Lesson 8 - Noir and Warp

Warp

Warp allows you transpile Solidity contracts into Cairo

Installation Instructions

See Warp installation instructions

Note WARP is still using Cairo v 0.11
Cairo v 1.0 development is on this branch

Using Warp

`warp transpile example_contracts/ERC20.sol`

`warp transpile example_contracts/ERC20.sol --compile-cairo`

You can then deploy your cairo code to the network, with the following commands you need to specify the network, in our case alpha-goerli

```
  warp deploy ERC20.json --network alpha-goerli
```

```
  Deploy transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash: 0x32ca42d1341703cc957845ea53a71b3eb2e762ff148cb9dc522322eede94b65
```

You can invoke a transaction on your contract

```
warp invoke --program test.json --address
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a  --network
alpha-goerli --function store --inputs [13]
```

```
  Invoke transaction was sent.
Contract address:
0x0403bd2f0abdd765398d6a50ff89cfe9ac48760f3b94ba2728bfbacdaff9f59a
Transaction hash: 0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881
```

And check the status

```
  warp status 0x1d1ec8278ccf41452737e80a54e7626299e598528363ced7a527d810f9d6881 --
network alpha-goerli
```

which will give a answer similar to

```
        {
                "block_hash":
"0x1c55254f16d087f0bf7776183c4d38549680e68600394167f304f1afe5a035e",
                "tx_status": "ACCEPTED_ON_L1"
        }
```

You should be able to see the details on the block explorer
[Voyager Block Explorer]

There is also now a [vyper transpiler]

Aztec updates

Sunsetting Aztec Connect

Aztec Connect allows users to bridge private assets to mainnet for a DeFi interaction and return to Aztec in the same transaction.

Aztec Connect serves as a bridge to Ethereum, allowing users to bring privacy-shielded zk-assets on Aztec to public DeFi protocols on Ethereum.

See [Medium Article]

Timeline

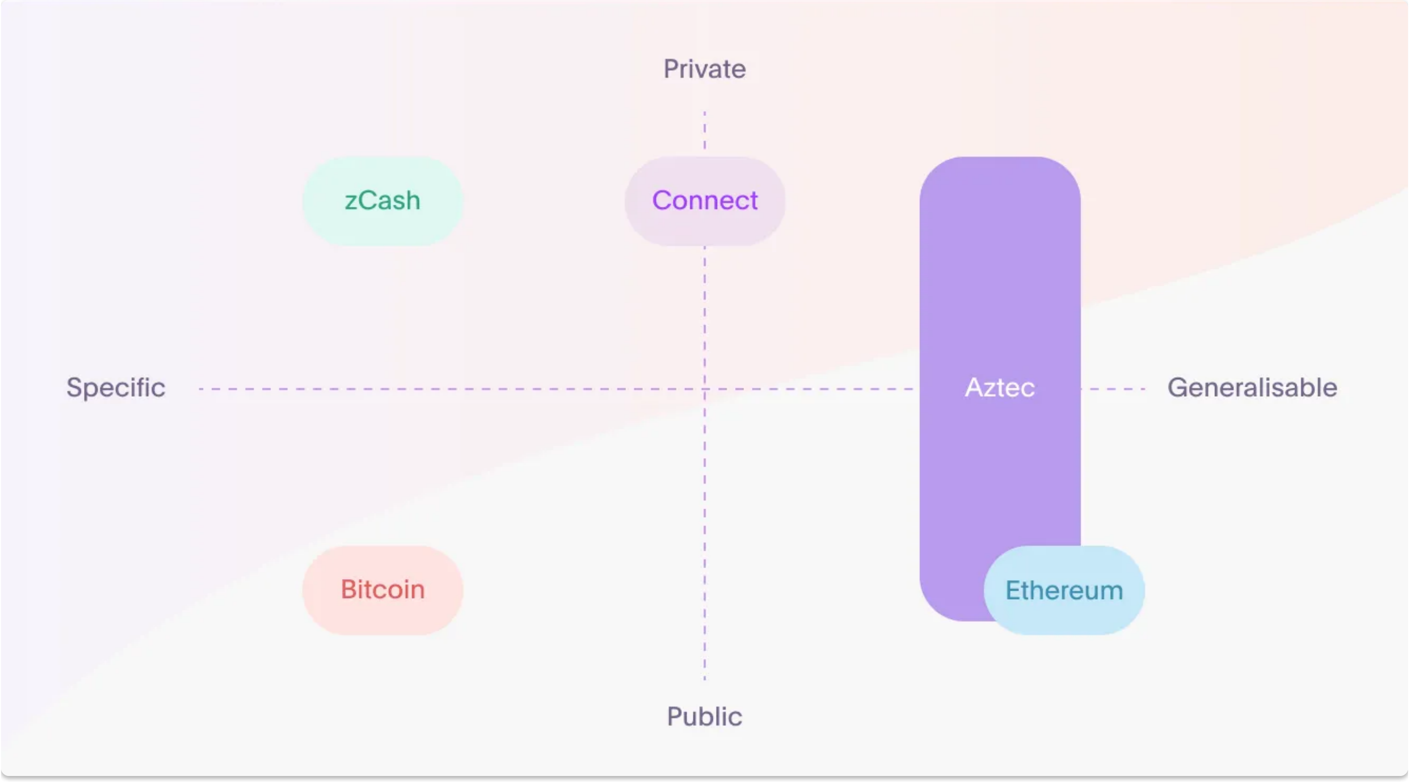
- **13th — 21st March:** Aztec Connect normal functionality
- **21st March 2023 at 2PM UTC:** Deposits disabled
- One-year withdrawal window (March 13th, 2023 — March 21st, 2024)
- **March 21st, 2024:** Last Aztec rollup sent from Aztec sequencer
- **March 21st, 2024 onwards:** users will have to run withdrawal software or rely on community-run sequencers

