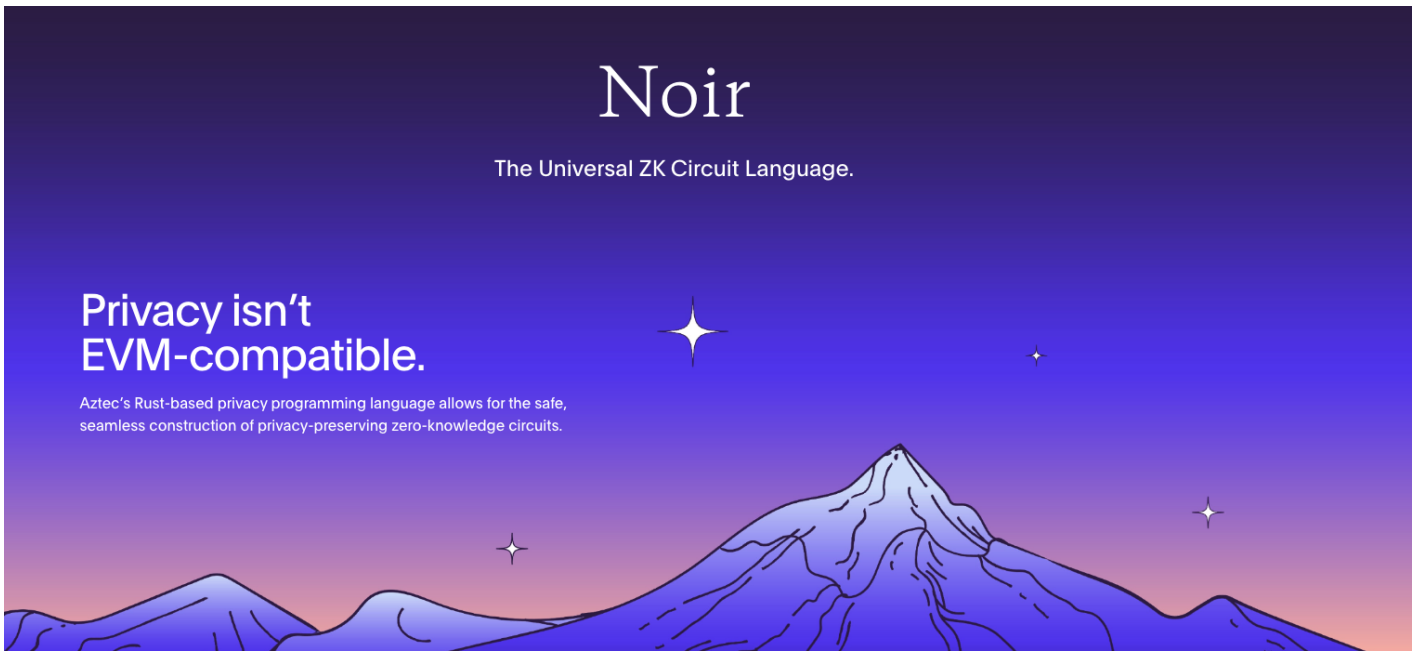
Lesson 7

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.

