Lesson 16 - Auditing / Verification / Course Review

Noir Example Projects

See this list of resources

- Anonymous proof of token ownership on Aztec (for token-gated access)
  - Sequi
  - Cyclone

Governance - MeloCafe
Circuits - Anonymous on-chain voting

Arkworks

`arkworks` is a Rust ecosystem for zkSNARK programming. Libraries in the `arkworks` ecosystem provide efficient implementations of all components required to implement zkSNARK applications, from generic finite fields to R1CS constraints for common functionalities.

Tutorial includes Exercises for

1. Merkle Tree
2. Validating a single transaction
3. Writing a rollup circuit

Halo2

See tutorial
See documentation
See Halo2 Book

Halo 2 is a proving system that combines the Halo recursion technique with an arithmetisation based on PLONK, and a polynomial commitment scheme based around the Inner Product Argument

CHIPS

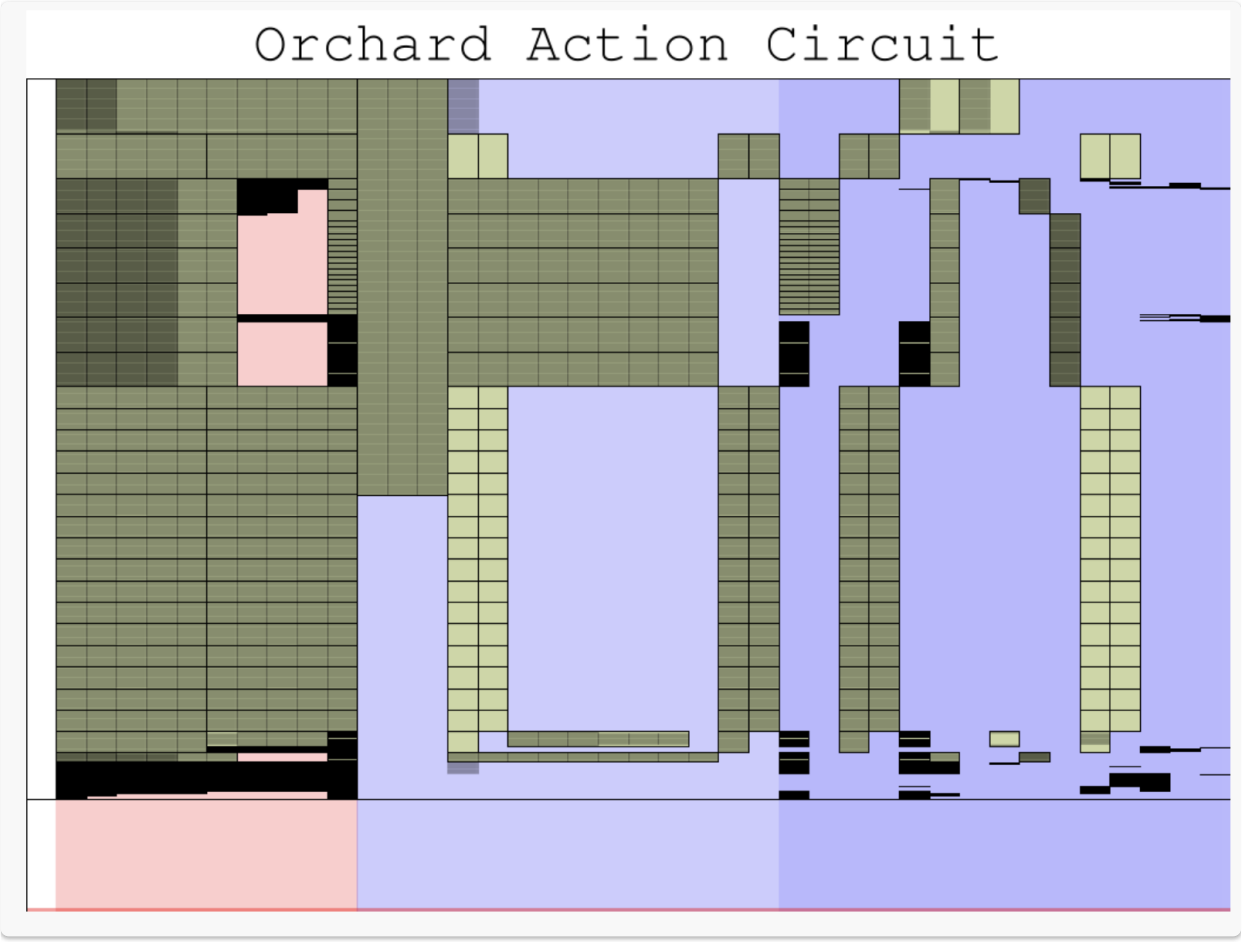
Using our API, we define chips that "know" how to use particular sets of custom gates. This creates an abstraction layer that isolates the implementation of a high-level circuit from the complexity of using custom gates directly.

Example Simple Circuit

```
trait NumericInstructions<F: Field>: Chip<F> {
    /// Variable representing a number.
    type Num;

    /// Loads a number into the circuit as a private input.
    fn load_private(&self, layouter: impl Layouter<F>, a: Value<F>) -
> Result<Self::Num, Error>;

    /// Loads a number into the circuit as a fixed constant.
    fn load_constant(&self, layouter: impl Layouter<F>, constant: F)
-> Result<Self::Num, Error>;

    /// Returns `c = a * b`.
    fn mul(
        &self,
        layouter: impl Layouter<F>,
        a: Self::Num,
        b: Self::Num,
    ) -> Result<Self::Num, Error>;

    /// Exposes a number as a public input to the circuit.
    fn expose_public(
        &self,
        layouter: impl Layouter<F>,
        num: Self::Num,
        row: usize,
    ) -> Result<(), Error>;
}
```

Halo 2 circuits are two-dimensional: they use a grid of "cells" identified by columns and rows, into which values are assigned.
Constraints on those cells are grouped into "gates", which apply to every row simultaneously, and can refer to cells at relative rows.
To enable both low-level relative cell references in gates, and high-level layout optimisations, circuit developers can define "regions" in which assigned cells will preserve their relative offsets.

