Noir Programming Language

Private Value Transfer in 10 Lines

Maxim Vezenov
Software Engineer, Aztec Network

Section 1

Introduction to Noir

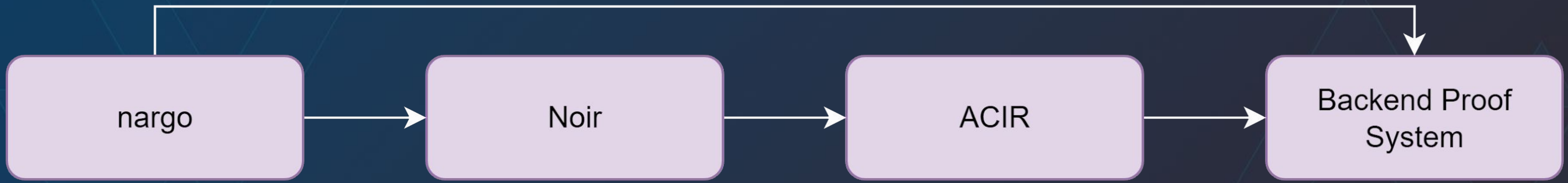
What is Noir and what is new?

Noir is more flexible in its design than other domain specific languages

- Compiles down to an intermediate representation
  - Abstract Circuit Intermediate Representation (ACIR)
  - The IR can to be compiled down to any NP complete language
- Enables the decoupling of the backend proof system and the language
  - Currently has one fully integrated backend that utilizes Aztec's barretenberg library
  - Plans for future integrations include arkworks proof systems such as Marlin and Groth16
- Only DSL that currently has fully integrated proving system optimizations
  - Custom gates

What is the benefit?

- A universal ZK DSL based on open source technology
  - Noir is Rust-based and draws on arkworks for its Field types
- Further collaboration in the ZK space that enables an open standard for ZK circuit construction
  - The EVM has created value that has extended past Ethereum itself
- Proof systems can supply a fixed list of optimized black box functions
  - These functions act as a standard library that the frontend can access
  - pedersen, merkle_membership, sha256, schnorr_verify
- Lower barriers to circuit development
  - Incorporate cryptographic safety into the language itself



compile, build, prove, verify

Text representing the circuit and program inputs

IR that can compile down to arithmetic circuit or R1CS constraints

Solve witness and create proof (PLONK, Groth16)

Section 2

Private Transfer Circuit

```
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] {
  ...
}
```

Private Transfer Circuit

- Rust-like syntax
- All inputs to main are private by default
  - The `pub` keyword makes them public, meaning they must also be supplied to the verifier
- One native Field type
  - Smaller data types such as u32 ultimately translate into a Field

Standard Library Functions

```
// 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]);
```

- Use the scalar_mul module to find the public key from the private key
- Hash the public key and random secret to hide the note commitments origin
- The standard library has multiple hash functions
  - Pedersen
  - Blake2s
  - Sha256
  - MiMC

```
    // 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]
}
```

Check Merkle Membership

- First, generate the nullifier to prevent double spends
  - This is public and returned from the circuit
- Standard library function for merkle membership
  - Currently very Aztec specific and limited to Pedersen for node compression
- Generics have recently been added with first-class functions on the timeline
  - Users will be able to specify which hasher they would like for their merkle membership proof

```
const N: Field = 5;

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; N]) {
    let res = x * N;
    constrain res == y[0];

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

    let bar1: Bar<Field> = Bar { one: res, two: y[0], other: mysubmodule::my_helper() };

    if bar1.other == 10 {
        foo(bar1);
    };
}

mod mysubmodule {
    const N: Field = 10;

    fn my_helper() -> const Field {
        N
    }
}
```

Additional Features

- Arrays, Tuples, Structs
- Submodules
- Global consts
- For loops
- If Statements
- Logical and Bitwise operators
- Generics

```
const N: Field = 5;

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; N]) {
    let res = x * N;
    constrain res == y[0];

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

    let bar1: Bar<Field> = Bar { one: res, two: y[0], other: mysubmodule::my_helper() };

    if bar1.other == 10 {
        foo(bar1);
    };
}

mod mysubmodule {
    const N: Field = 10;

    fn my_helper() -> const Field {
        N
    }
}
```

