
```

Underconstrained circuits

- Circom is a pretty low level language, you must hand wire constraints
- Static analysis tools exist
  - https://github.com/Veridise/Picus
  - https://github.com/trailofbits/circomspect
- Noir will wire these constraints for you.

Aztec

The usual suspects

The usual suspects

- Overflow / underflow
- Signature malleability

Overflow / Underflow

Overflow / Underflow: An example

- EVERYTHING IS MOD SOME PRIME FIELD
- Sometimes exists in the solidity context (Make sure any circuit values used in smart contracts are mod the prime field)
- In the circuit context, add range constraints wherever you can.

Aztec

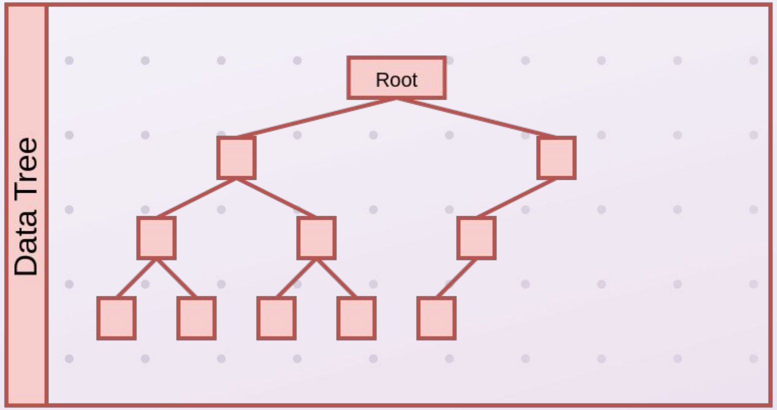
Signature Malleability

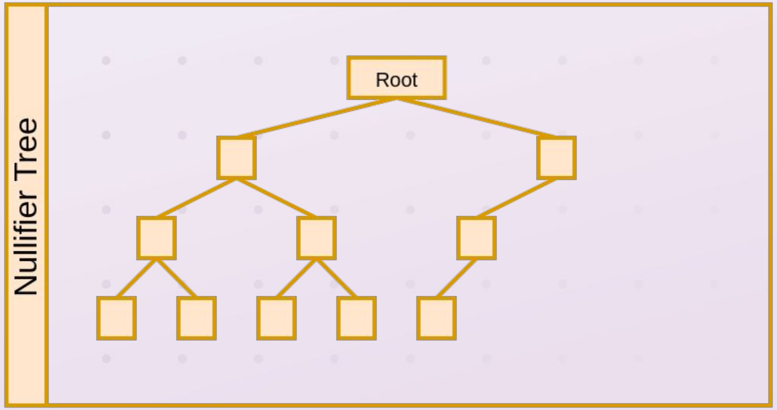
DON’T USE SIGNATURES AS YOUR NULLIFIERS

- This creates non deterministic nullifiers

Nullifier: recap / primer

- State will exist in two trees
    - Data tree and Nullifier Tree

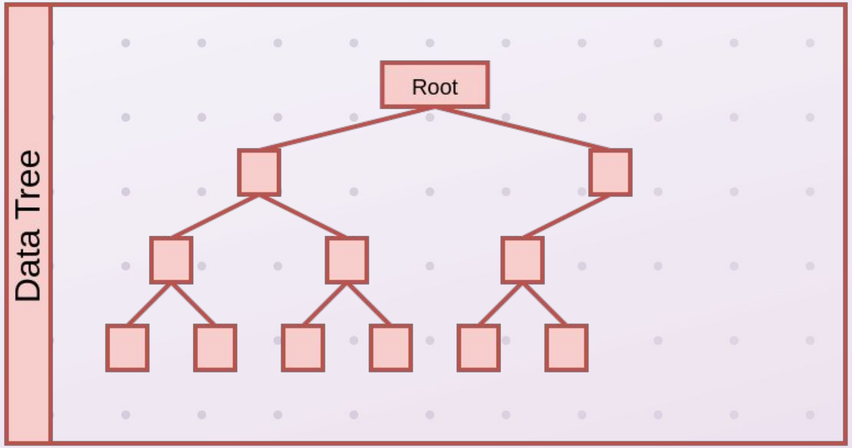


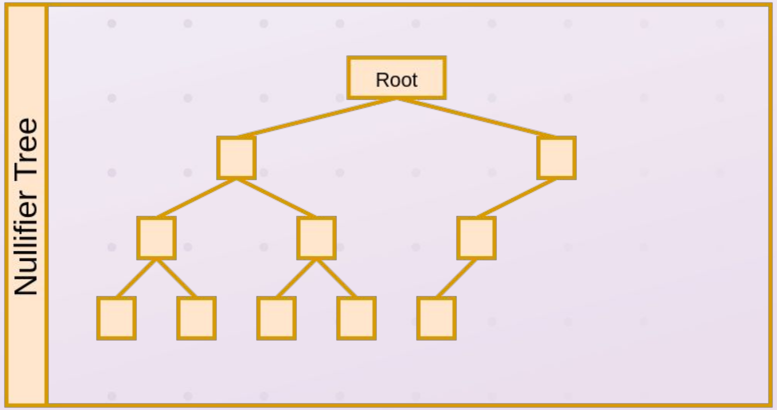


Aztec

Nullifier: recap / primer

- Insert value

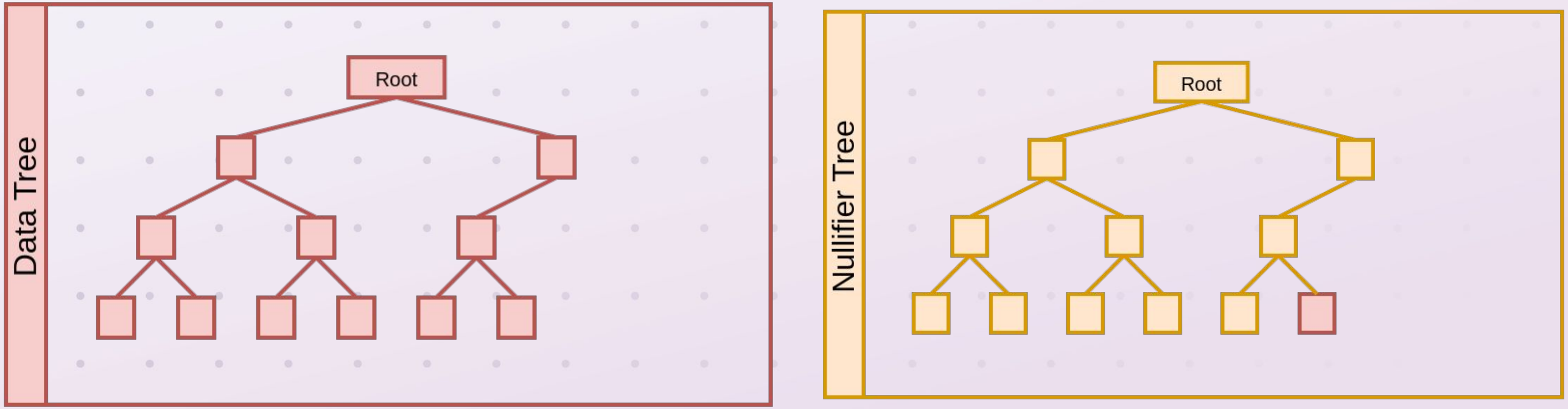




Aztec

Nullifier: recap / primer

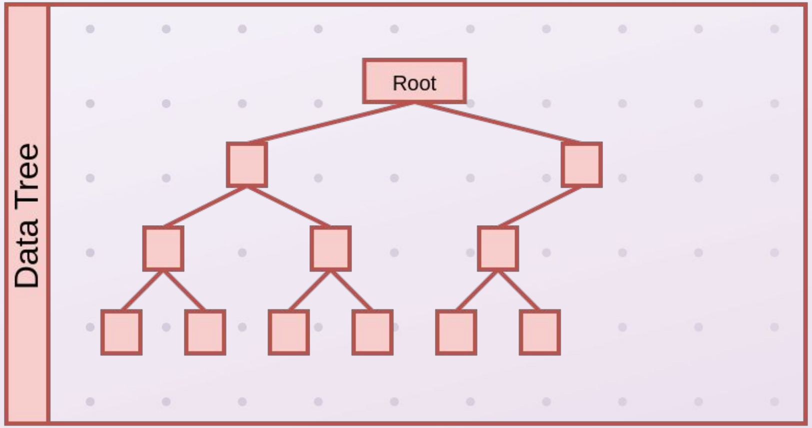
- Spend value
  - We don't update the data tree, we insert into the nullifier tree a value deterministically derived from some information in the data tree

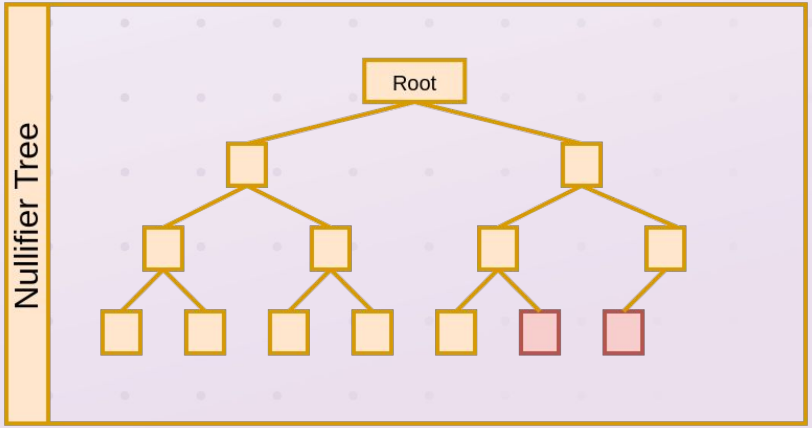


Aztec

Nullifier: recap / primer

- Double spend a value
    - Insert into the nullifier tree twice for a single value

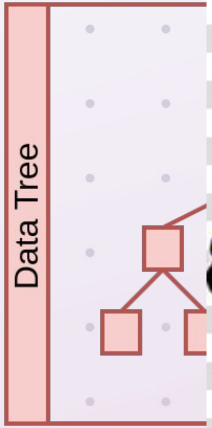




Aztec

Nullifier: recap / primer (bruh)

- Spend
  - We
    fro



Non Deterministic Nullifiers: example

- ECDSA construction is malleable
  - Why?
  - secp256k1 has a prime order of p
  - A signature (r, s) has another value signature (r, p-s).
- This leads to a many->one rel between nullifiers (there is more than one valid nullifier for each commitment)

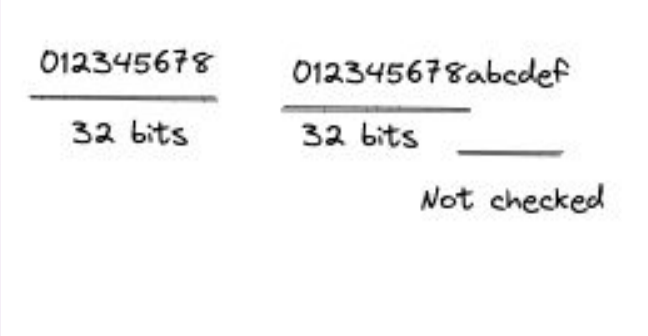
Non Deterministic Nullifiers: example

- This was a real world vulnerability (0xPARC stealthdrop)
- They have an excellent write up
- https://github.com/stealthdrop/stealthdrop/blob/main/README.md#ecdsa-signature-verification

Mismatching bit length

Vulnerability in Aztec Connect :(

- Nullifier double spend possibility
- Each nullifier had a tree 32bit index
- There was no check that the provided nullifier was 32 bits
- Only the first 32 bits were checked so a correct value appended with ANYTHING would become a valid nullifier



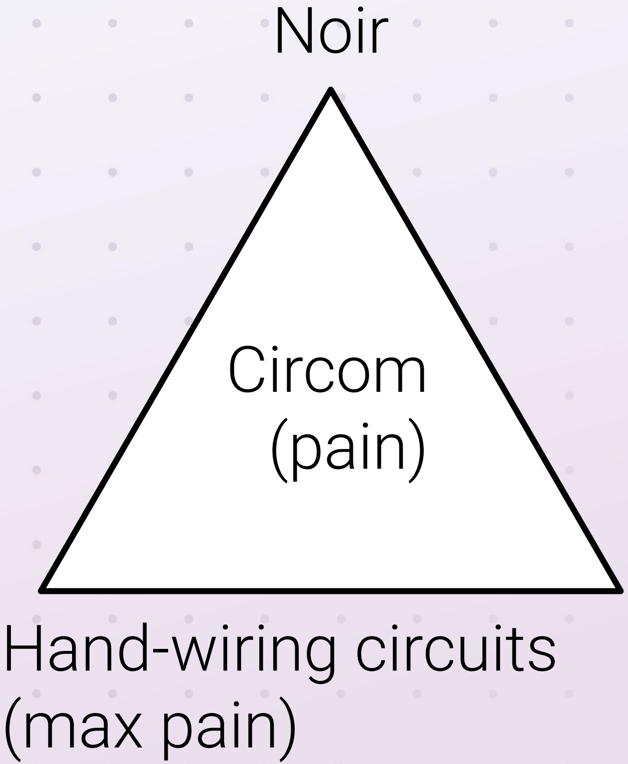
Aztec

Current ZK language landscape

Language landscape

- Circom (low level - very customisable

The pyramid of circuit construction



Writing circuits in cpp with aztec’s proving lib (my day job help)

