Table of Contents

About?

Cryptography is the scientific art of concealing data in a manner that makes it impractical or highly time-consuming to reverse engineer the encoded message. In the context of our scenario, where we aim to transmit a confidential message from one party to another, we'll adopt the commonly used actor names Alice and Bob.

Good old Alice and Bob! We'll get used to hearing about their adventures through out the book.

Let's say Alice and Bob are two parties who have never met, but want to communicate securely over the internet. How could they do this? In this thought experiment, we'll assume that any communication sent by Alice will be intercepted and read by unauthorized parties. Perhaps the first idea we might come with is to share some password to encrypt a message, but if the communications are always read in transit, how do we send the password? Perhaps the parties could arrange to meet physically and share some code or password and assumed that they're not being watched, but this of course is pretty impractical.

In 1977 Ron Rivest, Adi Shamir, and Leonard Adleman from MIT invented the RSA algorithm that is widely used for secure data transmission.

In this chapter, we will developer a deeper understanding of the RSA algorithm, and other fundamental cryptographic algorithms that are essential to our application.

Randomness

As your probably aware, computers are determistc, given a set of inputs they should always return the same outputs. If they didnt, it would be pretty chaotic using a computer! Random Number Generators or RNGs are .... The are USB hardware devices like the TrueRNG by ubld.it that create randomness for your device or server for around $100 USD.

Computers and software however, use Psudeo Random Number Generators to create a random output. However, they rely on a "seed" input, often the computers time stamp. When creating passwords or perhaps more importantly crypto private keys, using a random number is critcial. There is a good YouTube video by the former hacker "King Pin" where he cracks a password on a bitcoin wallet to recover a few million dollars (at time of writing) with of bitcoin. The software used to generate the password took the computers time as the seed to generate the password. The attackers took a range of dates where they thought the orignal password was genrated and replayed all the milliseconds in that range. The attack was sucessful and the coins where recovered!

RC4

```
i = (i + 1) mod 256
j = (j + Si) mod 256
swap Si and Sj
t = (Si + Sj) mod 256
K = St
```

[RC4 code todo]

Ciphertext (in base64): wqLCthTDlMKvZR7Dh8Kvf1JZw5s=
Decrypted text: Hello, World!

What is entropy?

Lets explore some Go code to explain the topic further, and how our validators will use a random seed to create a random number.

[TODO GO CODE]

Online randomness as a service...

The Avalanche Effect

The avalanche effect in cryptography refers to a desirable property of cryptographic algorithms, particularly block ciphers and hash functions. This property ensures that a small change in the input (such as flipping a single bit) results in a significant and unpredictable change in the output.

In functions like the hash function, a small change in the input message, even one bit, should result in a hashed value that looks completely different from the original hash. This proprty is critcal for ensuring data integreity and security.

Hash functions

Hash functions are "one way" algorithms that take an input (or 'message') and return a fixed-size string of bytes. The output, typically called the hash value or digest, appears random and unique for each unique input. Reversing the data is not possible. However, as mentioned in the chapter before, changing even one byte or character, results in a completely random output.

Here are some more properties of the hash function;

1. Deterministic: The same input will always produce the same output hash value.
2. Fixed Output Length: Regardless of the input size, the output hash is of a fixed length (e.g., 256 bits for SHA-256).
3. Efficent: Hash functions can process input data quickly.
4. It should be infeasible to generate the original input data given only the hash value.
5. It should be infeasible to find two different inputs that produce the same hash value.
6. A small change in the input should produce a significantly different hash value.

There are different types of Hash algoritms, but we will focus on the Secure Hash Algoritm 2, which includes SHA-256 and KECCAK-256.

Note: The security strenghts of the functions are defined as https://csrc.nist.gov/projects/hash-functions

[TODO]

sha256('Hello, World!") =
dffd6021bb2bd5b0af676290809ec3a53191dd81c7f70a4b28688a362182986f

Note, the output of the function is in hexidecimal (base16) so its case insenstive.