The Aztec Connect Repo has been open sourced
[Repo]

"We encourage the Aztec community to fork, deploy, and operate a new version of the system. We'd love to see an independently-operated Aztec Connect and are ready to fund it. Head to our open-source repo and docs to learn more about running an Aztec Connect fork."

Aztec (the new one)

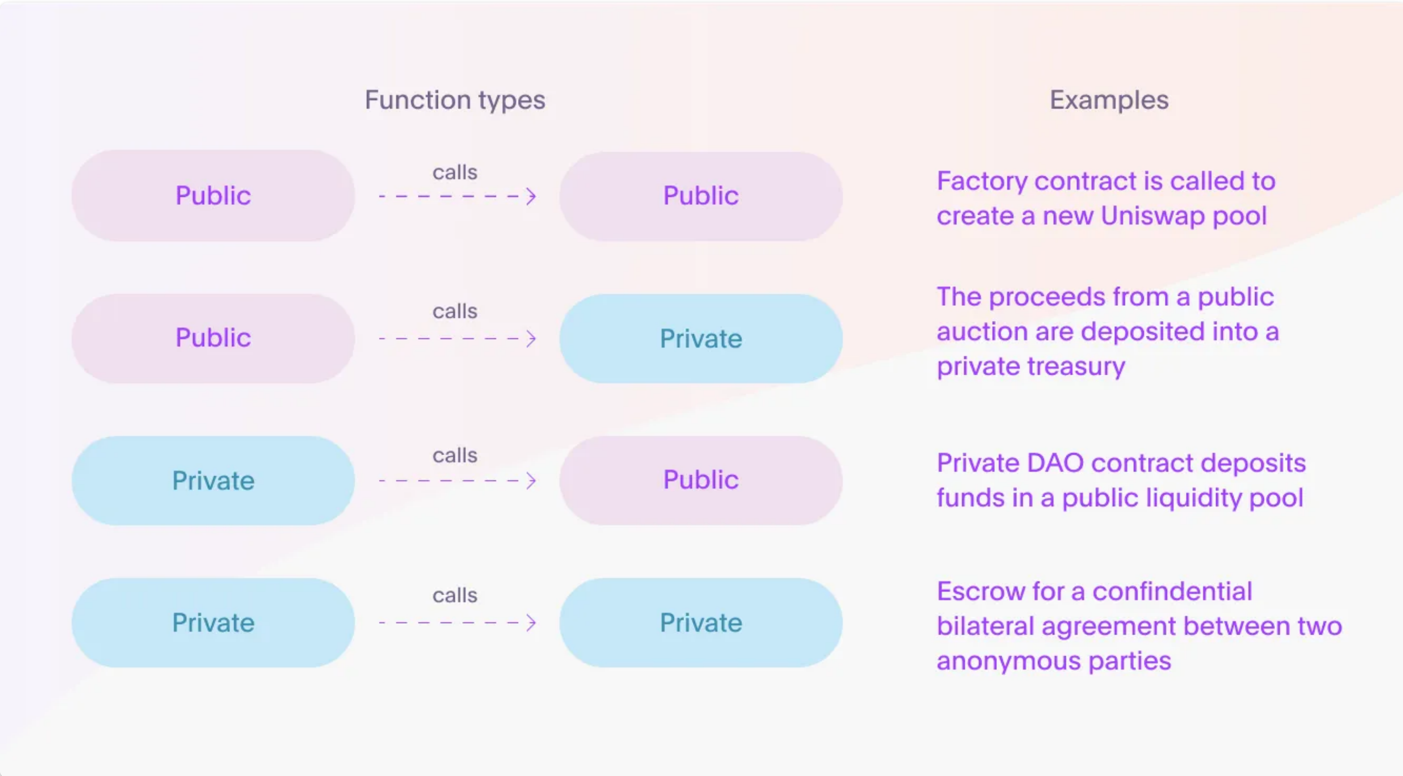
See [Blog post]
Ethereum is a state machine in which the state is public
Aztec aims to allow private state (in general)