```
void join_split_circuit(Composer& composer, join_split_tx const& tx)
{
    join_split_inputs inputs = {
        .proof_id = witness_ct(&composer, tx.proof_id),
        .public_value = suint_ct(witness_ct(&composer, tx.public_value), NOTE_VALUE_BIT_LENGTH, "public_value"),
        .public_owner = witness_ct(&composer, tx.public_owner),
        .asset_id = suint_ct(witness_ct(&composer, tx.asset_id), ASSET_ID_BIT_LENGTH, "asset_id"),
        .num_input_notes = witness_ct(&composer, tx.num_input_notes),
        .input_note1_index = suint_ct(witness_ct(&composer, tx.input_index[0]), DATA_TREE_DEPTH, "input_index0"),
        .input_note2_index = suint_ct(witness_ct(&composer, tx.input_index[1]), DATA_TREE_DEPTH, "input_index1"),
        .input_note1 = value::witness_data(composer, tx.input_note[0]),
        .input_note2 = value::witness_data(composer, tx.input_note[1]),
        .output_note1 = value::witness_data(composer, tx.output_note[0]),
        .output_note2 = value::witness_data(composer, tx.output_note[1]),
        // Construction of partial_claim_note_witness_data includes construction of bridge_call_data, which contains
        // many constraints on the bridge_call_data's format and the bit_config's format:
        .partial_claim_note = claim::partial_claim_note_witness_data(composer, tx.partial_claim_note),
        .signing_pub_key = stdlib::create_point_witness(composer, tx.signing_pub_key),
        .signature = stdlib::schnorr::convert_signature(&composer, tx.signature),
        .merkle_root = witness_ct(&composer, tx.old_data_root),
        .input_path1 = merkle_tree::create_witness_hash_path(composer, tx.input_path[0]),
        .input_path2 = merkle_tree::create_witness_hash_path(composer, tx.input_path[1]),
        .account_note_index =
            suint_ct(witness_ct(&composer, tx.account_note_index), DATA_TREE_DEPTH, "account_note_index"),
        .account_note_path = merkle_tree::create_witness_hash_path(composer, tx.account_note_path),
        .account_private_key = witness_ct(&composer, static_cast<fr>(tx.account_private_key)),
        .alias_hash = suint_ct(witness_ct(&composer, tx.alias_hash), ALIAS_HASH_BIT_LENGTH, "alias_hash"),
        .account_required = bool_ct(witness_ct(&composer, tx.account_required)),
        .backward_link = witness_ct(&composer, tx.backward_link),
        .allow_chain = witness_ct(&composer, tx.allow_chain),
    };
    auto outputs = join_split_circuit_component(inputs);
    const field_ct defi_root = witness_ct(&composer, 0);
```


Aztec

Writing circuits in circom (somehow worse than cpp imo)