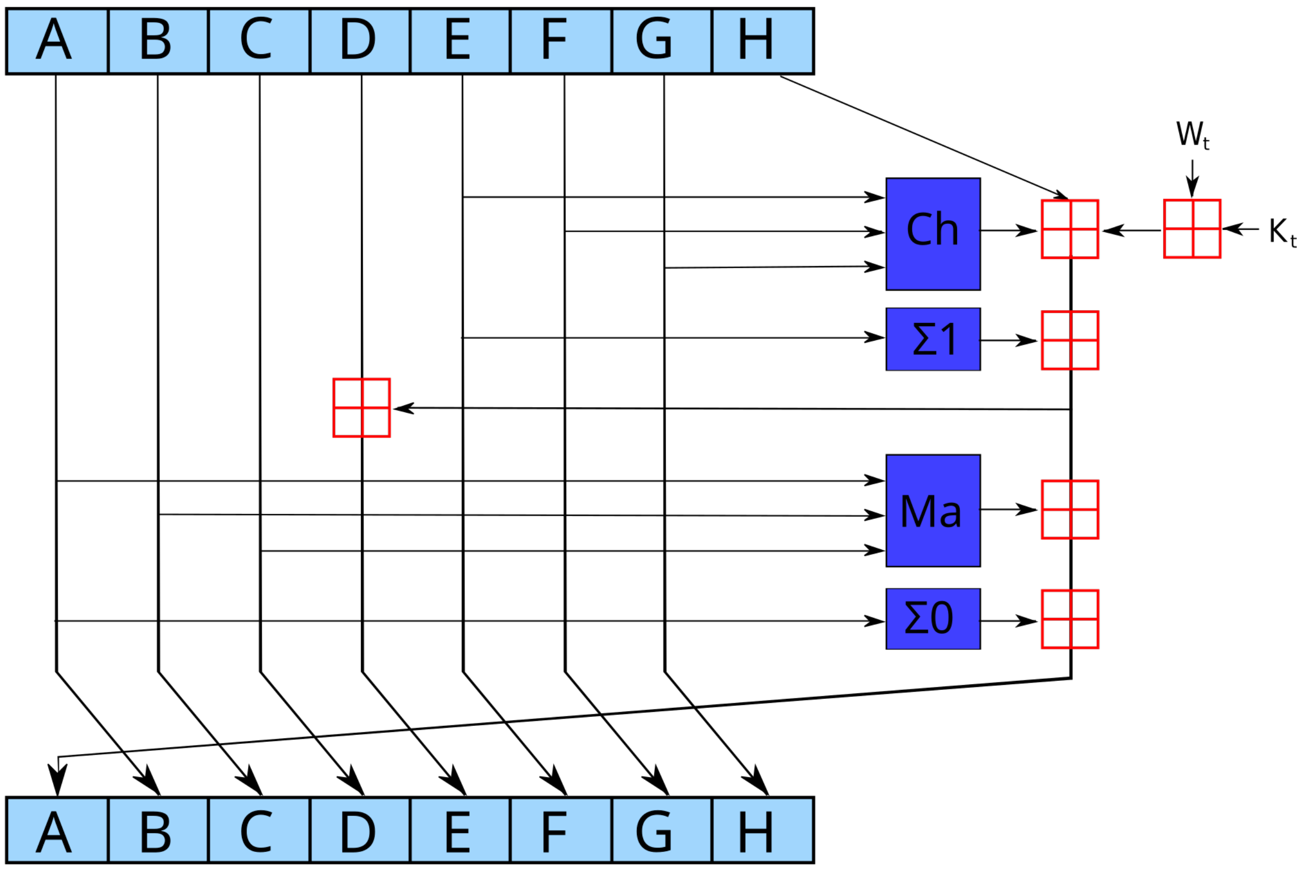
Remove the comma and we get the following output;

sha256("Hello World!") =
7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284addd200126d9069

Hash function are used to store passwords in a database, so that they're not stored in plain text. An attacker or even an inside employee, should be able to look up a users password and login to the system. We will discuss Rainbow table attacks later.

[TOOD node js passwod code example]

So, how do hash functions actually work and why cant they be reversed? You'll find many bad examples on the internet of an implmenation of a hash function, or just using libs or native functions in the langaguges. If you're not intersted in the algorithm, feel free to skip this part, but for me I *had* to know. Its critical to bitcoin mining, so I love to understand the algorithim in full.