Example from ZCash



In the example circuit layout pictured, the columns are indicated with different backgrounds.
The instance column in white;
advice columns in red;
fixed columns in light blue; and
selector columns in dark blue.
Regions are shown in light green, and assigned cells in dark green or black.

COLUMN TYPES

- Instance columns contain per-proof public values, that the prover gives to the verifier.
- Advice columns are where the prover assigns private (witness) values, that the verifier learns zero knowledge about.

> - Fixed columns contain constants used by every proof that are baked into the circuit.
> - Selector columns are special cases of fixed columns that can be used to selectively enable gates.

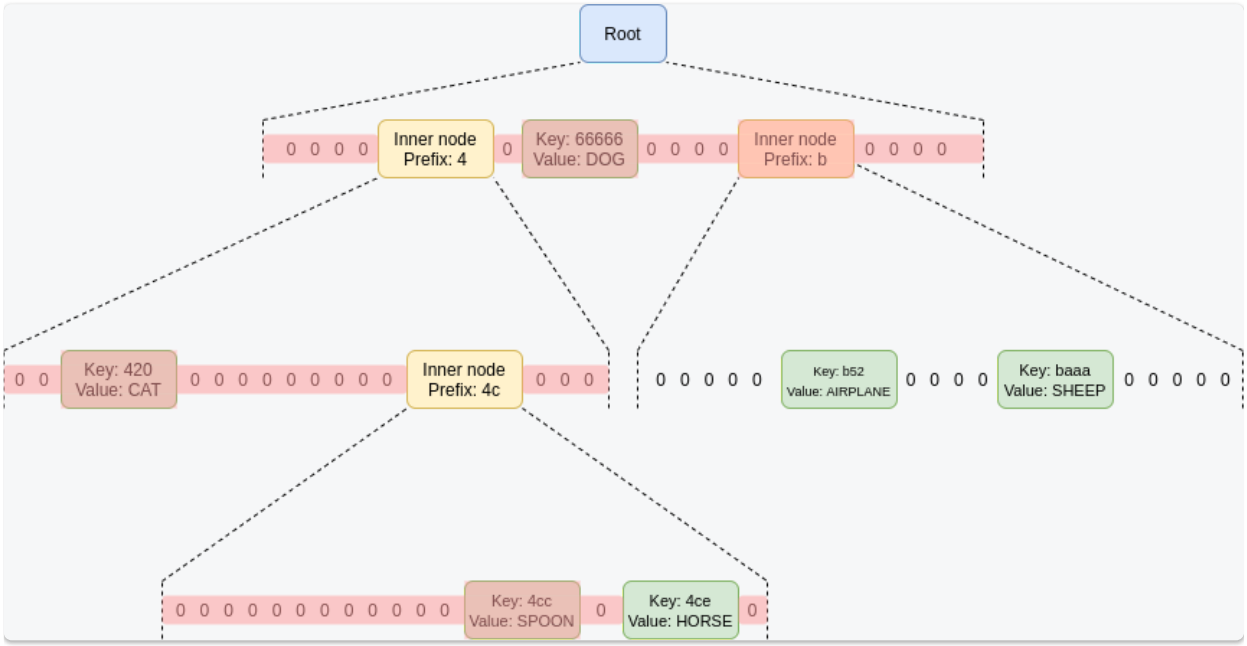
Verkle trees

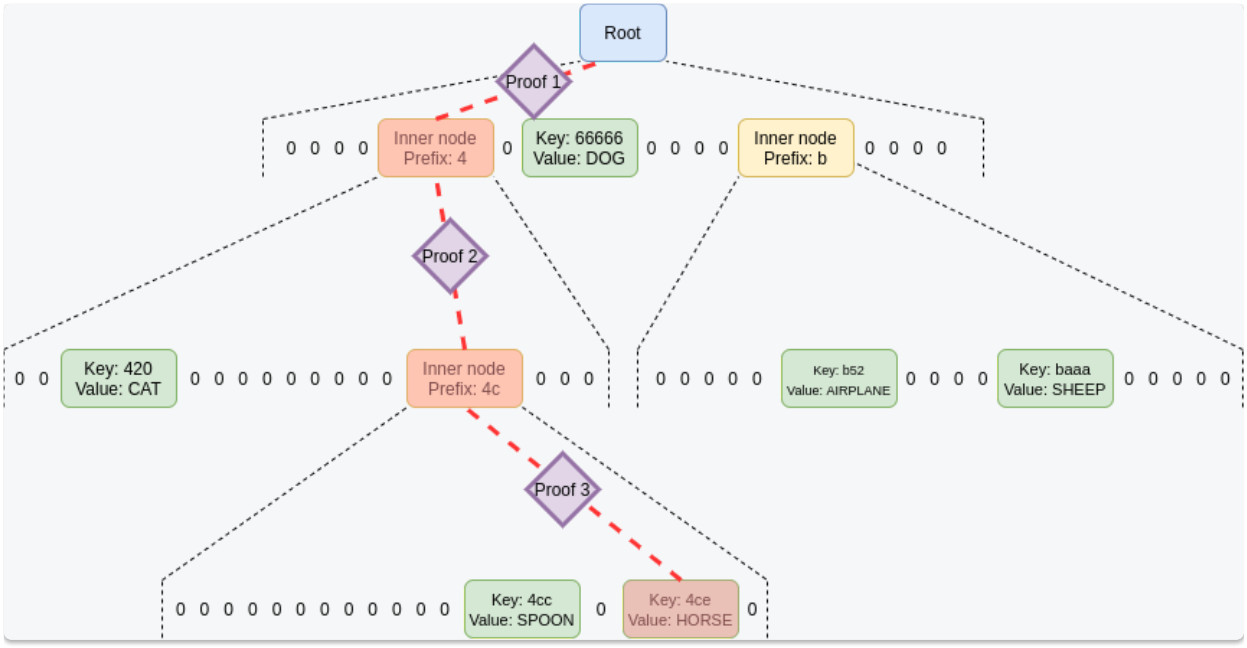
https://vitalik.ca/general/2021/06/18/verkle.html