This hybrid public-private state machine processes both public and private execution of smart contract logic. Simultaneously, it no longer executes transactions like a conventional blockchain, instead proving the correctness of transactions that have already been computed.

Note that it is not a zkEVM approach (more of which in a future lesson)

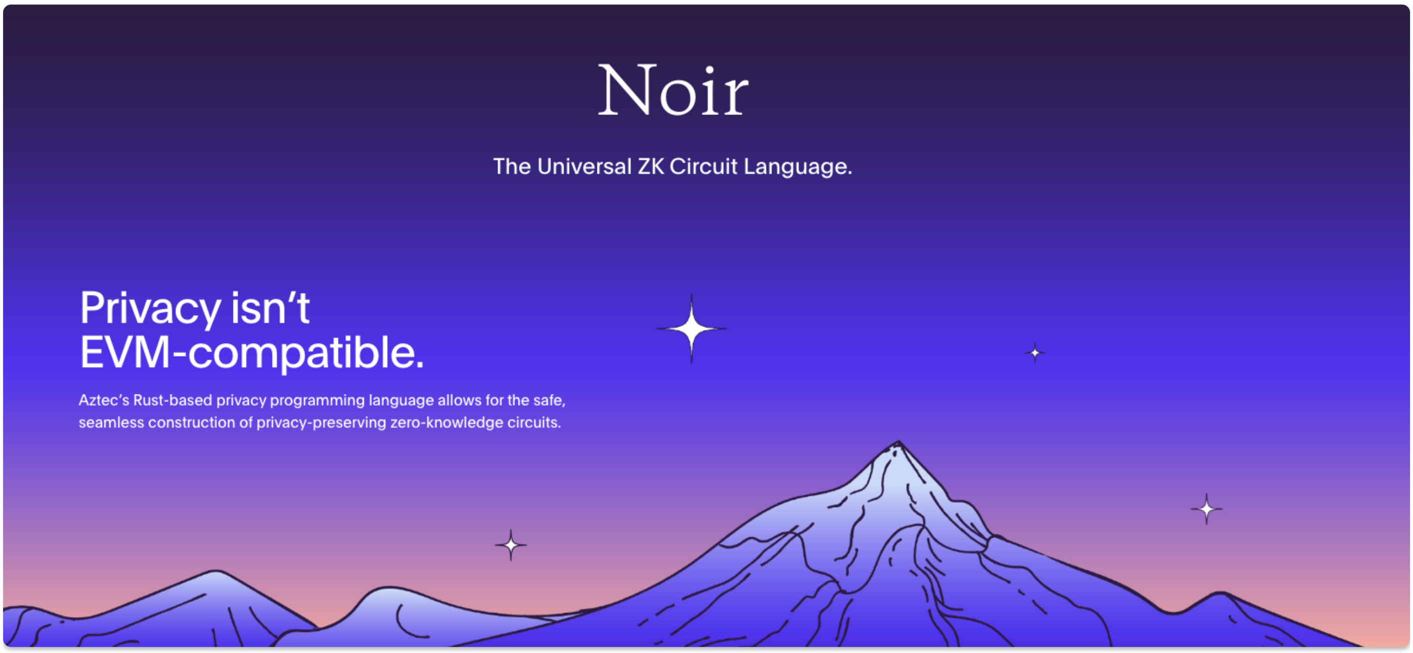
Aztec builds on the work done with Aztec connect, in that a UTXO represented some value, in Aztec the UTXO can represent a contract.