First we start with two sets of magic numbers. The inital hash values in hexadecimal, we will call H0 to H7 which are derived from the first 32 bytes of the fractional parts of the square roots of the first 8 primes resulting in irrational numbers. 2, 3, 5, 7, 11, 13, 17, 19

```
H0 = 0x6a09e667
H1 = 0xbb67ae85
H2 = 0x3c6ef372
H3 = 0xa54ff53a
```

```
H4 = 0x510e527f
H5 = 0x9b05688c
H6 = 0x1f83d9ab
H7 = 0x5be0cd19
```

sqrt(2) = 1.4142135623730951 thus the fractional part to eight decimals is 0.4142135623730951. Now we convert that number to binary.

bin(0.4142135623730951) = 01101010000010011110011001100111 and conver to hexadecimal,

hex(01101010000010011110011001100111) = 0x6a09e667

The next magic numbers are sixty four 32 bit words dereived from the fractional parts of the cube roots, of the first 64 prime numbers. 2 ... 311. We will call these K0 ... K64

From the diagram above, we also note in the two blue boxes the functions Ch and Ma. Those are the Choosing function (Ch) and the Majority function (Ma), formally defined as [TODO]

For completeness, lets write these functions in code:

Coinflip over the phone?

Private and public keys

Blockchains use a cryptography method called public key, private key cryptography to mathematically prove that the user or machine interfacing with the blockchain is who they say they are.

A user has a private key, which is a large number, represented as hexadecimal number, base 64 string or sometimes as a decimal. Because of their size, by using a larger base or radix, theyre easier to store in our application. Also, we will define memory on a chain as 32 byte works, so a 256 bit number conveniently fits within that memory allocation.

Schoor signatures are now implemented in the bitcoin protocol as of …
A relative new algorithm EdDSA known as Edwards-curve digital signature algorithm. It is based on the newley (2022) implementation of schnoor signatures.

Now lets discuss some crypto algorithms!

Discuss RSA

There are many algorithms to encrypt data

- bcrypt
- triple des
- pgp / GPG

Asymtric vs Symetric

Merkle Trees

Now that we have an understading about hashing there is one data strcuture or algorithim we need to unerstand and why its application and that is merkle trees. Merkle trees are a data stucture Pattented by Ralph Merkle in 1979.

The tree is created by hashing together two values in a binary tree to form a node. That node is then hash with another node and so on to from the tree.

Lets say we have the value "ACE OF SPADES". We will call this leaf 1 or L1. Our next leaf is "KING OF SPADES", set to L2. Assume the SHA-256 hash function for this example.

hash("ACE OF SPADES") =
6ee0afb99a734986c820025ef6d12812c43c10357779f05613be26cd

hash("KING OF SPADES") =
4dd528b068d435b667089e909dea6711531349ad0d96ea7452b60033e75ef48c

Now we combine hash(L1) with hash(L2) or hash(L1 + L2),

hash(6ee0afb99a734986c820025ef6d12812c43c10357779f05613be26cd4dd528b068d435b667089e909dea6
= a19335e738444f879b26e3ca5f01f3a95524cb25676b91a64fd5a74a55a27cca

Now, whats so important about this tree? There are a few interesting properties or applications that we can use. Lets say we want to know that our bitcoin exchange we use are solvent, and don't rug us like the FTX exchange. We can create a merkele tree of your bitcoin address and the balance that the exchange shows.

The exchange can the publish the proof which is an arbitary bytes 32 string like a hash. This does not disclose the full liquidity of the exchange or amount of bitcoin in custody, but the user can verify that this is correct.

We can argue that any random user can repeat the same and thus we can assume some cernity that the value held in your custodial bitcoin address is true.

Symetric

Symmetric encryption is a type of encryption where the same key is used for both encryption and decryption of data. This means that both the sender and the recipient must have access to the same secret key to encode and decode the information. They are generally faster algorithms than their asymmetric cousins.