See article
and Ethereum Cat Herders Videos

Like merkle trees, you can put a large amount of data into a Verkle tree, and make a short proof ("witness") of any single piece, or set of pieces, of that data that can be verified by someone who only has the root of the tree.
What Verkle trees provide, however, is that they are much more efficient in proof size. If a tree contains a billion pieces of data, making a proof in a traditional binary Merkle tree would require about 1 kilobyte, but in a Verkle tree the proof would be less than 150 bytes.
Verkle trees replace hash commitments with vector commitments or better still a polynomial commitment.
Polynomial commitments give us more flexibility that lets us improve efficiency, and the simplest and most efficient vector commitments available are polynomial commitments.

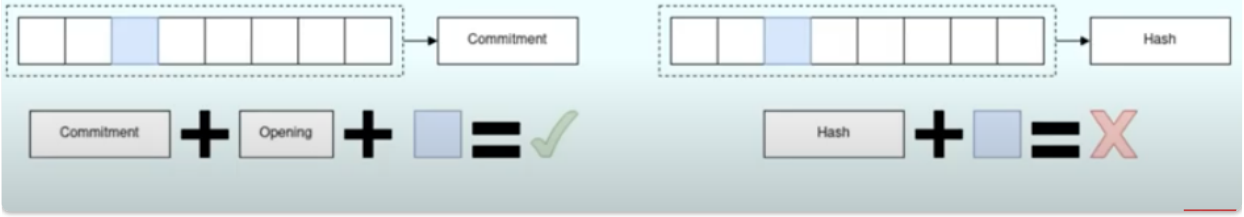
The number of nodes needed in a merkle proof is much greater than in a verkle proof





Vector commitments vs. Hash

- Vector commitments: existence of an “opening”, a small payload that allow for the verification of a portion of the source data without revealing it all.
- Hash : verifying a portion of the data = revealing the whole data.



Proof sizes

Merkle

Leaf data +
15 sibling
32 bytes each
for each level (~7)

= ~3.5MB for 1K leaves

Verkle

Leaf data +
commitment + value + index
32 + 32 + 1 bytes
for ~4 levels
+ small constant-size data

= ~ 150K for 1K leaves

ZKP Bridges

With succinct proofs, zkBridge not only guarantees strong security without external assumptions, but also significantly reduces on-chain verification cost. zkBridge provides a modular design supporting a base layer with a standard API for smart contracts on one chain to obtain verified block headers from another chain using snarks. By separating the bridge base layer from application-specific logic, zkBridge makes it easy to enable additional applications on top of the bridge, including message passing, token transfer, etc..

There is a ongoing hackathon including a track for ZKP bridges
ZK Hacking

Audit

A number of auditing companies are active in this space
Peckshield, Chainsecurity, Open Zeppelin
Nethermind are building their audit team, and have created tools and guidelines.
Consensys Dilligence have announced a partnership with Starkware for auditing Cairo contracts.

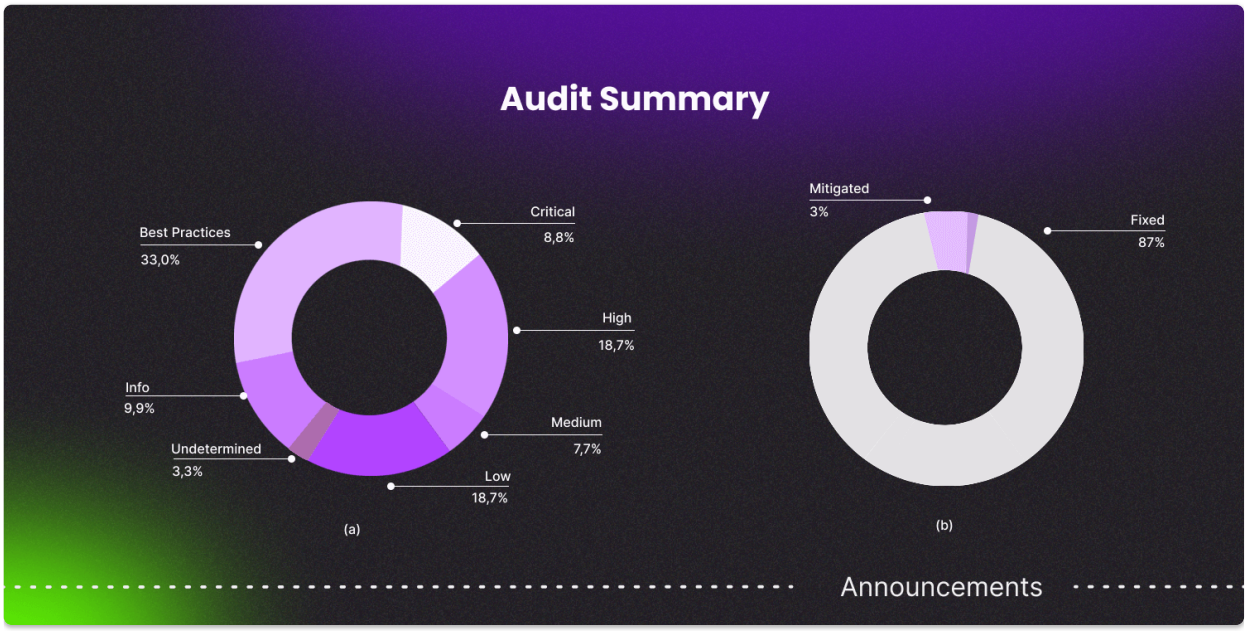
Security Guidelines

See this article from ctrlc03 an internal auditor and developer within the EF's PSE team.

This article also gives some vulnerabilities in cairo.

