Lesson 14 - Plonk, optimisations and Noir

Planned Updates to Cairo

Cairo v 1.0 - available in ~ 2 months
Focus - Developer happiness

Sierra

A new intermediate level representation
Transactions should always be provable
Asserts are converted to if statements, if it returns false we don't do any modifications to storage
Contracts will count gas
Still needs to be low level enough to be efficient
So the process would be
Cairo (new) => Sierra => Cairo bytecode
Sierra bytecode

- cannot fail
- counts gas
- compiles to Cairo with virtually no overhead

Plonkish protocols

(fflonk, turbo PLONK, ultra PLONK, plonkup, and recently plonky2.)

Before PLONK

Early SNARK implementations such as Groth16 depend on a common reference string, this is a large set of points on an elliptic curve.
Whilst these numbers are created out of randomness, internally the numbers in this list have strong algebraic relationships to one another. These relationships are used as short-cuts for the complex mathematics required to create proofs.
Knowledge of the randomness could give an attacker the ability to create false proofs.

A trusted-setup procedure generates a set of elliptic curve points $G, G \cdot s, G \cdot s^2 \ldots . G \cdot s^n$, as well as $G2 \cdot s$, where G and $G2$ are the generators of two elliptic curve groups and s is a secret that is forgotten once the procedure is finished (note that there is a multi-party version of this setup, which is secure as long as at least one of the participants forgets their share of the secret).
(The Aztec reference string goes up to the 10066396th power)

A problem remains that if you change your program and introduce a new circuit you require a fresh trusted setup.

In January 2019 Mary Maller, Sean Bowe et al released SONIC that has a universal setup, with just one setup, it could validate any conceivable circuit (up to a predefined level of complexity).
This was unfortunately not very efficient, PLONK managed to optimise the process to make the proof process feasible.

2^{17} Gates	PLONK		Marlin
Curve	BN254	BLS12-381 (est.)	BLS12-381
Prover Time	2.83s	4.25s	c. 30s
Verifier Time	1.4ms	2.8ms	8.5ms

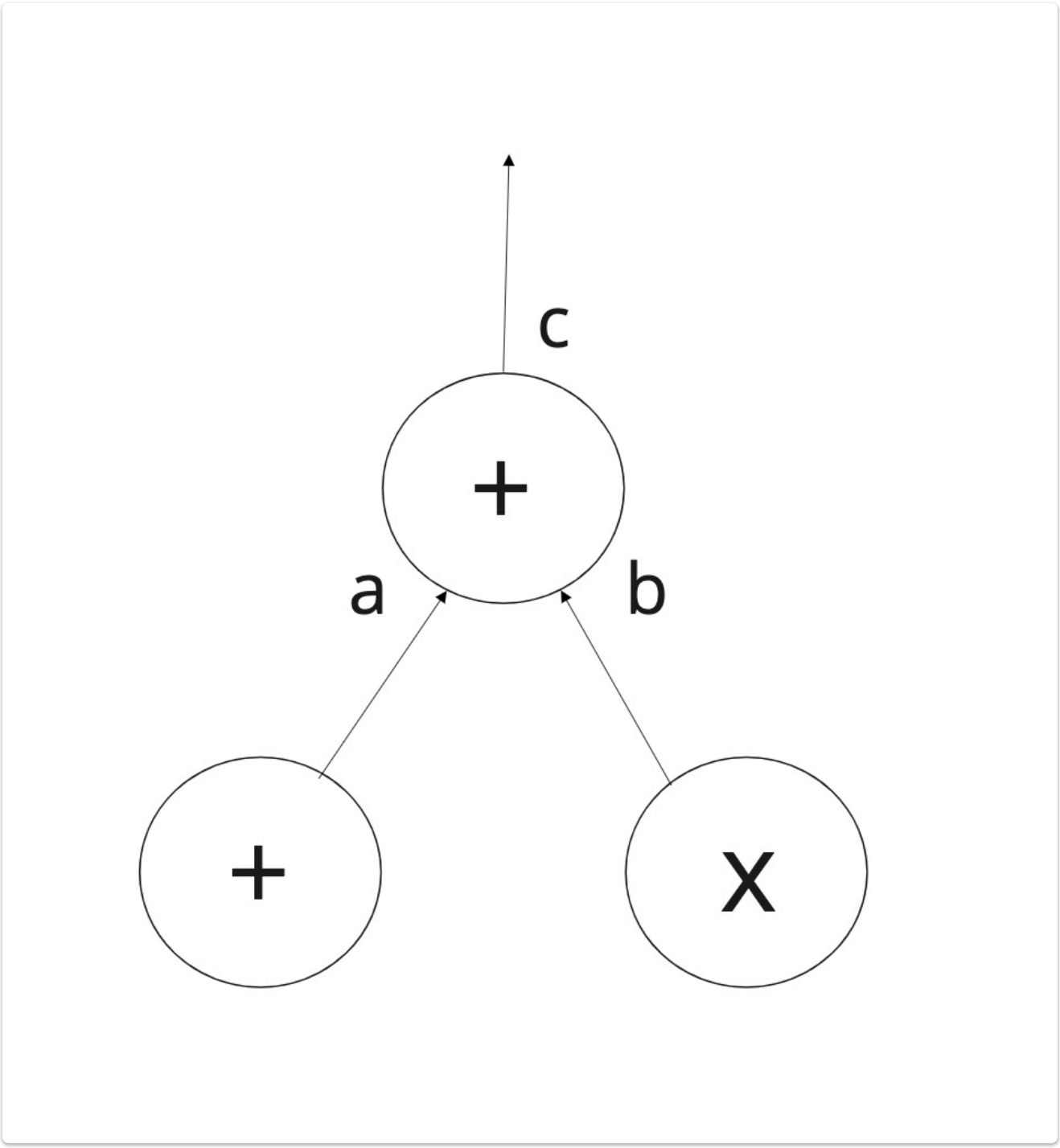
See PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge
Also see Understanding PLONK