```
template MerkleTreeInclusionProof(n_levels) {
    signal input leaf;
    signal input path_index[n_levels];
    signal input path_elements[n_levels][1];
    signal output root;

    component hashers[n_levels];
    component mux[n_levels];

    signal levelHashes[n_levels + 1];
    levelHashes[0] <== leaf;

    for (var i = 0; i < n_levels; i++) {
        // Should be 0 or 1
        path_index[i] * (1 - path_index[i]) === 0;

        hashers[i] = HashLeftRight();
        mux[i] = MultiMux1(2);

        mux[i].c[0][0] <== levelHashes[i];
        mux[i].c[0][1] <== path_elements[i][0];

        mux[i].c[1][0] <== path_elements[i][0];
        mux[i].c[1][1] <== levelHashes[i];

        mux[i].s <== path_index[i];
        hashers[i].left <== mux[i].out[0];
        hashers[i].right <== mux[i].out[1];
```


Aztec

Writing circuits in noir (very based)

```
use dep::std;

fn main(
  recipient: pub Field,
  // Private key of note
  // all notes have the same denomination
  priv_key: Field,
  // Merkle membership proof
  note_root: pub Field,
  index: Field,
  note_hash_path: [Field; 3],
  // Random secret to keep note_commitment private
  secret: Field,
  // Hash to be checked against the nullifier computed in the circuit
  nullifierHash: pub Field,
) -> pub Field {
    // Compute public key from private key to show ownership
    let pubkey = std::scalar_mul::fixed_base(priv_key);
    let pubkey_x = pubkey[0];
    let pubkey_y = pubkey[1];

    // Compute input note commitment
    let note_commitment = std::hash::pedersen([pubkey_x, pubkey_y, secret]);

    // Compute input note nullifier
    let nullifier = std::hash::pedersen([note_commitment[0], index, priv_key]);
    constrain nullifierHash == nullifier[0];

    // Check that the input note commitment is in the root
    let new_root = compute_root_from_leaf(note_commitment[0], index, note_hash_path);
    constrain new_root == note_root;

    // Cannot have unused variables, return the recipient as public output of the circuit
    recipient
```


Aztec

Why Noir?

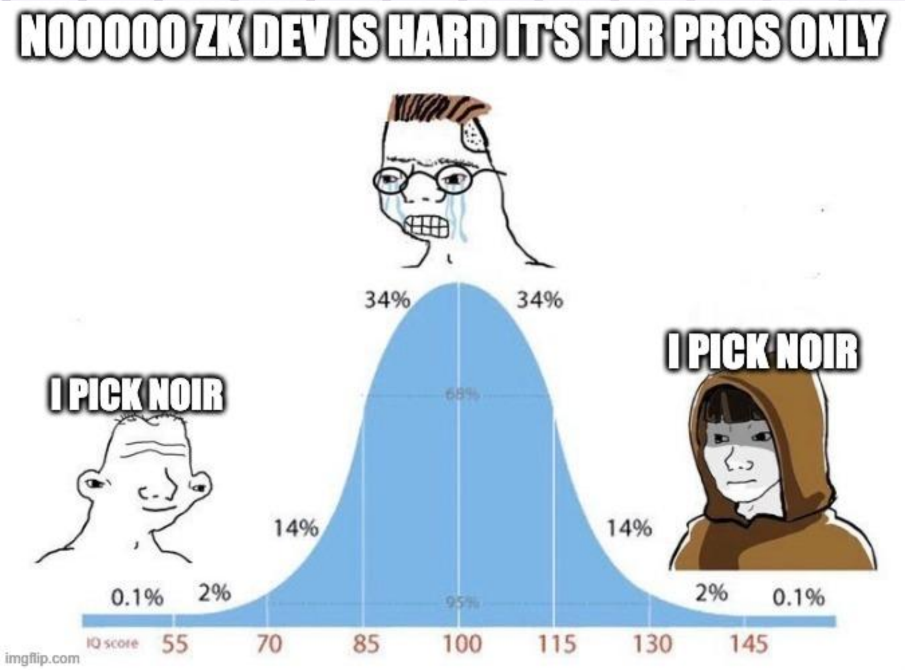
✅ Write ZK like a Dev 🧑‍💻

❌ not as a Mad Scientist 🧑‍🔬

Aztec

Before Noir

- **Writing ZK programs was hard**
  - Advanced cryptography background
  - Manually write system constraints
- **Gardens were walled**
  - Tied to proving backends
  - Tied to ecosystems / chains


NOOOOO ZK DEV IS HARD IT'S FOR PROS ONLY
I PICK NOIR
I PICK NOIR
34%
34%
68%
14%
14%
0.1%
2%
95%
2%
0.1%
IQ score
55
70
85
100
115
130
145
imgflip.com

Aztec

What Noir Offers

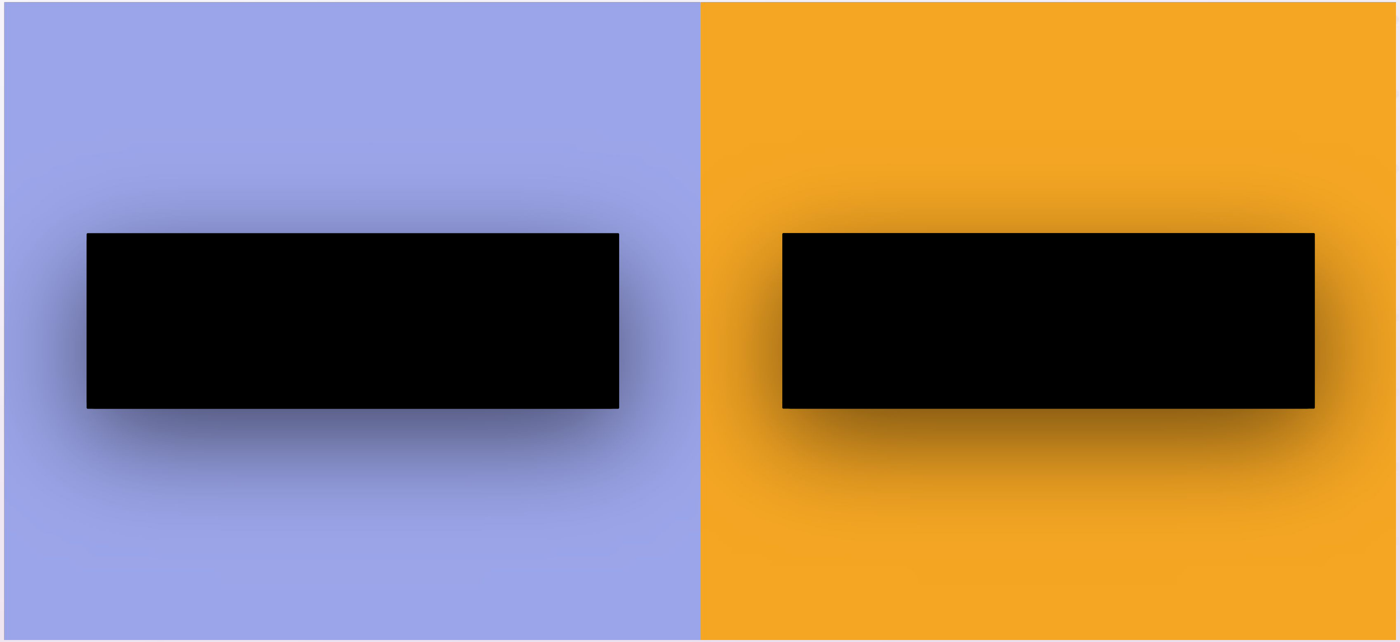
⚠️ we’re still in alpha

Spot the Difference