Example audits of cairo contracts

ZKX - overview of audit



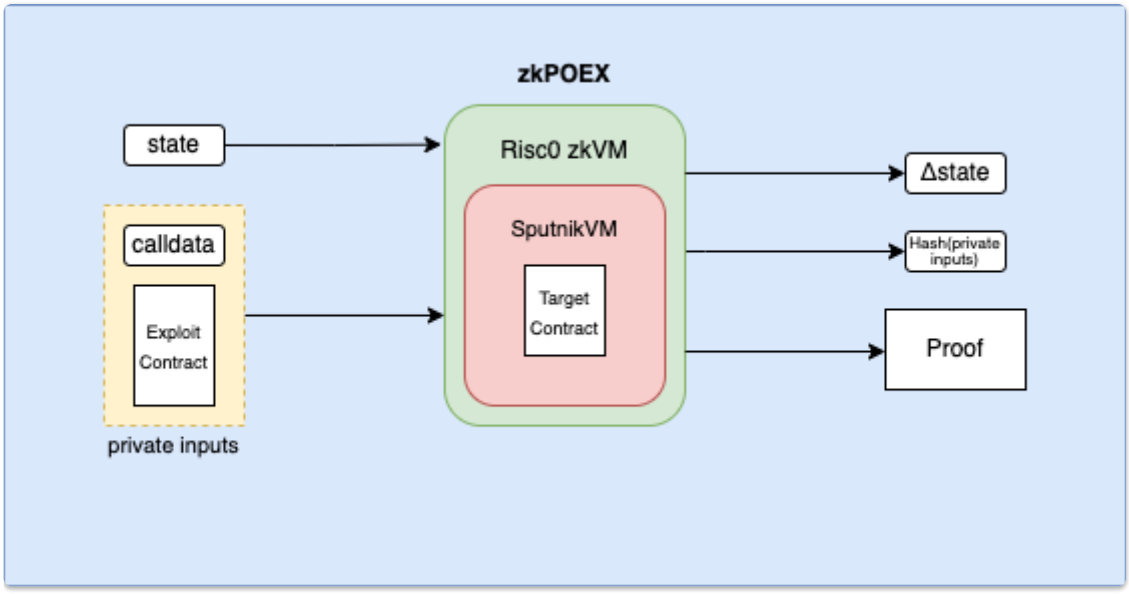
Maker DAO - Chain Security Report

Static Analysis

Armana Repo

Using ZKPs to verify exploits

See the recent project from ETHDenver

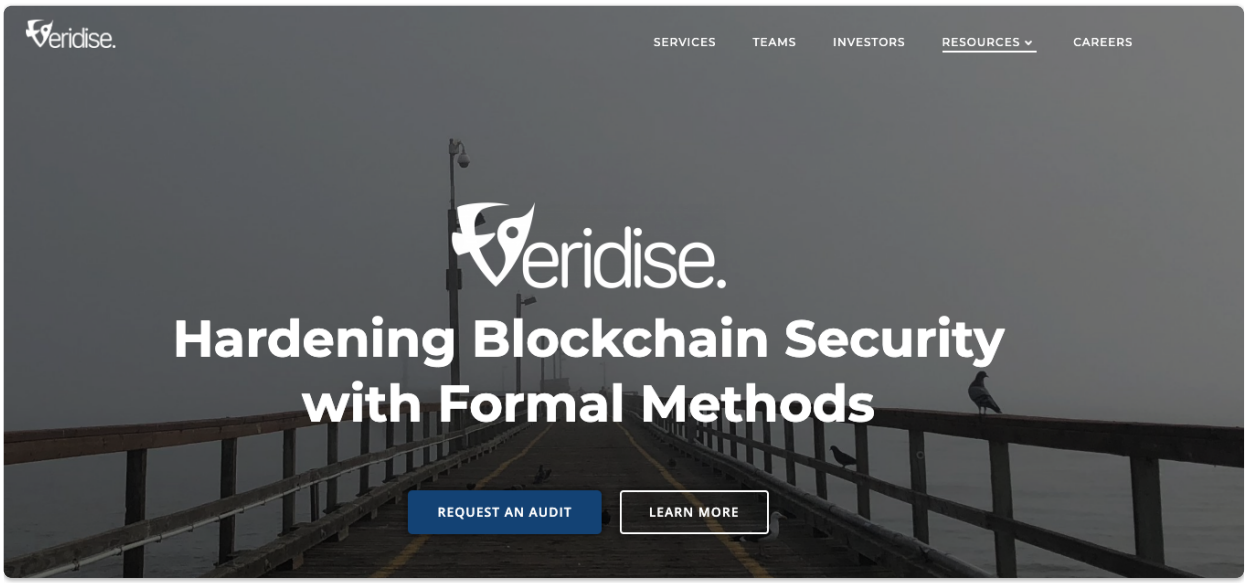


Technologies Used

The project utilises the following technologies:

- Risc0: A General Purpose Zero-Knowledge VM that allows to prove and verify any computation. The RISC Zero ZKVM is a verifiable computer that works like a real embedded RISC-V microprocessor, enabling programmers to write ZK proofs like they write any other code.
- SputnikVM: A high-performance, modular virtual machine for executing Ethereum smart contracts.

Formal verification of ZKPs



Picus - Formal Verification of R1CS

See repo

Medjia - Symbolic execution

See Repo

FEATURES

- Program Exploration: Medjai can execute Cairo program with symbolic inputs and explore all its possible program states.
- Property Verification: Medjai can check whether certain properties hold on Cairo program.
- Attack Synthesis: Medjai can automatically solve for concrete inputs that crash given Cairo program.
- Integrations with Veridise Product Lines: Medjai integrates with V specification language](https://github.com/Veridise/V) that allows developers to express correctness properties.

```
@external
func move_demo_spec{
    syscall_ptr : felt*, pedersen_ptr : HashBuiltin*,
range_check_ptr, bitwise_ptr : BitwiseBuiltin*
}():
    alloc_locals
    local otherUser
    local src
    local dst
```

```
    let (local src) = medjai_make_symbolic_felt()
    let (local dst) = medjai_make_symbolic_felt()

    # _dai.write(src, Uint256(low=0,
high=340282366920938463463374607431768211455))
    let (local daiSrcBefore : Uint256) = _dai.read(src)
    medjai_assume_valid_uint256(daiSrcBefore)  # Assume the value is
a valid uint256
    let (local daiDstBefore : Uint256) = _dai.read(dst)
    medjai_assume_valid_uint256(daiDstBefore)  # Assume the value is
a valid uint256

    let (local rad : Uint256) = medjai_make_symbolic_uint256()  #
Checked in move
    move(src, dst, rad)

    let (local daiSrcAfter : Uint256) = _dai.read(src)
    medjai_assert_le_uint256(daiSrcAfter, daiSrcBefore)
    return ()
end
```

Giza

See Repo

Giza leverages the Winterfell library to prove and verify the execution of programs running on the Cairo VM.