This can be useful for encrypting a file on disk, where the user is the person that locks and unlocks the data, or between parties who have shared a secret key beforehand. This often happens when someone sends an encrypted file by email and they will say "I'll send you the password via Signal" or other end to end encrypted services. Good enough for most everyday communications.

TripleDES and BlowFish are two such symmetric encryption algorithms. TripeDES involves there stages of encryption / decryption which can be described as follows:

1. Encrypt the first key (K1)
2. Decrypt the second key (K2)
3. Encrypt the third key (K3)

Each stage operates on a block of plaintext or intermediate cipher text of size 64 bytes.

Here is two NodeJS examples on using TripleDES to encrypt a file.

- `touch tripledes.js`
- `node tripledes.js`

```
const crypto = require("crypto");
// Define the key and IV (Initialization Vector)
const key = Buffer.from("0123456789abcdef01234567", "utf8"); // 24
bytes key for 3DES
const iv = Buffer.from("12345678", "utf8"); // 8 bytes IV for DES

// Function to encrypt plaintext using 3DES
function encrypt(text) {
const cipher = crypto.createCipheriv("des-ede3-cbc", key, iv);
let encrypted = cipher.update(text, "utf8", "base64");
encrypted += cipher.final("base64");
return encrypted;
}
```

```
// Function to decrypt ciphertext using 3DES
function decrypt(encryptedText) {
const decipher = crypto.createDecipheriv("des-ede3-cbc", key, iv);
let decrypted = decipher.update(encryptedText, "base64", "utf8");
decrypted += decipher.final("utf8");
return decrypted;
}

// Test the encryption and decryption
const plaintext = "Hello, this is a plaintext message!";
console.log("Plaintext:", plaintext);

const encryptedText = encrypt(plaintext);
console.log("Encrypted:", encryptedText);

const decryptedText = decrypt(encryptedText);
console.log("Decrypted:", decryptedText);
```

We can see on line 2 we set the key to a 24 byte array, as that is the key size required for TripleDES. As the functions are deterministic, you should see the outputs as:

```
Plaintext: Hello, this is a plaintext message!
Encrypted: HYtEf/sQLP/Sybpf7b+Yql5zQajIKC0oAbN1iWVUykQ1EwbAGZHl8A==
Decrypted: Hello, this is a plaintext message!
```

Let's do the same code snippet using Python, so we're sure the code is language agnostic. After setting up the python virtual environment;

- touch tripledes.py
- pip install pycryptodome
- python tripledes.py

```
Plaintext: Hello, this is a plaintext message!
Encrypted: HYtEf/sQLP/Sybpf7b+Yql5zQajIKC0oAbN1iWVUykQ1EwbAGZHl8A==
Decrypted: Hello, this is a plaintext message!
```

We can now be sure if we send the encrypted text HYtEf/sQLP/Sybpf7b+Yql5zQajIKC0oAbN1iWVUykQ1EwbAGZHl8A== that we encrypted in NodeJS to our friend who then uses Python to decrypt that they will receive the correct message, and the authors of the libraries we have used have implemented the algorithm correctly, or both incorrectly which is unlikely.

Asymetric

In our GPG example in Chapter 1, we encrypted a message to myself using my PGP public key. This is an example of Asymentric Encryption!

The advange of using an Asymetric schema, is that the parties like yourself and myself, never have to meet to send a secure communication. It is considered safe that I can publish my public key to the world + dog.

Replay attacks

To bridge between the ethereum main net contract and our layer 2, we ill use signed messages.

In our deposit contract later on we will use a nonce to prevent a replay attack, but first of all lets discuss what that is. As the name suggests, an attacker can try to send the same request or payload and create subsequent requests.

Send a request to bob to transfer $100
Send the same signed request to alice for the same $100

Essentially, this is an example of a double spend. An attacker has been able to perform the same action twice. We can solve this by using a nonce. A use once number.

Lets look at a bit of javascript and solidity code to defend against this.

[TODO]

Base64, Hex and other encodings

Other alpahbets too, like base 58 for bitcoin.

Zero Knowledge Proof

Now lets dicuss zero knowledge proofs. This is the defnintion from Wikipedia: "In cryptography, a zero-knowledge proof (also known as a ZK proof or ZKP) is a protocol in which one party (the prover) can convince another party (the verifier) that some given statement is true, without conveying to the verifier any information beyond the mere fact of that statement's truth". I like the word convince here, rather than "proove".