```
fn sum(x : u32, y : u32) -> u32 {
  x + y
}
```

```
fn sum(x : u32, y : u32) -> u32 {
  x + y
}
```

Spot the Difference



Spot the Difference

```
fn sum(x : u32, y : u32) -> u32 {
  x + y
}
```

Noir

```
fn sum(x : u32, y : u32) -> u32 {
  x + y
}
```

The Noir Programming Language

Primitive Types

- Integer
- Field
- Boolean
- String

```
fn main() {
  let x : u32 = 10;        // Integer
  let y : Field = 10;      // Field
  let true = true;         // Boolean
  let hello = "hello";     // String
}
```

The Noir Programming Language

Compound Types

- **Arrays**
- Tuples
- Structs

```
fn main(x : Field, y : Field) {
    let my_arr = [x, y];
    let your_arr: [Field; 2] = [x, y];
}
```

The Noir Programming Language

Compound Types

- Arrays
- **Tuples**
- Structs

```
fn main() {
    let tup: (u8, u64, Field) = (255, 500, 1000);
}
```

The Noir Programming Language

Compound Types

- Arrays
- Tuples
- **Structs**

```
struct Animal {
  hands: Field,
  legs: u8,
  eyes: u32,
}

fn main() {
  let dog = Animal {
    eyes: 2,
    hands: 0,
    legs: 4,
  };
}
```

The Noir Programming Language

Modularity

- Functions

```
fn main(x : Field) {
    let z = sum(x);
}

fn sum(x : Field) -> Field {
    x + x
}
```

The Noir Programming Language

Control Flow

- For loops
- If Statements

```
fn sort(mut a: [u32; 4]) -> [u32; 4] {
    for i in 1..4 {
        for j in 0..i {
            if a[i] < a[j] {
                let c = a[j];
                a[j] = a[i];
                a[i] = c;
            }
        }
    }
    a
}

fn main(a: [u32; 4]) {
    let sorted_arr = sort(a);

    constrain sorted_arr[0] < sorted_arr[3];
}
```

The Noir Programming Language

Operators

- Logical
- Bitwise

```
fn main() {
    let my_val = 5;
    let mut shifted_val : u8 = 0;

    if (my_val != 1) & (my_val < 10) {
        shifted_val = my_val as u8 << 1;
    }
    constrain shifted_val == 10;
}
```

The Noir Programming Language

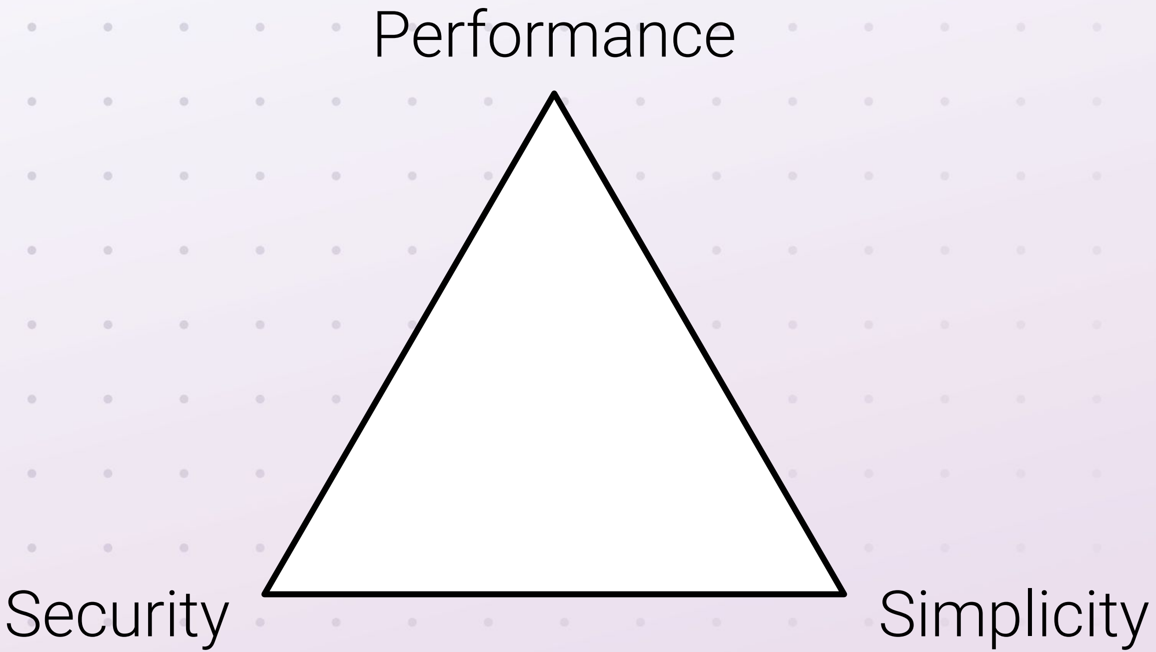
And many more

- Comments
- Logging
- Generics
- First-class functions
- Dynamic array indexing
- …

The Noir Programming Language

- **Compiles down to an IR**
  - Abstract Circuit Intermediate Representation (ACIR)
  - The IR can then be compiled down to any NP complete language
- **Backend Agnostic**
  - Fully integrated with TurboPlonk
  - Extendable to Groth16, Halo2, Gnark, etc.
  - Enables proving system optimizations
    - Custom gates
    - Efficient black box functions (e.g. ECDSA, Keccak, SHA256)

The Perfect Balance



Additional Features

That the Noir tech stack offers

Tests

in Noir

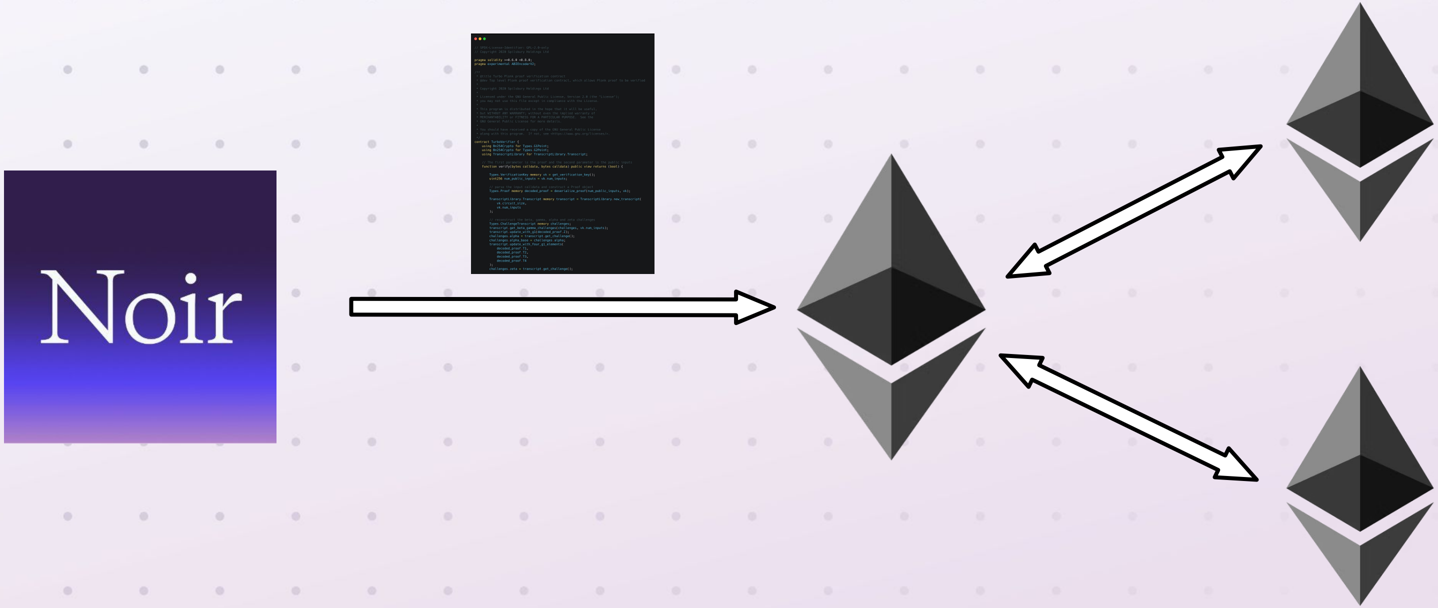
```
fn add(x: u64, y: 64) -> u64 {
    x + y
}