Horus

See article
See repo

Horus is a command-line formal verification tool for Starknet contracts.

The installation is 'convoluted' , the docker container might be the best option

Annotations are used to specify invariants

For example the
@post annotation specifies what should happen when a function returns
In this case, that `res` equals 3

```
// @post $Return.res == 3
func example() -> (res: felt) {
        return (3,);
}
```

Similarly `@pre` defines pre function conditions

`@storage_update` specifies values for storage variables
`// @storage_update x() := x() + 1`

`@assert` can be used as a check on a variable
For example
`// @assert j >= 10`

See the documentation for the other annotations

Once you have annotated your contract, you can compile it

`horus-compile a.cairo --output b.json --spec_output spec.json`

then you can verify the contract with

`horus-check b.json spec.json`

This verifies the compiled StarkNet contract `b.json` and its
specification `spec.json` with the default SMT solver `cvc5`, and prints the output
to `stdout`.

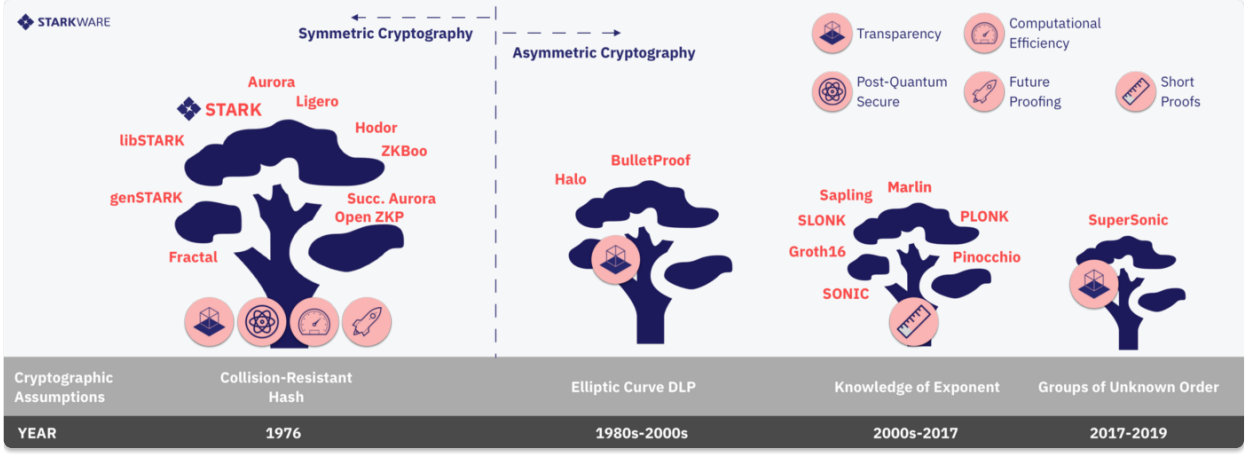
Useful talks from Devcon

ZKP Workshop video
Zk Application showcase video
The KZG Ceremony video
zkEVM Technical details video
Vampire SNARK from Nethermind video
Shielded Voting and Threshold encryption video
Self Sovereign Identity video
Are your ZKPs correct ? video
ZKP Performance video
Proving execution in zkEVM video
ZKP Introduction video
Autonomous Worlds workshop video