There are a few annologys that I like when explaining what Zero Knowlege proofs are.

Imagine you have two balls that appear identical to your colorblind friend - but you can see one is red and one is green. You want to prove to them that the balls are actually different, without revealing anything about how they're different (since you can't explain color to someone who's never seen it).

Here's the proof: Your friend holds both balls behind their back, randomly switches them around (or doesn't), then brings them forward. You tell them whether they switched or not. You repeat this 20 times, and you're correct every single time.

After enough rounds, your friend becomes convinced the balls must be different (the odds of you guessing right 20 times by chance are 1 in a million).

Python code:

```
def probability_of_random_success(n_trials):
"""
Calculate the probability of randomly guessing correctly n times in
a row
when each guess has a 50/50 chance (switched or not switched).

Args:
n_trials: Number of consecutive correct guesses

Returns:
Dictionary with probability as fraction and decimal, plus "1 in X"
format
"""
# Probability of one correct guess is 1/2
# Probability of n correct guesses is (1/2)^n
probability = (1/2) ** n_trials
```

```
# Express as "1 in X"
one_in_x = 1 / probability

return {
'probability': probability,
'percentage': probability * 100,
'one_in': one_in_x,
'formatted': f"1 in {one_in_x:,.0f}"
}
# Test with 20 trials
result = probability_of_random_success(20)
print(f"Probability: {result['probability']}")
print(f"Percentage: {result['percentage']:.6f}%")
print(f"Odds: {result['formatted']}")
```

The outputs are:

```
Probability: 9.5367431640625 * 10 ^ 7
Percentage: 0.000095%
Odds: 1 in 1,048,576
```

Yet they still have zero knowledge about the actual difference - they never learned what "red" and "green" are, or anything about color. They just know the balls are distinguishable to you.

For a less abstract use case, the age verification "problem"

Lets say you want to enter your favourite bar, and the door man (bouncer) needs to check your ID for your age. Your age is actually a funciton of you birth day. However, by revlaing your birthyday you're acutally giving away more metadata than you may be compforatble

The question is "are you above this age", not what is your birthday.

Now we have a basic understanding of what they are, perahps you can already guess in how the apply to a card game? Having a shuffle deck that we can prove is shufled, but not know what the deck state is, is an intresting part of the system and something we can solve with ZK.

Lets write out first Zero Knowledge proof, but first some some termonology

The Proover:

Verifier:

Gadget:

Circut:

Lets try write that proof as a circut. Theres a langauge or framework that I use called Circom. Our protocol must do the following:

1. Create the outcome or choices at time t.
2.
3.

In the section on hashing, we discussed an protocol for playing a coinflip game over the phone. We can also implement this as zero knowledge proof, and the extend that idea to a zero knowledge deck of cards.

As our book is heavly card focused, we will use the Fisher Yates algoritim for shuffling the deck. While naive approaches like repeated random swaps or sorting by random keys can introduce bias or be computationally expensive, the **Fisher-Yates (Knuth) shuffle** stands out as the gold standard for creating uniformly random permutations. Invented by Ronald Fisher and Frank Yates in 1938 and later popularized by Donald Knuth, the algorithm works by iterating through the array from the first element to the second-to-last, and at each position *i*, selecting a random index *j* from *i* to *n-1* and swapping the elements at positions *i* and *j*.