#[test]
fn test_add() {
    constrain add(2,2) == 4;
    constrain add(0,1) == 1;
    constrain add(1,0) == 1;
}
```

Q&A Pause

Noir in Aztec 3

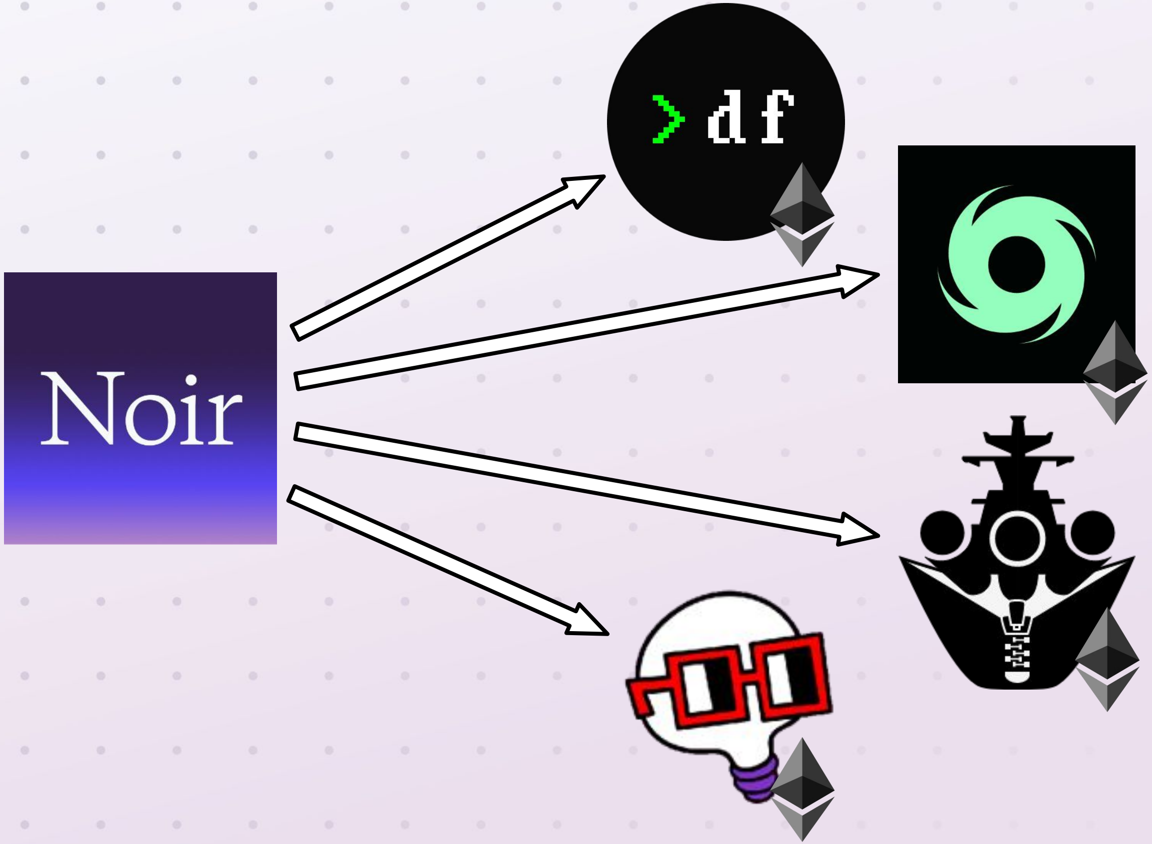
ZK dApp Today



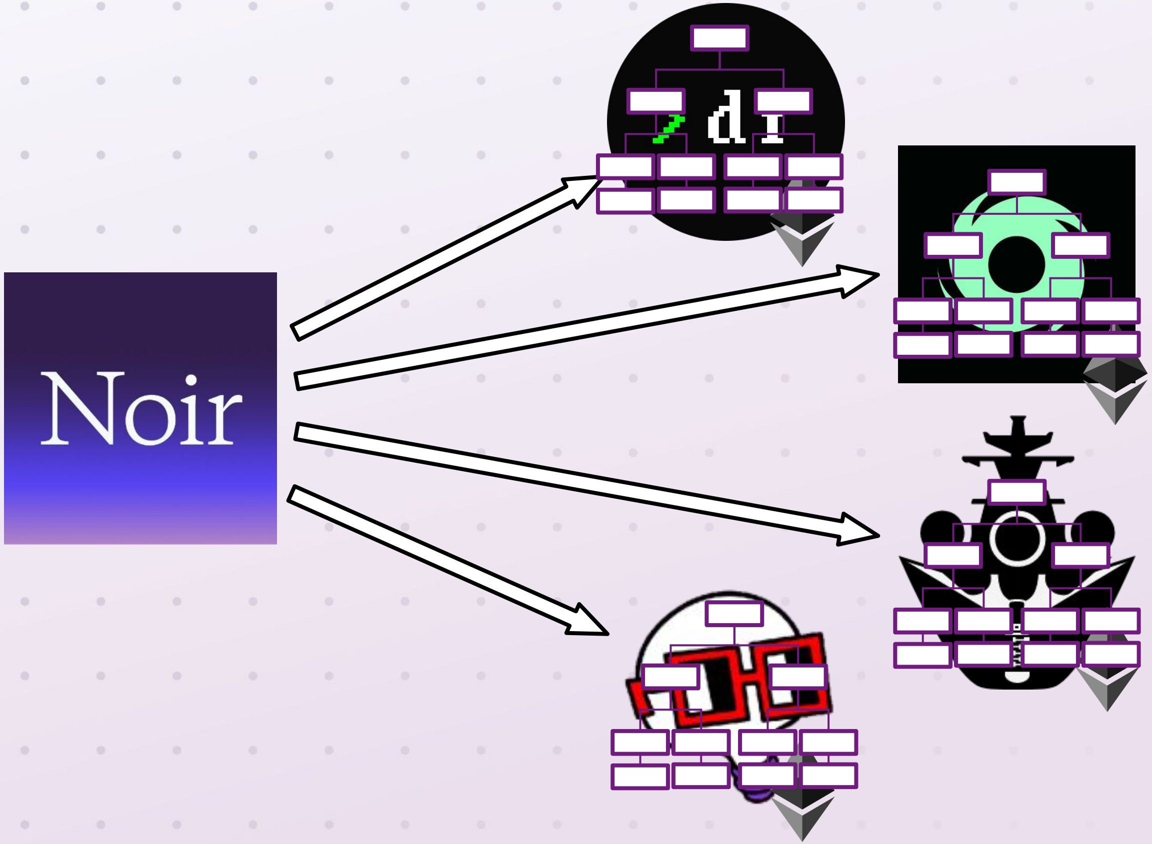
Problem?

Aztec

ZK dApp Today



ZK dApp Today

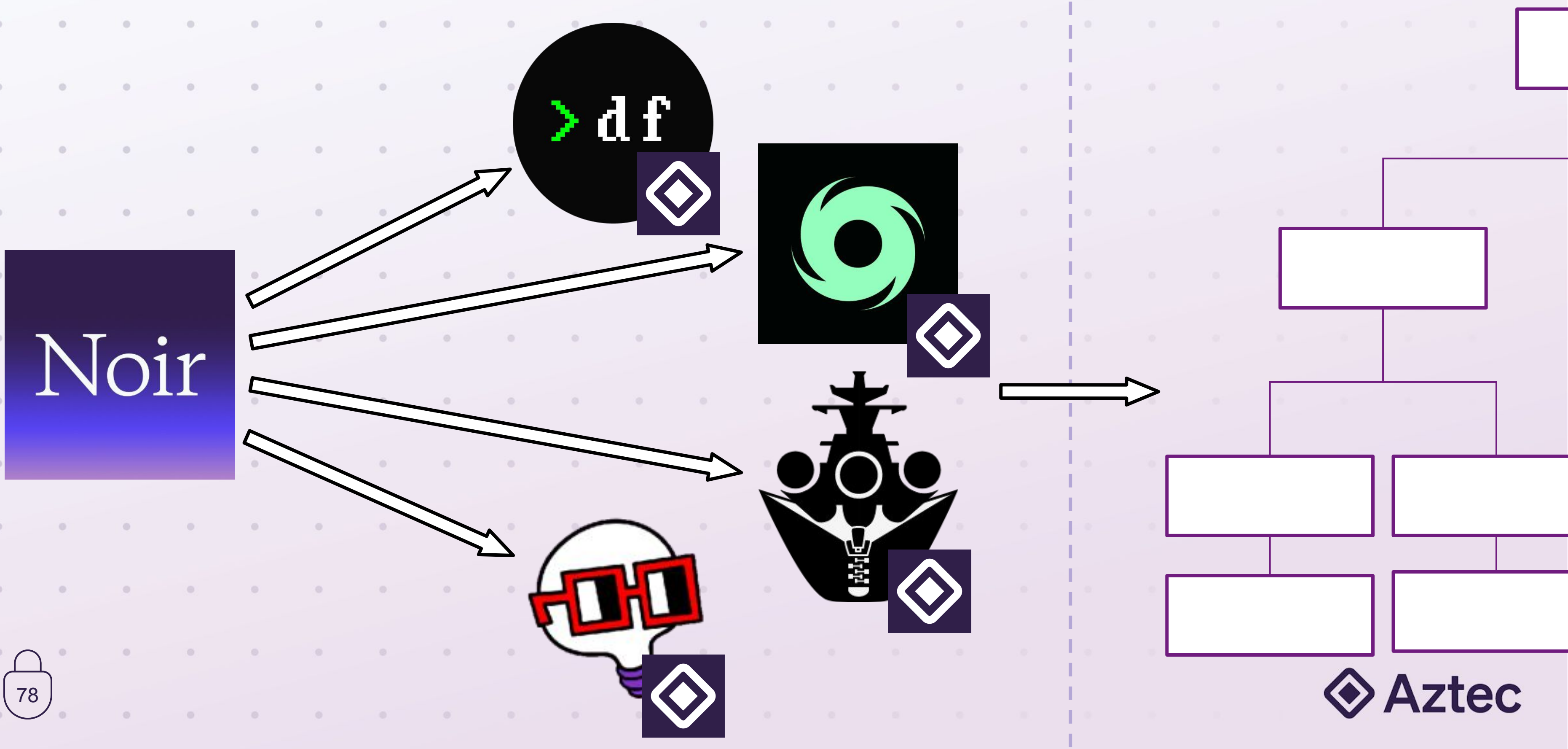


Aztec

What if…


NOIR CONTRACTS
NOIR DEV
PRIVATE
STATE TREE +
NULLIFIER TREE +
SOLIDITY LOGIC
imgflip.com

Noir Smart Contract



Noir Smart Contract

This is soon™ Noir

This is Aztec 3

Aztec 3

- **ZK-ZK Rollup**
    - Inherits Ethereum's security
    - Supports both private and public states
    - Fully programmable
    - Lower gas fees

- Noir → private + public smart contracts
- Aztec 3 → a decentralized storage and execution backend for Noir

Aztec

Non-determinism

Aztec

Directives

- Do not apply any constraints
- Can be thought of as an oracle
  - Fetch extra external inputs to the circuit
  - Allow one to use non-determinism
- Noir wants to prove some computation is correct
  - Not just perform the computation!
  - But it is not enough to simply give the result of the program
- In many cases, non-determinism provides a neat shortcut, but must be handled correctly

Directives Explained

Let's decompose a Field element into an array of bytes

let x: Field = 1000;

let byte_array = x.to_le_bytes(); // 0x03E8

- Straightforward approach
  - Convert x into a u32
  - Perform bit operations to find each element of the array