Course Review

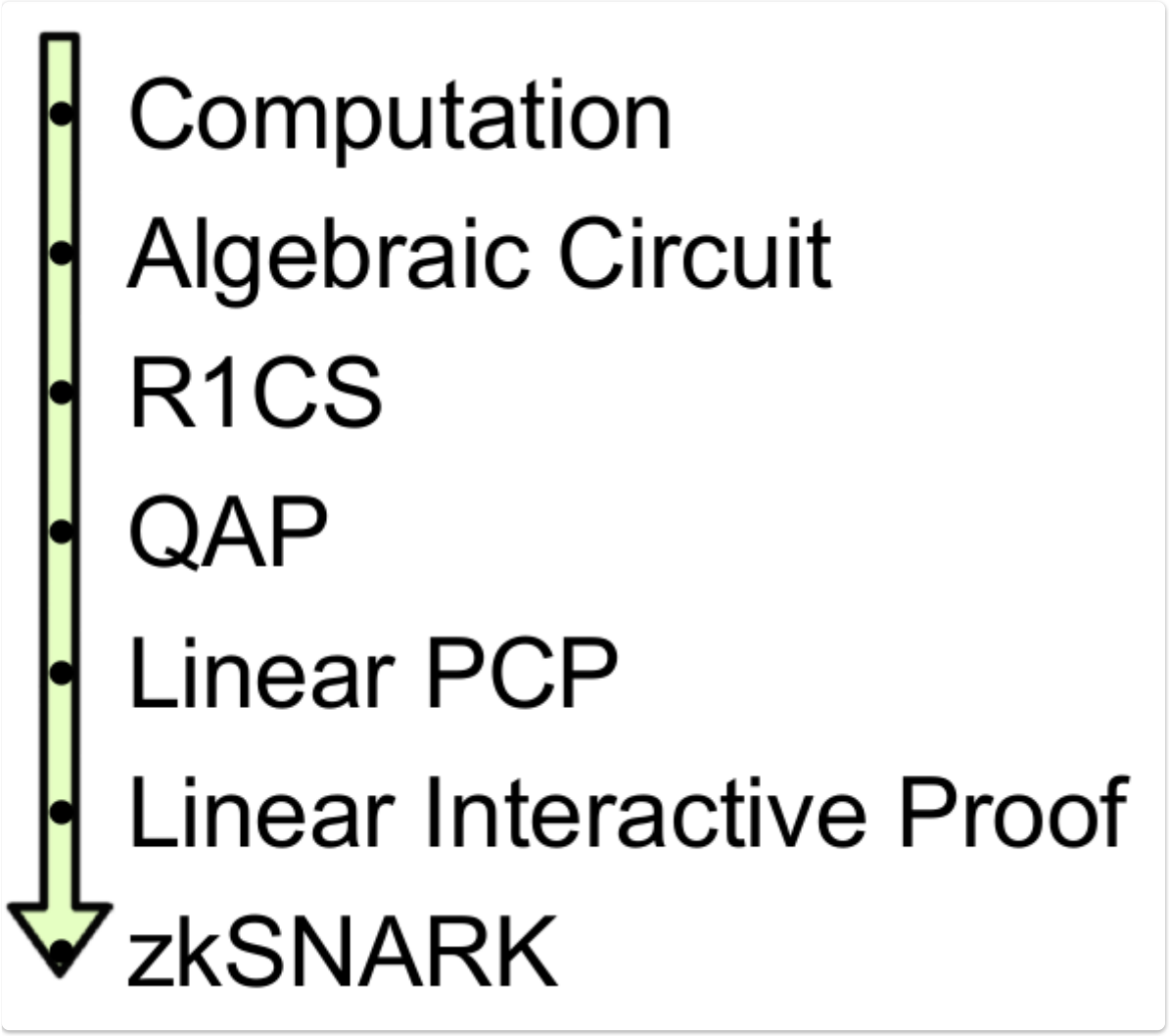
Lesson 1 & 2

- Introductory maths & cryptography
- General ZKP theory



	SNARKs	STARKs	Bulletproofs
Algorithmic complexity: prover	O(N * log(N))	O(N * poly-log(N))	O(N * log(N))
Algorithmic complexity: verifier	~O(1)	O(poly-log(N))	O(N)
Communication complexity (proof size)	~O(1)	O(poly-log(N))	O(log(N))
- size estimate for 1 TX	Tx: 200 bytes, Key: 50 MB	45 kB	1.5 kb
- size estimate for 10.000 TX	Tx: 200 bytes, Key: 500 GB	135 kb	2.5 kb
Ethereum/EVM verification gas cost	~600k (Groth16)	~2.5M (estimate, no impl.)	N/A
Trusted setup required?	YES 😒	NO 😄	NO 😄
Post-quantum secure	NO 😒	YES 😄	NO 😒
Crypto assumptions	Strong 😒	Collision resistant hashes 😄	Discrete log 😒