This approach guarantees that every possible permutation of *n* elements has an equal probability of 1/n! of being generated, making it perfectly unbiased. With O(n) time complexity and O(1) space complexity (when shuffling in-place), The algorithm is **deterministic** - given the same sequence of random values, it produces the same shuffle - making it ideal for cryptographic verification. This determinism allows players to prove a shuffle was performed correctly without revealing the random values used, enabling trustless, verifiable card games on-chain.

```
pragma circom 2.0.0;
include "circomlib/circuits/comparators.circom";
include "circomlib/circuits/gates.circom";
```

```
// Fisher-Yates Shuffle Verification Circuit
// Proves that outputDeck is a valid permutation of inputDeck
// using Fisher-Yates algorithm with given random values
template FisherYatesShuffle(n) {
// n = number of cards (52 for standard deck)

signal input inputDeck[n]; // Original ordered deck
signal input outputDeck[n]; // Shuffled deck
signal input randomValues[n]; // Random values used for shuffling
(private)

// Intermediate deck states during shuffling
signal deck[n][n];

// Initialize first row with input deck
for (var i = 0; i < n; i++) {
deck[0][i] <== inputDeck[i];
}

// Perform Fisher-Yates shuffle step by step
for (var i = 0; i < n - 1; i++) {
// Calculate swap index: j = i + (randomValues[i] % (n - i))
signal swapIndex[i];
signal range[i];
range[i] <== n - i;

// Modulo operation: randomValues[i] % range[i]
signal quotient[i];
signal remainder[i];
quotient[i] <-- randomValues[i] \ range[i];
remainder[i] <-- randomValues[i] % range[i];

// Verify: randomValues[i] = quotient[i] * range[i] + remainder[i]
randomValues[i] === quotient[i] * range[i] + remainder[i];

// Verify remainder is in valid range [0, range[i])
component ltCheck = LessThan(32);
ltCheck.in[0] <== remainder[i];
ltCheck.in[1] <== range[i];
ltCheck.out === 1;

// Calculate actual swap index
swapIndex[i] <== i + remainder[i];

// Perform swap between position i and swapIndex[i]
for (var j = 0; j < n; j++) {
if (j == i) {
// Position i gets value from swapIndex[i]
deck[i+1][j] <== deck[i][swapIndex[i]];
} else if (j == swapIndex[i]) {
// Position swapIndex[i] gets value from i
deck[i+1][j] <== deck[i][i];
} else {
// Other positions remain unchanged
deck[i+1][j] <== deck[i][j];
}
}
}

// Verify final output matches the shuffled deck
for (var i = 0; i < n; i++) {
outputDeck[i] === deck[n-1][i];
}

// Verify that output is a valid permutation (all unique values)
component uniqueCheck[n][n];
signal isEqual[n][n];
signal matchCount[n];

for (var i = 0; i < n; i++) {
var count = 0;
for (var j = 0; j < n; j++) {
uniqueCheck[i][j] = IsEqual();
uniqueCheck[i][j].in[0] <== outputDeck[i];
uniqueCheck[i][j].in[1] <== inputDeck[j];
isEqual[i][j] <== uniqueCheck[i][j].out;
count += isEqual[i][j];
}
// Each output card must appear exactly once in input
matchCount[i] <-- count;
matchCount[i] === 1;
}
}
// Main component for standard 52-card deck
component main {public [inputDeck, outputDeck]} =
FisherYatesShuffle(52);
```

To compile

```
circom deck_shuffle.circom --r1cs --wasm --sym -o build
```

So what have we built here? And what is wasm? Mozzilla "WebAssembly is a type of code that can be run in modern web browsers. It is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++, C# and Rust with a compilation target so that they can run on the web. It is also designed to run alongside JavaScript, allowing both to work together."

Ill add both the react / vite version and the pure HTML version to the GitHub repo

```
# Clone the project
git clone /book/chapterx
cd zk-poker-shuffle
# Install dependencies
yarn install
# Install circom globally (if not already installed)
yarn install -g circom snarkjs
```

Project stucture

```
zk-poker-shuffle/
├── circuits/
│ ├── deck_shuffle.circom # Basic circuit
│ └── optimized_deck_shuffle.circom # Optimized version
│
├── public/
│ ├── circuits/
│ │ ├── deck_shuffle_js/
│ │ │ └── deck_shuffle.wasm # ← Used in browser
│ │ ├── deck_shuffle.zkey # Proving key
│ │ └── verification_key.json # Verification key
│ └── demo.html # Browser demo
│
├── src/
│ ├── zkProof/
│ │ ├── browserProofGenerator.ts # Browser proof generation
│ │ ├── zkDeckShuffler.ts # Shuffle implementation
│ │ └── zkShuffleHelpers.ts # Advanced features
│ └── components/
│ └── PokerGame.tsx # React component
│
├── scripts/
│ └── test-proof.js # Test script
│
├── contracts/
│ └── DeckShuffleVerifier.sol # Solidity verifier
│
└── package.json
```