Trusted Setup

This is still needed, but it is a "universal and updateable" trusted setup.

- There is one single trusted setup for the whole scheme after which you can use the scheme with any program (up to some maximum size chosen when making the setup).
- There is a way for multiple parties to participate in the trusted setup such that it is secure as long as any one of them is honest, and this multi-party procedure is fully sequential:

Arithmetic Circuits



Developing a circuit

Once we have a (potentially large)circuit, we want to get it into a more usable form, so we can put the values into a table detailing the inputs and outputs for a gate.

So for gates 1 to i we can represent the a, b and c values as
a_i, b_i, c_i

And if the circuit is correct then for an addition gate

a_i+ b_i = c_i

or
a_i+ b_i - c_i = 0

and for a multiplication gate

$a_i.b_i$ - c_i = 0

We would end up with a table like this

	a	b	c	S
1	a_1	a_1	a_1	1
2	a_2	b_2	c_2	1
3	a_3	b_3	c_3	0

But we would also want to know what type of gate it is, there is a useful trick where we introduce a selector S, which is 1 for an addition gate and 0 for a multiplication gate.

We can then generalise our equation as

$$S_i(a_i + b_i) + (1 - S_i)a_i.\, b_i - c_i = 0$$

These are called the *gate constraints* because they refer to the equalities for a particular gate.

We can also have *copy constraints* where we have a relationship between values that are not on the same gate, for example it may be the case that

a_7 = b_5 for a particular circuit, in fact this is how we link the gates together.

In PLONK we also have constant gates and more specialised gates.
For example representing a hash function as a series of generic gates (addition, multiplication and constant) would be inefficient.

From Zac Williamson

"PLONK’s strength is these things we call custom gates. It’s basically you can define your own custom bit arithmetic operations. I guess you can call them mini gadgets.
That are extremely efficient to evaluate inside a PLONK circuit.
So you can do things like do elliptical curve point addition in a gate.
You can do things like efficient Poseidon hashes, Pedersen hashes. You can do parts of a SHA-256 hash. You can do things like 8-bit logical XOR operations.
All these like explicit instructions which are needed for real world circuits, but they’re all quite custom."

Plookup

Some operations such as boolean operations do not need to be computed, but can be put into a lookup table. Similarly public data can be put in a lookup table.
For example we could use a lookup table for the XOR operation, or include common values as we saw in the range proofs in Aztec.

From circuit to R1CS to QAP

Once we have our circuit we can transform it into a set of polynomials which allows us to represent the large amount of data in a compact way.

Polynomial Commitments

A problem we face is that we can construct polynomials to represent our constraints, but these end up being being quite big.