zkSNARKS



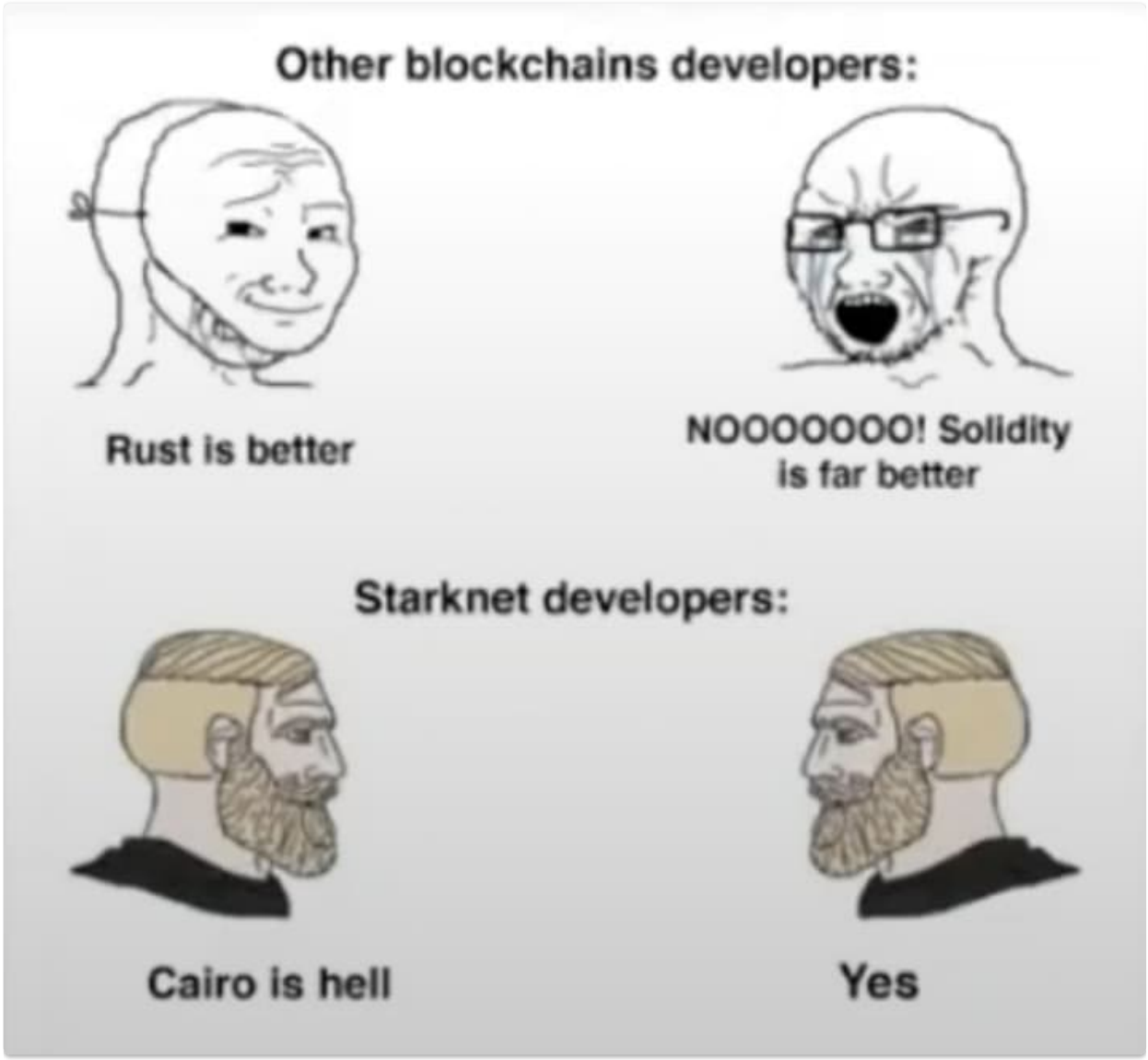
Lesson 3

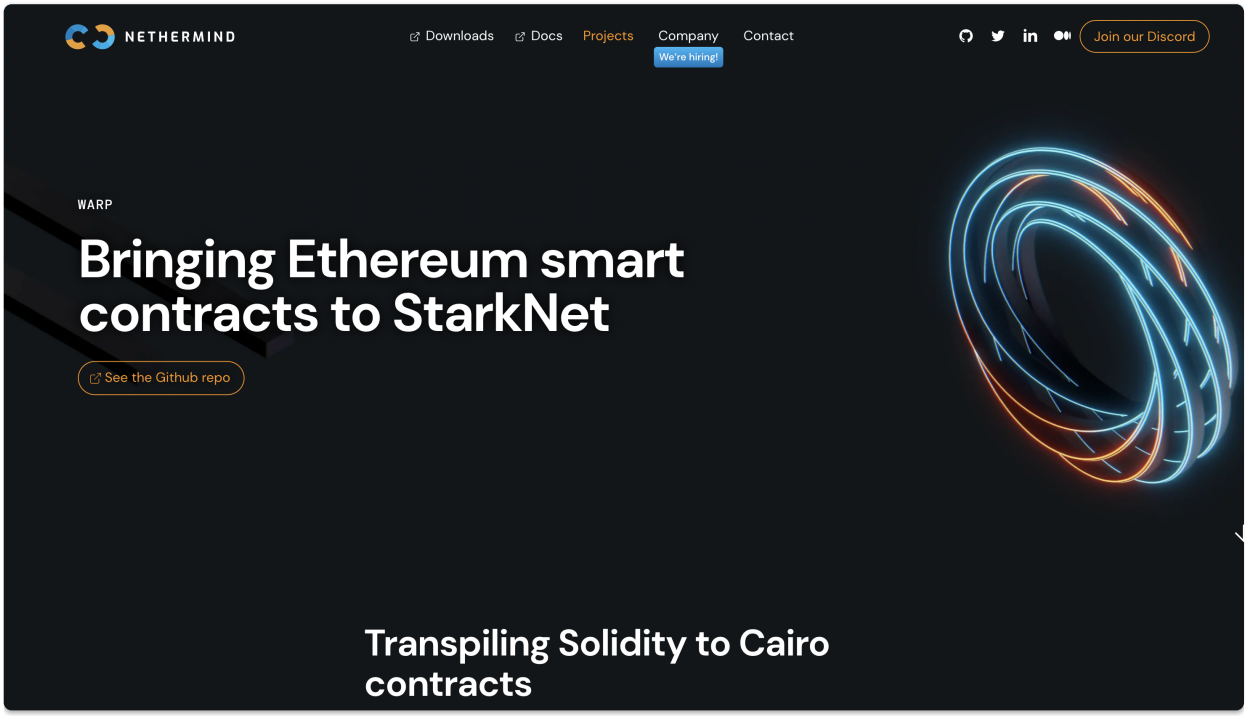
- ZKP use cases / rollups

Lessons 4 - 7

- Starknet / Cairo / Warp

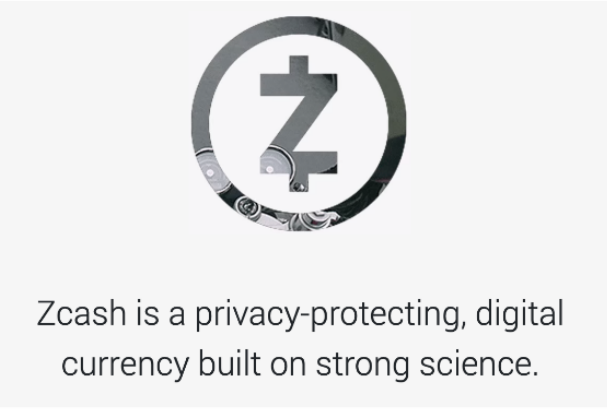
Cairo


Other blockchains developers:
Rust is better
NOOOOOOO! Solidity is far better
Starknet developers:
Cairo is hell
Yes


NETHERMIND
Downloads
Docs
Projects
Company
We're hiring!
Contact
Join our Discord
WARP
Bringing Ethereum smart contracts to StarkNet
See the Github repo
Transpiling Solidity to Cairo contracts