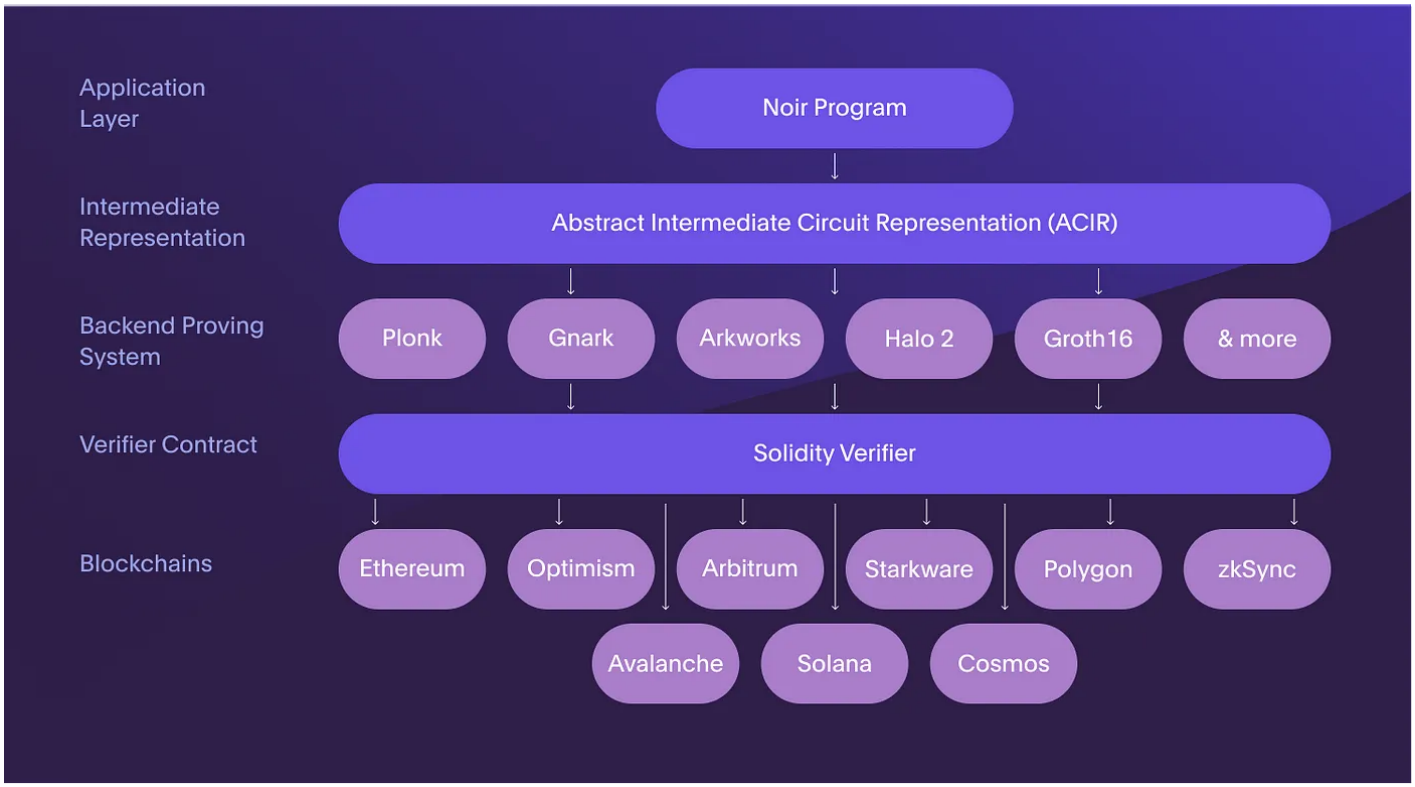
Noir Language

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.



Resources

Awesome-noir

Features

Backends:

- Barretenberg via FFI
- Marlin via arkworks

Compiler:

- Module System
- For expressions
- Arrays
- Bit Operations
- Binary operations `(<, <=, >, >=, +, -, *, /, %)`
- 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;
}
```

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,

```
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 `assert` statements or print them with `std::println()`.

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

```
        assert 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);
    assert s.sum() == 42;
}
```

You could also do

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

Global variables

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

For example

```
global N: Field = 5;

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

    assert(res == y[0]);

    let res2 = x * mysubmodule::N;
    assert(res != res2);
}

mod mysubmodule {
    use dep::std;

    global N: Field = 10;

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

Functions

`fn` keyword

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

Lambda functions / Anonymous functions

See Docs

A recent addition to Noir, these allow functions without a name, for example

```
let add_50 = |val| val + 50;
assert(add_50(100) == 150);
```

The general format is `|arg1, arg2, ..., argN|` followed by a code block.

Closures are also possible, see Closures

Loops

only for loops are possible

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

Recursion is not yet possible

If expressions

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

if a == 0 {
    if a != 0 {
        x = 6;
    } else {
        x = 2;
    }
} else {
    x = 5;
    assert(x == 5);
}
assert(x == 2);
```

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

```
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.

Unconstrained Functions / non deterministic code

See article and docs

The above article describes the role of constraints in Noir. Remember that the proof we produce is a proof that our program executed correctly. From a business logic perspective however, we are really concerned with more than that, we want to show that a user knew a particular secret, or had more than a certain balance.
To achieve that, we add constraints to our code.
An unconstrained function is closer to traditional programs, where we are more concerned with a result, and less with showing the integrity of execution.

Unconstrained functions however reduce what needs to be in the circuit, and can therefore act as an optimisation.
The trick is to know when and how to use unconstrained functions, the approach is

1. Use an unconstrained function to produce a result, this is probably a correct result, but we

have no guarantee of that, it is almost like making a guess.

2. We can then take the result and by using a constrained function show that it is correct.
   If both functions were equally costly, there would be no benefit to this, but if we can write a cheap constrained function, then this could be beneficial.

For example, division is costly in a circuit, but multiplication is cheap.
Therefore if we have to calculate
z = x / y we can

1. Do the division outside a circuit (un constrained) to give our answer z
2. Write a constrained function to show that y * z = x This function, though constrained is 'cheap' since it is using multiplication.

Recursive Proofs

Noir supports recursively verifying proofs, meaning you verify the proof of a Noir program in another Noir program. This enables creating proofs of arbitrary

size by doing step-wise verification of smaller components of a large proof.

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 as of June 2023

- Plookup Dynamic Arrays

  Enable dynamic arrays to make use of RAM/ROM opcodes when supported by the proving backend of choice, heavily optimizing constraint counts hence proving speed of dynamic arrays.
- Recursion

  Enable recursive proofs when supported by the proving backend of choice, helpful for compressing and aggregating multiple proofs into one.
- Traits

  An abstract interface for types to implement.
- Sandbox Noir
  Public/private smart contract syntaxes MVP.
- References

  Borrow data rather than move or copy it.
- Conditional Compilation

  Adaptive compilation based on field of choice, e.g. `#[cfg(backend = "bn254")]`.

- Renewed NPM Packages

  For compiling & proving Noir programs written with the latest Noir syntaxes in JavaScript / web browser environments.
- Linguist Support

  Language recognition and syntax highlighting on GitHub.
- VS Code Sidebar

  Carrying out Noir actions such as proving and verifying within the Noir VS Code extension.
- In-browser VS Code

  Writing, proving and verifying Noir programs with VS Code in web browsers.
- Language Server Protocol Support

  Enhanced Noir DevEx across IDEs, adding features such as auto complete, go to definition, documentation on hover, etc.
- Formatter

  Automatic formatting of Noir source code.
- Package Registry

A site for discovering and downloading Noir packages written by different developers.

- CLI Package Manager

  Installing Noir packages in CLI.

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

(this uses an older version of Noir, 'constrain' has been replaced with assert)

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