Timeline

Aztec will target 100+ TPS and single-digit cent transactions by mainnet launch.

Noir



Introduction

Noir is an open-source, generalized zero knowledge circuit writing language compatible with any proving back-end and verifiable on any blockchain with customizable smart contract verifiers.

Noir Language

Noir is a domain specific language for creating and verifying proofs. Design choices are influenced heavily by Rust.

Noir is much simple and flexible in design as it does not compile immediately to a fixed NP-complete language. Instead Noir compiles to an intermediate language which itself can be compiled to an arithmetic circuit or a rank-1 constraint system.

From Documentation

Noir can be used for a variety of purposes.

Ethereum Developers

Noir currently includes a command to publish a contract which verifies your Noir program. This will be modularised in the future, however as of the alpha you can use the `contract` command to create it.

Protocol Developers

As a protocol developer, you may not want to use the Aztec backend due to it not being a fit for your stack or maybe you simply want to use a different proving system. Since Noir does not compile to a specific proof system, it is possible for protocol developers to replace the PLONK based proving system with a different proving system altogether.

Blockchain developers

As a blockchain developer, you will be constrained by parameters set by your blockchain, ie the proving system and smart contract language has been pre-defined. In order for you to use Noir in your blockchain, a proving system backend must be implemented for it and a smart contract interface must be implemented for it.

Resources

[Awesome-noir]

Features

Backends:

- Barretenberg via FFI
- Marlin via arkworks

Compiler:

- Module System
- For expressions
- Arrays
- Bit Operations
- Binary operations `(<, <=, >, >=, +, -, *, /, %)` [See documentation for an extensive list]
- Unsigned integers
- If statements
- Structures and Tuples
- Generics

ACIR Supported OPCODES:

- Sha256
- Blake2s
- Schnorr signature verification
- Merkle Membership
- Pedersen
- HashToField

You can create a solidity verifier contract automatically

Installation

See Docs

IDEs

- Noir Editor - Browser IDE
- VSCode Extension - Syntax highlight
- Vim Plugin - Syntax highlight
- hardhat-noir - Hardhat plugin

Building the constraint system

`nargo check`

Add the inputs

In the created prover.toml add values for the inputs to the main function

Create the proof

`nargo prove p`

Verify the proof

`nargo verify p`

Language

Private & Public Types

```
  fn main(x : Field, y : pub Field) -> pub Field {
    x + y
}
```

All data types in Noir are private by default. Types are explicitly declared as public using the `pub` modifier
Note: Public types can only be declared through parameters on `main`.

Mutability

Mutability is possible with the use of the `mut` keyword.

```
  let mut y = 3;
let y = 4;
```

Note Mutability is local and everything is passed by value, so if a called function mutates its parameters then the parent function will keep the old value of the parameters.

```
  fn main() -> Field {
          let x = 3;
          helper(x);
          x // x is still 3
  }

  fn helper(mut x: i32) {
          x = 4;
  }
```

language-rust

Primitive Types

Field
Integer
Boolean
String

Field

The field type corresponds to the native field type of the proving backend.