Simple Circuit Syntax

- Noir aims to be Rust-like in its syntax while abstracting away low-level concepts
- Complex cryptographic functionality can be supplied by the proving system through the stdlib rather than through new Noir libraries
- All smaller data types translate to a Field type
    - Can constrain on any of the data types Noir supports

Proving and Verifying in Typescript

- NoirJS
  - Enables compilation of a Noir program
  - Can read an ACIR from file generated by nargo
- Specify the program's ABI directly in Typescript
  - ABI parameters can be a NodeJS number type or hex string

```
let compiled_program = compile(
  path.resolve(__dirname, '../circuits/src/main.nr')
);
let acir = compiled_program.circuit;

let merkleProof = tree.proof(0);
let note_hash_path = merkleProof.pathElements

let abi = {
  recipient: recipient,
  priv_key: `0x` + sender_priv_key.toString('hex'),
  note_root: `0x` + note_root,
  index: 0,
  note_hash_path: [
    `0x` + note_hash_path[0],
    `0x` + note_hash_path[1],
    `0x` + note_hash_path[2],
  ],
  secret: `0x` + secret.toString('hex'),
  return: `0x` + nullifier.toString('hex'),
};
```

Proving and Verifying in Typescript

- We set up the prover and verifier using a Typescript wrapper around the proving system
    - @noir-lang/barretenberg
- As the proving system is compatible with the ACIR it just needs this as a parameter to set up a prover and verifier
- The ABI is used to solve the circuit's witness and ultimately generate the proof
- The public inputs are prepended to the proof
    - Formatted as 32 byte hex values
    - The inputs remain in order of how they are specified in the ABI

```
let [prover, verifier] = await setup_generic_prover_and_verifier(acir);

const proof: Buffer = await create_proof(prover, acir, abi);

const verified = await verify_proof(verifier, proof);
```

Verification with Solidity

- Aztec's barretenberg allows to compile from a Noir program to an Ethereum contract
  - Other proving systems must supply their own implementation
  - Same goes for verification with a different smart contract platform

```
async function generate_sol_verifier() {
    let compiled_program = compile(
      resolve(__dirname, '../circuits/src/main.nr')
    );
    const acir = compiled_program.circuit;

    let [_, verifier] = await setup_generic_prover_and_verifier(acir);

    const sc = verifier.SmartContract();
    syncWriteFile("../contracts/plonk_vk.sol", sc);
}

function syncWriteFile(filename: string, data: any) {
    writeFileSync(join(__dirname, filename), data, {
      flag: 'w',
    });
}

generate_sol_verifier().then(() => process.exit(0)).catch(console.log);
```

Verification with Solidity

- The proof can be passed to the Solidity verifier exactly as generated by the backend
  - No serialization or re-formatting is necessary
- This flow may differ with different proving systems and depends on the backend implementation being used with Noir
- Full example can be seen at https://github.com/vezenovm/simple_shield

```
let Verifier: ContractFactory =
    await ethers.getContractFactory("TurboVerifier");

let verifierContract: Contract = await Verifier.deploy();

const sc_verified = await verifierContract.verify(proof);

expect(sc_verified).eq(true)
```

Future Work

Here’s the timeline

Verify Proof

Recursive proofs inside of Noir

Effective Tooling

Improve the development experience through REPLs, IDE integrations, debugging tools

Noir Contracts

Noir-specific user-defined data type to enable public/private smart contracts in Noir

Thank you!

Maxim Vezenov
Software Engineer, Aztec Protocol
maxim@aztecprotocol.com
@maximvezenov

Noir offers simple syntax with optimized functionality

Simple Circuit Syntax

- Noir aims to be Rust-like in its syntax while abstracting away low-level concepts
- Complex cryptographic functionality can be supplied by the proving system through the stdlib rather than through new Noir libraries
- All smaller data types translate to a Field type
    - Can constrain on any of the data types Noir supports