```
for i in 0..FIELD_SIZE_IN_BYTES {
  let arr[i] = (x >> (8*i)) & 0xff;
}
```

Directives Explained

Let's decompose a Field element into an array of bytes

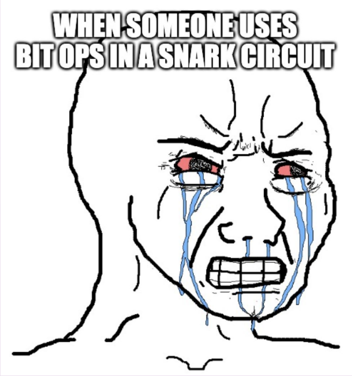
let x: Field = 1000;

let byte_array = x.to_le_bytes(); // 0x03E8

- Straightforward approach
  - Convert x into a u32
  - Perform bit operations to find each element of the array

EXPENSIVE
&
INCOMPLETE

```
for i in 0..FIELD_SIZE_IN_BYTES {
  let arr[i] = (x >> (8*i)) & 0xff;
}
```



Directives Explained

Let's use a shortcut instead!

- When generating the ACIR we will lay down the following arithmetic constraints

```
let x: Field = 1000;
constrain [(arr[0]*2^0) + (arr[1]*2^8) + … + (arr[31]*2^248)] - x == 0;
```

- Do we see a problem with this pseudocode?
  - How is the expression viewed by the prover?
  - How is the expression viewed by the verifier?

Directives Explained

This is non-determinism!

constrain (arr[0]*2^0) + (arr[1]*2^8) + … + (arr[31]*2^248) - x == 0;

- The prover does not know what values of `arr` are implicitly
  - arr[0] could be 1000 and the rest of the array values are 0
  - OR the array could hold a valid byte array
- The prover must inject the correct values into the arithmetic constraint above
- Inside the directive we decompose the Field element `x` into a byte array
  - Fill in the witness values specified from ACIR generation
  - DO NOT lay down any new constraints

Unconstrained Functions

Extended Noir to Aztec 3

- Add full non-determinism to Noir
  - Users will be able to run unconstrained code
  - These functions execute outside the circuit, similarly to our existing directives
  - Users must manually constrain outputs of unconstrained circuits, using a noir program
  - We need to match values in the noir program with values from unconstrained code

Extending ACVM to Aztec 3 VM

- Unconstrained code
  - Can be added within the Noir program using unsafe blocks: clean and safer!
  - Can be handled by the A3 Simulator: more flexible (can do db queries, call web API, etc..)