Fields support integer arithmetic and are the optimal numeric type,

language-rust

```
fn main(x : Field, y : Field)  {
        let z = x + y;
}
```

Integer

An integer type is a range constrained field type. The Noir frontend currently supports unsigned, arbitrary-sized integer types.

An integer type is specified first with the letter `u`, indicating its unsigned nature, followed by its length in bits (e.g. `32`).
For example, a `u32` variable can store a value in the range of `([0,` $2^{32}-1$ `])`:
Using the integer datatype rather than a field requires additional range proofs.

Boolean

The `bool` type in Noir has two possible values: `true` and `false`:

```
fn main() {    let t = true;    let f: bool = false;}
```

String

The string type is a fixed length value defined with `str<N>`.

You can use strings in `constrain` statements or print them with `std::println()`.

```
fn main(message : pub str<11>, hex_as_string : str<4>) {
std::println(message);    constrain message == "hello world";    constrain
hex_as_string == "0x41";}
```

Compound Types

Array

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

Tuple

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

Struct

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

fn main() {
    let legs = 4;

    let dog = Animal {
        eyes: 2,
        hands: 0,
        legs,
    };

    let zero = dog.hands;
}
```

De structuring Structs

```
fn main() {
    let Animal { hands, legs: feet, eyes } = get_octopus();

    let ten = hands + feet + eyes as u8;
}

fn get_octopus() -> Animal {
    let octopus = Animal {
        hands: 0,
        legs: 8,
        eyes: 2,
    };

    octopus
}
```

You can also define methods within a struct

```
struct MyStruct {
    foo: Field,
    bar: Field,
}

impl MyStruct {
    fn new(foo: Field) -> MyStruct {
        MyStruct {
            foo,
            bar: 2,
        }
    }

    fn sum(self) -> Field {
        self.foo + self.bar
    }
}

fn main() {
    let s = MyStruct::new(40);
    constrain s.sum() == 42;
}
```

You could also do

```
constrain MyStruct::sum(s) == 42
```

Comptime values

These are values which are known at compile time.
They are declared with the `comptime` keyword

```rust
fn main() {
let a: comptime Field = 5;

// `comptime Field` can also be inferred:
let a = 5;
}
```

Comptime variables cannot be declared mutable

Global variables

We can also use the `global` keyword to declare comptime variables,(you do not need to also use the `comptime` keyword).

Globals are currently limited to Field, integer, and bool literals.
For example

```rust
global N: Field = 5; // Same as `global N: comptime Field = 5`

fn main(x : Field, y : [Field; N]) {
        let res = x * N;

        constrain res == y[0];

        let res2 = x * mysubmodule::N;
        constrain res != res2;
}

mod mysubmodule {
use dep::std;

        global N: Field = 10;

        fn my_helper() -> comptime Field {
                let x = N;
                x
        }
}
```

Functions

`fn` keyword

```rust
fn foo(x : Field, y : pub Field) -> Field {
        x + y
}
```
language-rust

Loops

only for loops are possible

```rust
for i in 0..10 {
    // do something
};
```
language-rust

Recursion is not yet possible

If expressions

```rust
let a = 0;
let mut x: u32 = 0;

if a == 0 {
    if a != 0 {
        x = 6;
    } else {
        x = 2;
    }
} else {
    x = 5;
    constrain x == 5;
}
constrain x == 2;
```
language-rust

Constrain / Assert Statement

Constrain has been deprecated, use the assert statement instead.

```
  fn main(x : Field, y : Field) {
    assert(x == y);
}
```

`assert` which will explicitly constrain the predicate/comparison expression that follows to be true. If this expression is false at runtime, the program will fail to be proven.

Noir Standard Library

See Documentation

Cryptographic Functions

- sha256
- blake2s
- pedersen
- poseidon
- mimc_bn254 and mimc
- scalar multiplication
- schnorr signature verification
- elliptic curve data structures and primitives

Array functions

- len
- sort
- sort_via
- map
- fold
- reduce
- all / any

Field functions

- bytes conversion
- vector conversion
- power

Logging

There is a version of rust's `println!` macro, this can be used for fields, integers and arrays (including strings).
To view the output of the `println` statement you need to set the `--show-output` flag when using nargo