Lesson 8

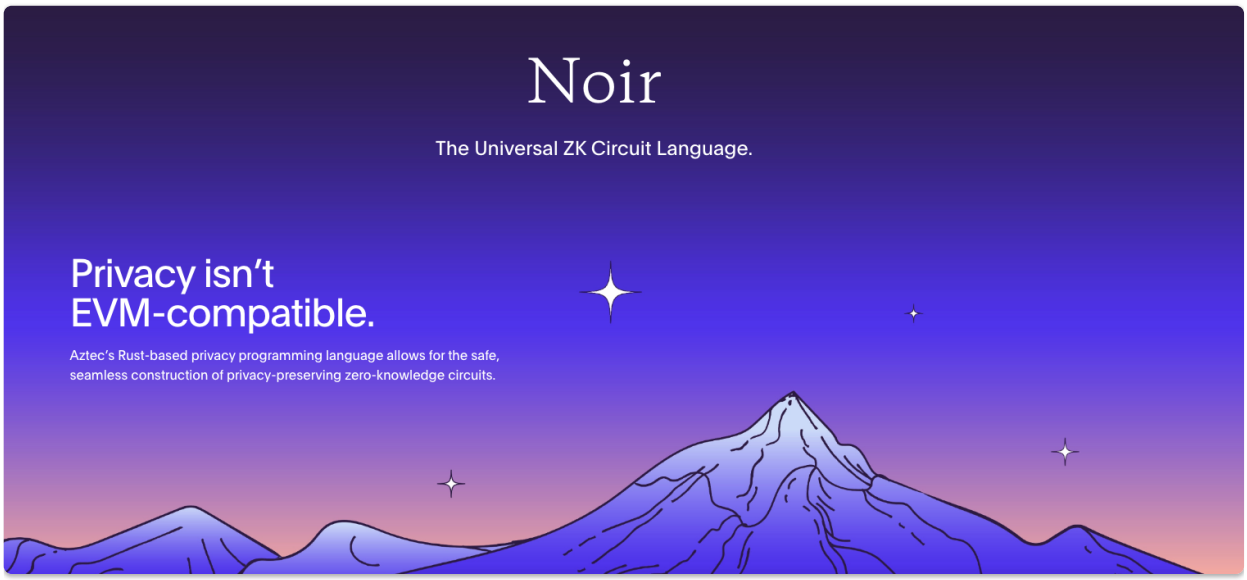
- Confidential tokens



Also Nightfall , ZKDai

Lesson 9

- Noir

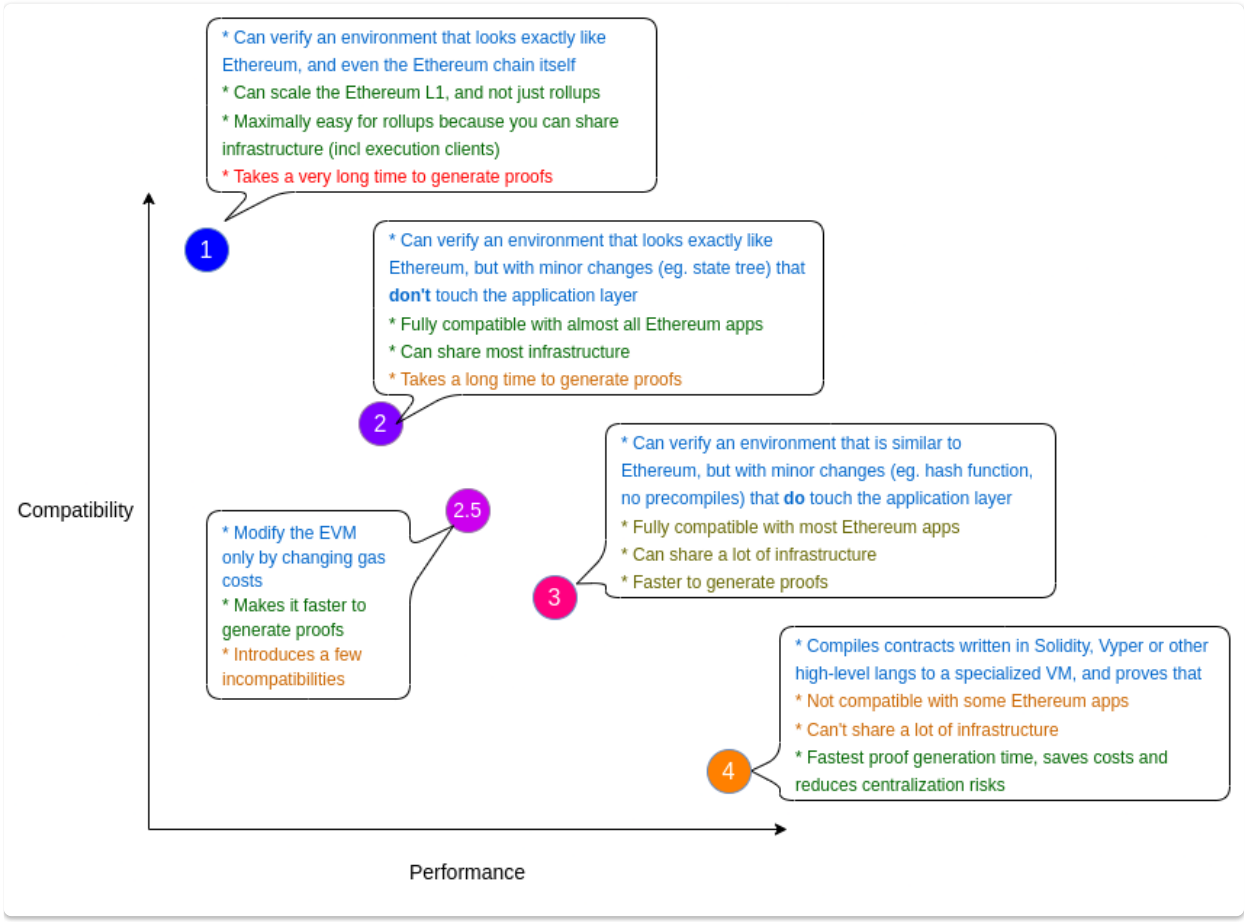


Lesson 10

- Mina
- zkApps

Lesson 11

zkEVM solutions



Lesson 12 & 13

- SNARK and STARK theory

Lesson 14

- Circom
- Alternative technologies
- Voting systems

Lesson 15

- Identity Solutions
- Oracles

Lesson 16

- Auditing / verification

Resources for further learning

Groups

Mina UK Meetup group
Starknet Meetup groups

Podcasts

Zero knowledge podcast
Starknet podcast

Online courses / sessions

ZK Whiteboard sessions
Cairo mummies course

Extropy Foundation resources