```
secret fn transfer(amount: Field, to: Field) {

    unconstrained fn reduce<NUM_NOTES: u32>(all_notes: [Note]) -> [Note; NUM_NOTES] {
        // Imagine some code which reduces an unbounded array to a static-array
        // ... and also ensures note.owner == msg.sender;
    }

    let my_old_notes: [Note; 2] = balances[msg.sender].get( // UTXOSet::get
        2, // number of notes requested
        reduce // <-- the unconstrained function defined immediately above.
    );

    // Now constrain the `reduce` logic:
    constrain my_old_notes.all(|n| n.owner_or(msg.sender) == msg.sender);
```

Links & Resources

GitHub

Docs

Awesome Noir

Reference Slides

The Noir Programming Language

- **Generics**
- First-class functions

```
struct Bar<T> {
    one: Field,
    two: Field,
    other: T,
}

fn foo<T>(bar: Bar<T>) {
    constrain bar.one == bar.two;
}

fn main(x: Field, y: Field) {
    let bar1: Bar<Field> = Bar { one: x, two: y,
other: 0 };
    let bar2 = Bar { one: x, two: y, other: [0] };

    foo(bar1);
    foo(bar2);
}
```

The Noir Programming Language

- Generics
- **First-class functions**

```
let f = if 3 * 7 > 200 { foo } else { bar };
constrain f()[1] == 2;

// Lambdas:
constrain twice(|x| x * 2, 5) == 20;
constrain (|x, y| x + y + 1)(2, 3) == 6;

// Closures:
let a = 42;
let g = || a;
constrain g() == 42;