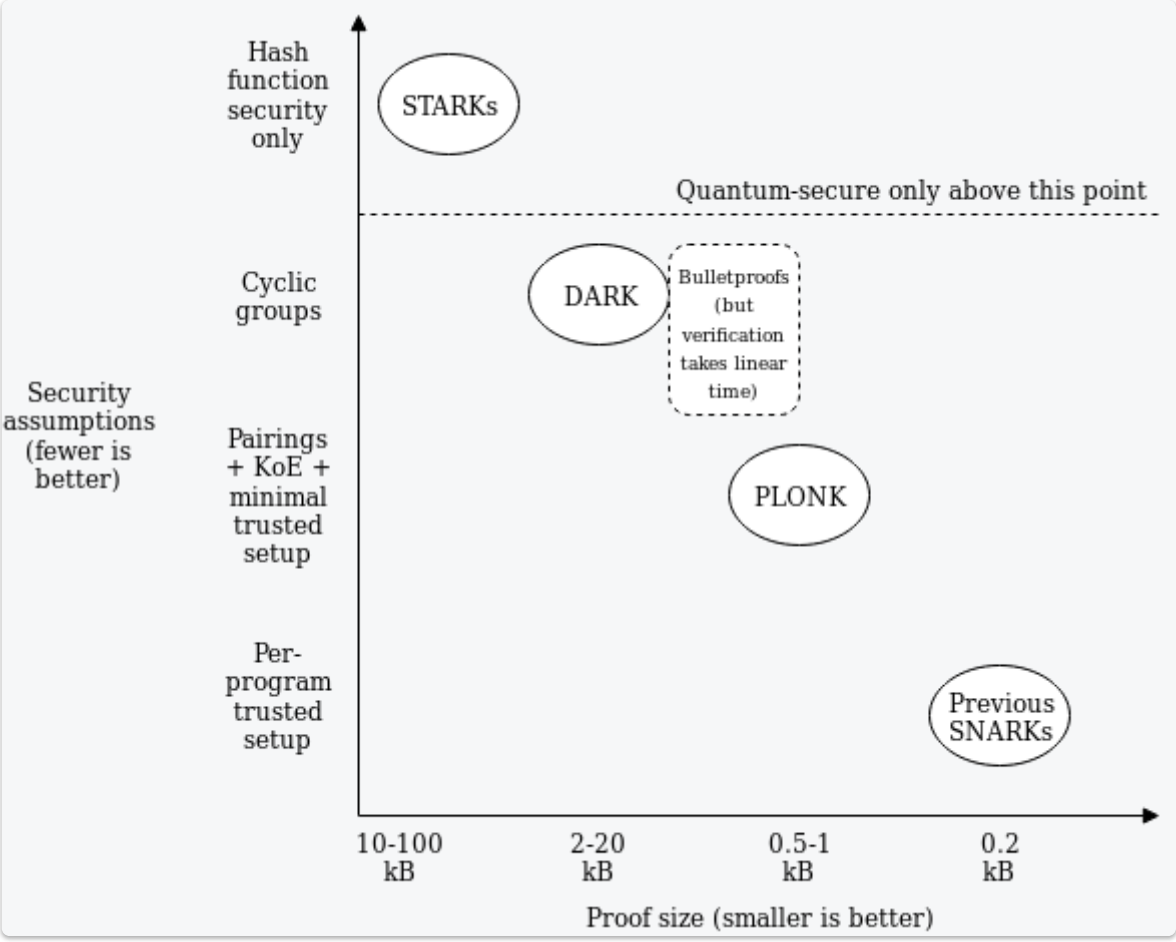
A polynomial commitment is a short object that "represents" a polynomial, and allows you to verify evaluations of that polynomial, without needing to actually contain all of the data in the polynomial.

That is, if someone gives you a commitment c representing $P(x)$, they can give you a proof that can convince you, for some specific z, what the value of $P(z)$ is.

There is a further mathematical result that says that, over a sufficiently big field, if certain kinds of equations (chosen before z is known) about polynomials evaluated at a random z are true, those same equations are true about the whole polynomial as well.

For example,
if $P(z) \cdot Q(z) + R(z) = S(z) + 5$, where z is a specific point
then we know that it's overwhelmingly likely that
$P(x) \cdot Q(x) + R(x) = S(x) + 5$ in general.