```rust
use dep::std;

fn main(string: pub str<5>) {
        let x = 5;
        std::println(x)
}
```

Merkle Trees

- check membership

- compute root

ACIR

Noir comples to [ACIR (Abstract Circuit Intermediate Representation)](#) which later can compile to any ZK proving system.
The purpose of ACIR is to act as an intermediate layer between the proof system that Noir chooses to compile to, and the Noir syntax.
This separation between proof system and programming language, allows those who want to integrate proof systems to have a stable target.

Compiling a proof

When inside of a given Noir project the command `nargo compile my_proof` will perform two processes.

- First, compile the Noir program to its ACIR and solve the circuit's witness.
- Second, create a new `build/` directory to store the ACIR, `my_proof.acir`, and the solved witness, `my_proof.tr`

These can be used by the Noir Typescript wrapper to generate a prover and verifier inside of Typescript rather than in Nargo.

UI

https://github.com/noir-lang/noir-web-starter-next

Solidity Verifier

You can create a verifier contract for your Noir program by running:

`nargo contract`

A new `contract` folder would then be generated in your project directory, containing the Solidity file `plonk_vk.sol`. It can be deployed on any EVM blockchain acting as a verifier smart contract.

> ***Note:*** *It is possible to compile verifier contracts of Noir programs for other smart contract platforms as long as the proving backend supplies an implementation.*
>
> *Barretenberg, the default proving backend Nargo is integrated with, supports compilation of verifier contracts in Solidity only for the time being.*

Roadmap

Concretely the following items are on the road map:

- Prover and Verifier Key logic. (Prover and Verifier pre-process per compile)
- Fallback mechanism for backend unsupported opcodes
- Visibility modifiers
- Signed integers
- Backend integration: (Bulletproofs)
- Recursion
- Big integers

References

https://aztec.network/noir/
Developer Docs
https://docs.aztec.network/

Resources
https://github.com/noir-lang/awesome-noir

Noir Book
https://noir-lang.github.io/book/

Noir Examples

Mastermind

```
  use dep::std;

fn main(
    guessA: pub u4,
    guessB: pub u4,
    guessC: pub u4,
    guessD: pub u4,
    numHit: pub u4,
    numBlow: pub u4,
    solnHash: pub Field,
    solnA: u4,
    solnB: u4,
    solnC: u4,
    solnD: u4,
    salt: u32
) {
    let mut guess = [guessA, guessB, guessC, guessD];
    let mut soln = [solnA, solnB, solnC, solnD];

    for i in 0..4 {
        let mut invalidInputFlag = 1;
        if (guess[i] > 9) | (guess[i] == 0) {
            invalidInputFlag = 0;
        }
        if (soln[i] > 9) | (soln[i] == 0) {
            invalidInputFlag = 0;
        }
        constrain invalidInputFlag == 1;
        for j in (i+1)..4 { // Check that the guess and solution digits are unique
            constrain guess[i] != guess[j];
            constrain soln[i] != soln[j];
        };
    };

    let mut hit: u4 = 0;
    let mut blow: u4 = 0;

    for i in 0..4 {
        for j in 0..4 {
            let mut isEqual: u4 = 0;
            if (guess[i] == soln[j]) {
```

```
                isEqual = 1;
                blow = blow + 1;
            }
            if (i == j) {
                hit = hit + isEqual;
                blow = blow - isEqual;
            }
        };
    };

    constrain numBlow == blow;

    constrain numHit == hit;

    let privSolnHash = std::hash::pedersen([salt as Field, solnA as Field, solnB as Field, solnC as Field, solnD as Field]);

    constrain solnHash == privSolnHash[0];
}
```

Tornado Cash Example - see repo

```
  use dep::std;

fn main(
  recipient : 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
) -> pub [Field; 2] {
    // 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]);

    // Check that the input note commitment is in the root
    let is_member = std::merkle::check_membership(note_root, note_commitment[0],
index, note_hash_path);
    constrain is_member == 1;

    // Cannot have unused variables, return the recipient as public output of the
circuit
    [nullifier[0], recipient]
}
```

Noir minimal Template

See Repo

Use template will bring up the template in a codespace