fn foo() -> [u32; 2] {
    [1, 3]
}

fn bar() -> [u32; 2] {
    [3, 2]
}

fn twice(f: fn(Field) -> Field, x: Field) -> Field {
    f(f(x))
}
```


Aztec

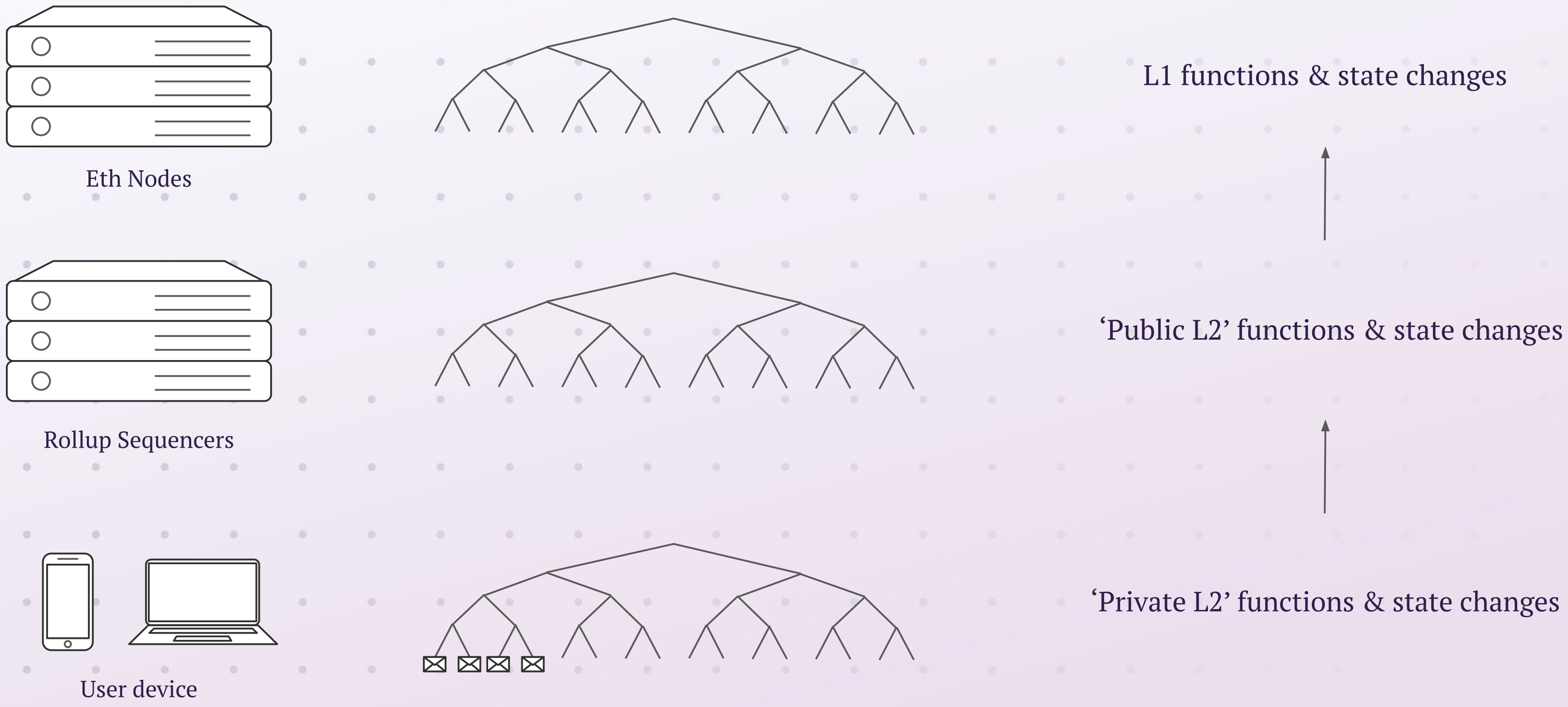
Public and Private State

- Public state
  - **Account model**
  - Updateable public data tree
  - Managed by **rollup sequencers**
- Private state
  - **UTXO model**
  - Append-only private data tree
  - Append-only “indexed” nullifier tree
  - Membership proofs on **user-devices**

Aztec

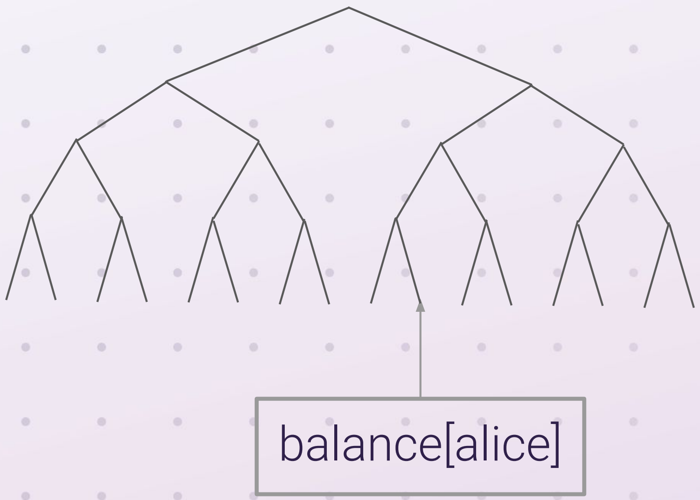
Noir → Smart Contracts

- Public and private storage reads/writes
- Contract scopes and silos
- Contract functions as independent ZK circuits
- Nested calls to other functions/circuits
- Calls to L1 portal contracts


Eth Nodes
Rollup Sequencers
User device
L1 functions & state changes
‘Public L2’ functions & state changes
‘Private L2’ functions & state changes

Public State

Public Data Tree



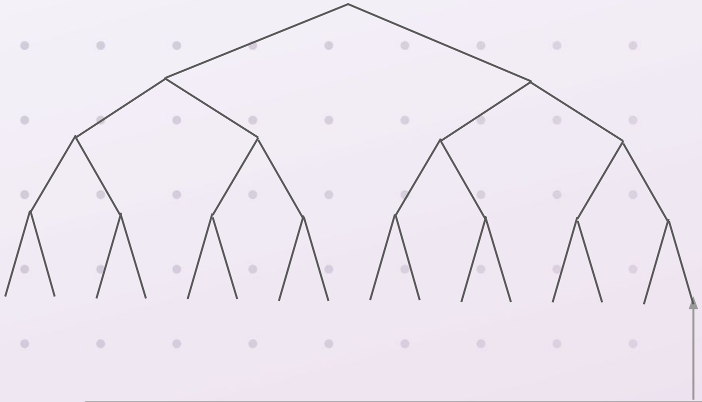
Just a value in a tree.

Accessed and updated by its key.

key: hash(contractAddress, storageSlot)

Private State

Private Data Tree



A commitment to a value in a tree.

Inserted in next available slot.

Similar for nullifier tree…

hash(contractAddress, hash(storageSlot, value, owner, creator, memo, salt, inputNullifier)

Function Trees

- Each contract has a function tree
- A Noir *contract* function compiles to
  - ACIR Opcodes
  - Circuit
  - Verification key
- Function tree leaf
  - functionSelector
  - isPrivate
  - vkHash

Aztec

Contracts Tree

- One leaf per contract
- Append-only
- Contract tree leaf contains
    - contractAddress
    - portalContractAddress
    - functionTreeRoot

Aztec

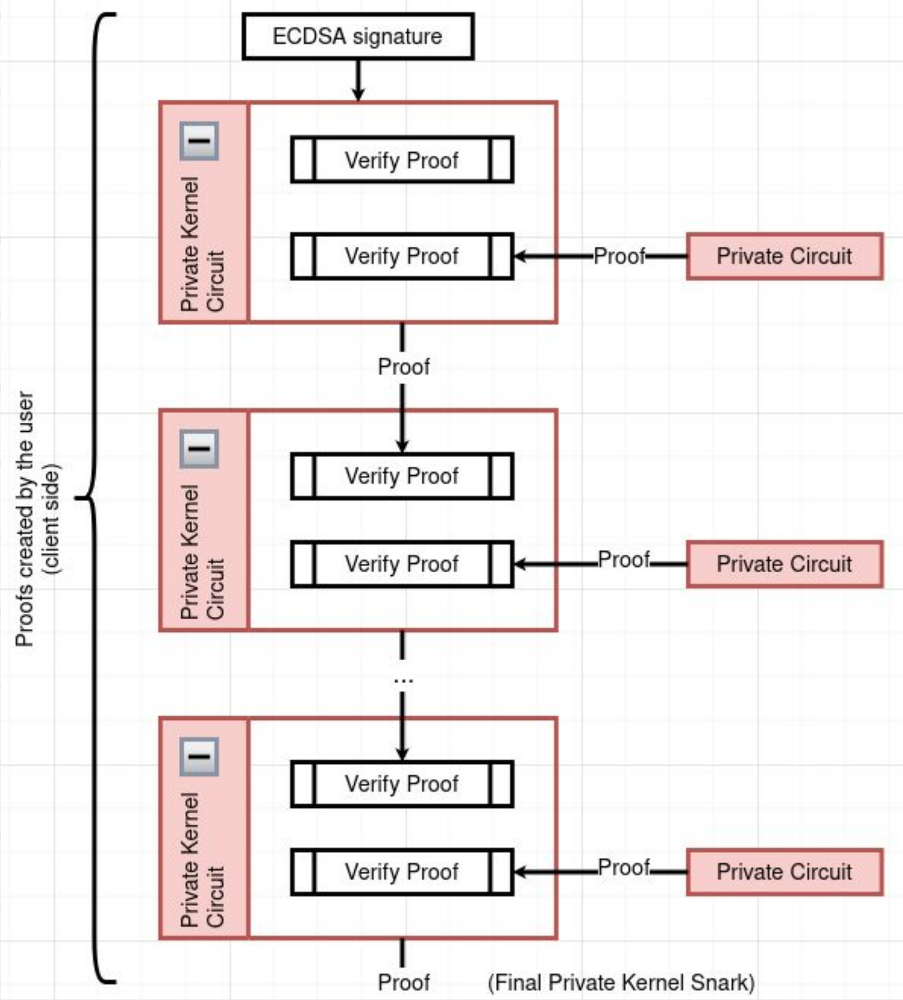
Kernel Circuits

- Stitch together nested function calls
- Call stack / context as public inputs
- Check function against functions tree
- Check contract against contracts tree
- Check state accesses against data trees

Aztec

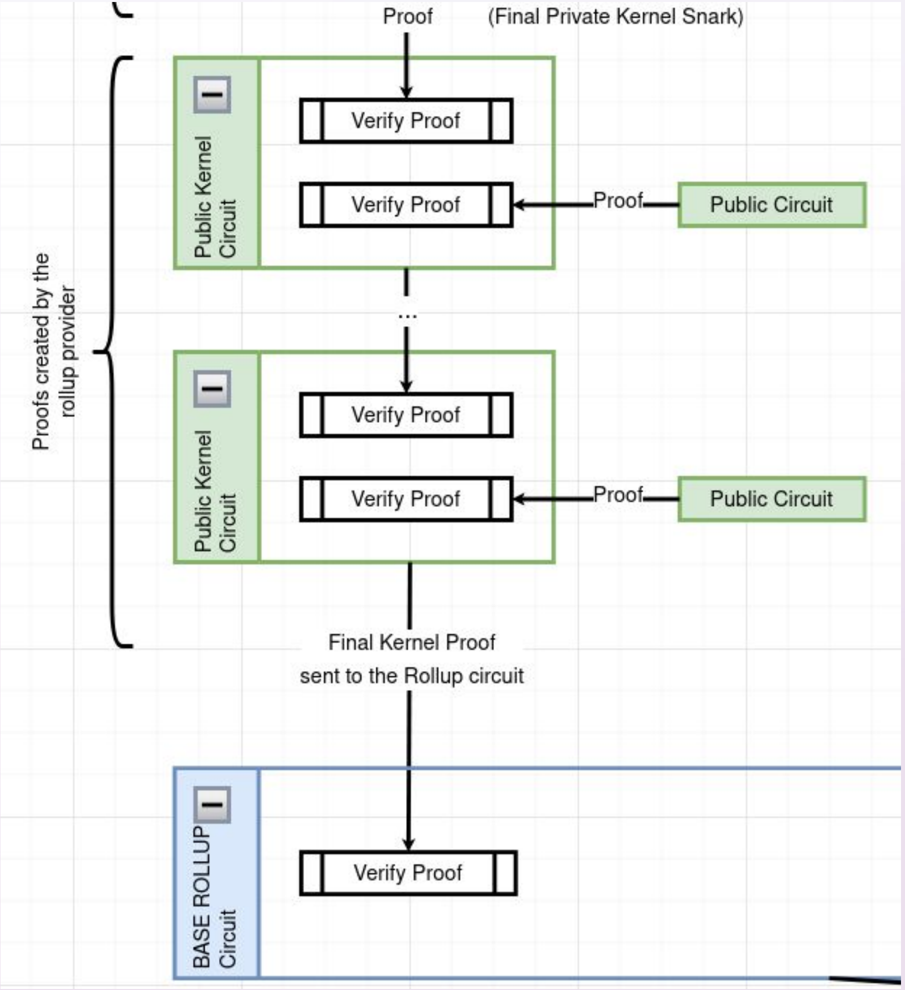
Private Kernel Circuit

- Proofs created by user
- Preserves privacy of
  - TX origin / sender
  - Function arguments
  - State accesses
- Stitch together nested private function calls
- Final proof and outputs sent to sequencer as input to public kernel



Public Kernel Circuit

- Proofs created by sequencer (actually prover)
- Accept private kernel results as input
- Preserves privacy of
  - TX origin / sender
  - All private kernel info
- Stitch together nested public function calls
- Handle L1 portal calls
- Final proof and outputs rolled up



New Noir Opcodes

- UTXO_SLOAD, UTXO_NULL, UTXO_SSTORE
- PRIV_FUNC_CALL, PUB_FUNC_CALL, L1_FUNC_CALL
- ADDRESS, PORTAL_ADDRESS
- DELEGATE_CALL, STATIC_CALL
- REVERT
- RAND
- others…

Aztec