Here is simple HTML, followed by React

```
<script type="module">
import * as snarkjs from 'https://cdn.jsdelivr.net/npm/snarkjs@latest/build/snarkjs.min.js';

const { proof, publicSignals } = await snarkjs.groth16.fullProve(
input,
'/circuits/deck_shuffle.wasm',
'/circuits/deck_shuffle.zkey'
);
</script>
```

Section Notes

Outline Notes

About

Description

A description of who this book is for and what its about.

The idea

Description

What is your original contribution? Why should the examiner care about your research? What is the thesis problem statement? What do you (not) hope to achieve? What are the research questions and hypotheses? What are your epistemological and ontological positions? What is your contribution to the field? How is the thesis laid out?

Architecture

Description

Lets define the layers of our stack, discuss what langauges and frameworks that are required.

Crypto cryptography

Description

This chapter we discuss some of the fundamentals behind blockchains and crypto currenices, and some of the maths beind them.

Domain specific language

Description

This will be our attempt at writing our own purpose built language, fit for the job.

Langauge design is an interesting topic itself. We will discuss lexers, pasers, ASTs and more.

Blockchain email. The anti pattern

Description

In this chapter, we will look at blockchain anti patterns. Some concrete applications as use cases where we dont need a chain or token.

Ethereum smart contracts

Description

This book would not do its readers justice without talking about the granddaddy of smart contracts.

Consensus mechanisms

Description

How does a distributed network stay in sync? What are the ways we can achieve network consensus?

Notes

Carnot

Our blockchain

Description

The Poker Virtual Machine

Description

Our solution invloves a purpose built game engine, which we call the poker vm.

Writting a compiler

Description

Coming from a microsoft stack background in the late 90s and early 2000s, MS build was closed source. I didnt really understand how compilers worked, or even considered that there could be different versions of them or even bugs in them. I just hit f5 and out popped an exe.

The desktop client

Description

In this chapter, we will discuss how to build a desktop application, and why we need one rather than a web app.

Packaging and releasing

Description

Now we have finally build the thing, we have to distribute the app to the world. Lets expore some ways we can do this.

Working remotely, distributed teams and goverance

Description

Crypto apps are senonumus with working in remote, distributed teams. Due to the complexities and lack of legal guidelines, teams often choose to run annon with shell companies in juristictions like Panamma and the British Virgin Island. Ill expore some of the challanges around this and is code truely the law?

References

Description

All references cited in the text must appear in the reference list. Do your references follow your university’s house style? Are all references contained in the text listed in the bibliography, and vice versa.

Chapters Notes

About?

Description

Bit of an overview, we discussed some of this stuff before.

Randomness

Description

The Avalanche Effect

Description

Hash functions

Description

Private and public keys

Description

Need to do some research here on different methods.

Merkle Trees

Description

In cryptography and computer science, a hash tree or Merkle tree is a tree in which every "leaf" node is labelled with the cryptographic hash of a data block, and every node that is not a leaf (called a branch, inner node, or inode) is labelled with the cryptographic hash of the labels of its child nodes. A hash tree allows efficient and secure verification of the contents of a large data structure. A hash tree is a generalization of a hash list and a hash chain.

Demonstrating that a leaf node is a part of a given binary hash tree requires computing a number of hashes proportional to the logarithm of the number of leaf nodes in the tree.[1] Conversely, in a hash list, the number is proportional to the number of leaf nodes itself. A Merkle tree is therefore an efficient example of a cryptographic commitment scheme, in which the root of the tree is seen as a commitment and leaf nodes may be revealed and proven to be part of the original commitment.[2]

The concept of a hash tree is named after Ralph Merkle, who patented it in 1979.

Notes

https://ethereum.stackexchange.com/questions/149500/how-to-implement-poseidon-hash-function-into-a-smart-contract?rq=1

Symetric

Description

Asymetric

Description

Replay attacks

Description

Base64, Hex and other encodings

Description

Zero Knowledge Proof

Description