PLONK uses Kate commitments based on trusted setup and elliptic curve pairings, but these can be swapped out with other schemes, such as FRI (which would turn PLONK into a kind of STARK



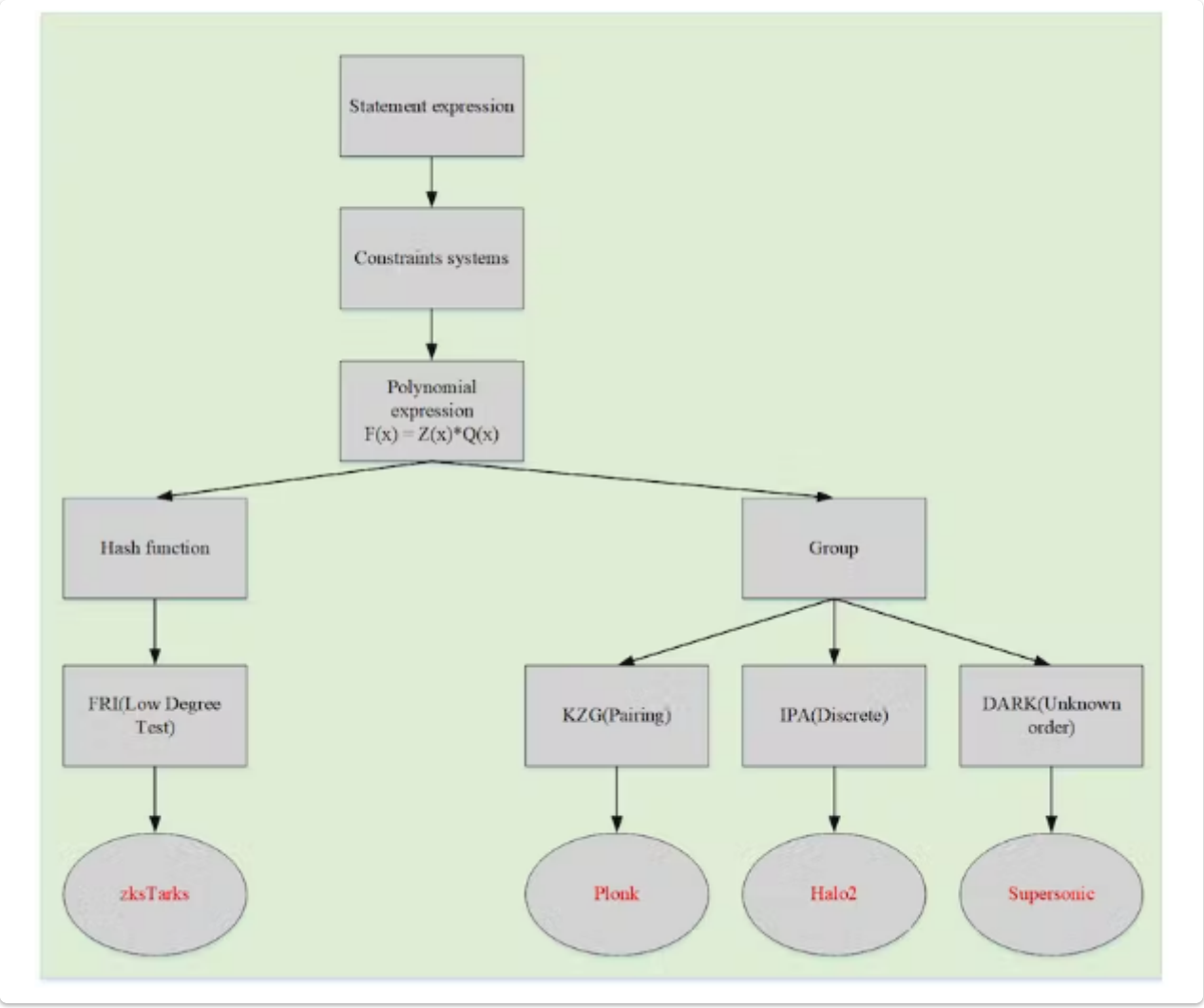
This means the arithmetisation - the process for converting a program into a set of polynomial equations can be the same in a number of schemes.
If this kind of scheme becomes widely adopted, we can thus expect rapid progress in improving shared arithmetisation techniques.

For a detailed talk about some of the custom gates used in Plonk see this video

For more information about the KZG commitment scheme see this introduction

Other commitment schemes are available :
Dory
Dark

For an analysis of polynomial commitment schemes, see this article



From the Scroll documentation

Recent advances to improve efficiency of SNARKS

- The usage of polynomial commitment.
  In the past few years, most succinct zero-knowledge proof protocols stick to R1CS with a query encoded in an application-specific trusted setup.
  The circuit size usually blows up and you can't do many customised optimisations since the degree of each constraint needs to be 2 (bilinear pairing only allows one multiplication in the exponent).
- With polynomial commitment schemes, you can lift your constraints to any degree with a universal setup or even transparent setup. This allows great flexibility in the choice of backend.
- The appearance of lookup table arguments and customized gadgets.
  Another strong optimisation comes from the usage of lookup tables. The optimisation is firstly proposed in Arya and then gets optimised in Plookup. This can save A LOT for zk-unfriendly primitives (i.e., bitwise operations like AND, XOR, etc.) . Customized gadgets allow you to do high degree constraints with
  efficiency. TurboPlonk and UltraPlonk defines elegant program syntax to make it easier to use lookup tables and define customised gadgets. This can be extremely helpful for reducing the overhead of EVM circuit.
- A Recursive proof is more and more feasible.
  Recursive proof has a huge overhead in the past since it relies on special pairing-friendly cyclic elliptic curves (See paper). This introduces a large computation overhead. However, more techniques are making this possible without sacrificing the efficiency.
- For example, Halo can avoid the need of pairing-friendly curve and amortize the cost of recursion using a special inner product argument. Aztec shows that you can do proof aggregation for existing protocols directly (lookup tables can reduce the overhead of non-native field operation thus can make the verification circuit smaller). It can highly improve the scalability of supported circuit size.
- Hardware acceleration is making proving more efficient.
  To the best of our knowledge, we have made the fastest GPU and ASIC/FPGA accelerator for the prover. Our paper describing ASIC prover has already been accepted by the largest computer conference (ISCA) this year. The GPU prover is around 5x-10x faster than Filecoin's implementation. This can greatly improve the prover's computation efficiency.

Lurk Language

See blog
Lurk is a Turing-complete language produced by Filecoin to allow creation
of recursive zk-SNARKs.
It is a statically scoped dialect of Lisp
Features

- Verifiable computation
- Succinct proofs
- Zero knowledge
- Turing-completeness
- Arbitrary traversal of content-addressable data
- Higher-order functions (e.g. functions as public inputs to computations, with proof)
- Content-addressable data for natural integration with IPFS and IPLD

Noir

Documentation
Installing Noir

Noir is similar to Rust (it uses `nargo` rather than `cargo`)

Process

Say we have a main method thus

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

The parameters x and y are supplied by the prover, neither are public.
The prover supplies the values for `x` and `y` in the `Prover.toml` file.

```
x = "5"
y = "10"
```

When the command `nargo prove my_proof` is executed, two processes happen:

- First, Noir creates a proof that `x` which holds the value of `5` and `y` which holds the value of `10` is not equal. This not equal constraint is due to the line `constrain x != y`.

- Second, Noir creates and stores the proof of this statement in the `proofs` directory and names the proof file `my_proof`.

When the command `nargo verify my_proof` is executed, two processes happen:

- Noir checks in the `proofs` directory for a file called `my_proof`
- If that file is found, the proof's validity is checked.

ACIR

This is an intermediate representation between the proof system and noir syntax, it allows the frontend to compile to any ACIR compatible proof system.
When we compile a proof
1. The Noir code is compiled to ACIR and it's witness is checked.

2. Details are stored in the build directory as my_proof.acir for the acir and my_proof.tr for the witness.

Noir Language

Mutability

We need to specify mutability if we wish to change variables

```
let x = 2;
x = 3; // error: x must be mutable to be assigned to

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

Datatypes

Noir has
`Field`,
integer types - u4, u24 etc.
`bool`,
arrays,
structs,
tuples

A field type corresponds to a native field type in the backend. Usually this is a (roughly) 256-bit integer. This should generally be the default type reached for to solve problems. Using a smaller integer type like `u64` incurs extra range constraints and so is less efficient rather than more.

Constants

A constant type is a value that does not change per circuit instance. This is different to a witness which changes per proof. If a constant type that is being used in your program is changed, then your circuit will also change.

Below we show how to declare a constant value:

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

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

Functions

Functions look similar to Rust

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

Conditionals

If statements are much simpler than in other DSLs

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

if a == 0 {
    x = 7;
    }
} else {
    x = 5;
    constrain x == 5;
}
constrain x == 2;
```

Loops

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

Constraints

Noir includes a special keyword `constrain` 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.

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

Mastermind Example

```
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]
}
```