Blockchain Guide Documentation

Release 1.0a1

Firescar96

Nov 04, 2017

Contents:

This documentation is currently in version 1.0-alpha any comments and suggestions can be referred to Firescar96 at nchinda2@gmail.com.

The goal of this documentation is to introduce you to blockchain technology and get you started writing smart contracts for Ethereum. Writing smart contracts requires thinking in a way unique from other systems as it also requires understanding of the benefits and limitations of the blockchain technology the smart contracts run on. This guide assumes the reader is a proficient coder in any other programming language, but does is not familiar with blockchain technology.

CHAPTER 1

Cryptology Primer

Cryptocurrencies uniquely combine pieces of cryptography and game theory to create a distributed, decentralized peer to peer payment network.

1.1 Alice and Bob

It's imperative when talking about crypto to use the correct naming for example people. At minimum the first two participants should be named Alice and Bob. Any attacker must be called Eve. From there continue from C down the alphabet. For example: Charlie, Dawn, Frank, Gilbert, Jill, etc.

1.2 Cryptographic Primitives and Data Structures

Like every other field in existence, cryptography is just applied math. Expressing data as sequences of numbers allows for simple, yet powerful applications of mathmetical formulas to accomplish a wide range of goals.

1.2.1 Hash Functions

A hash function maps an arbitrary amount of data to a value of fixed size. It's deterministic, the same key as input will always return the same value as output.

Example: A function *H* that takes characters, words, or sentences and returns the first character.

- H(a) -> a
- H(alphabet) -> a
- H(The light at the end of the tunnel.) -> T

This isn't considered a "good" hash function, because if used in practice there will probably be many *collisions* where two keys map to the same value. A *perfect* hash function maps every key to a different value. Perfect hash functions don't exist unless the range of keys is constrained to a fixed number of possibilities.

An example of a non-perfect but still very good hash function is SHA-3. Multiple variants exist such as SHA3-256 and SHA3-512 which return 256 bit and 512 bit values respectively. Because hash function can have collisions, they also function as *one-way functions* because there is no way to retrieve the original keys from the value. At the same time 256 is enough bits of entropy that collisions are practically not worth considering, as there are an estimated 2^{272} atoms in the entire universe. 256 bits of entropy is also secure enough, and 512 bits is for sure enough, to deter attempts to brute force determine keys from the value.

1.2.2 Randomness

It's hard to get a truly random number that can't be predicted over time. The strength of a random number lays in it's source of entropy, this can be: mouse movements, the temperature, air currents, the time, etc.

Hashing can be used to combine sources of entropy into a final psuedo-random number and obscure the origin of the sources of entropy.

1.3 Public-key Cryptography

This refers to any system that uses key pairs. The private key is kept secret and used by the holder to authenticate their identity. From the private key an infinite number of public keys can be derived. By design public keys can be shared freely without compromising the private key.

1.4 Signing

Public-key cryptography can be used for authentication of a person's identity, *assuming that a person's private key hasn't been compromised.* The private key can be used to sign a document, and the signature can be verified with the public key.

Appendix A contains an example of a signing algorithm. For a challenge try implementing the given formulas in the programming language of your choice.

1.5 Encryption

Going further than just authentication, public-key cryptography can be used to encrypt a message so that only a select party can read it. Messages can be encrypted with a private key and decoded with a public key and vice versa.

Appendix B contains an example of an encryption algorithm. For a challenge try implementing the given formulas in the programming language of your choice.

1.6 Diffie-Hellman Key Exchange

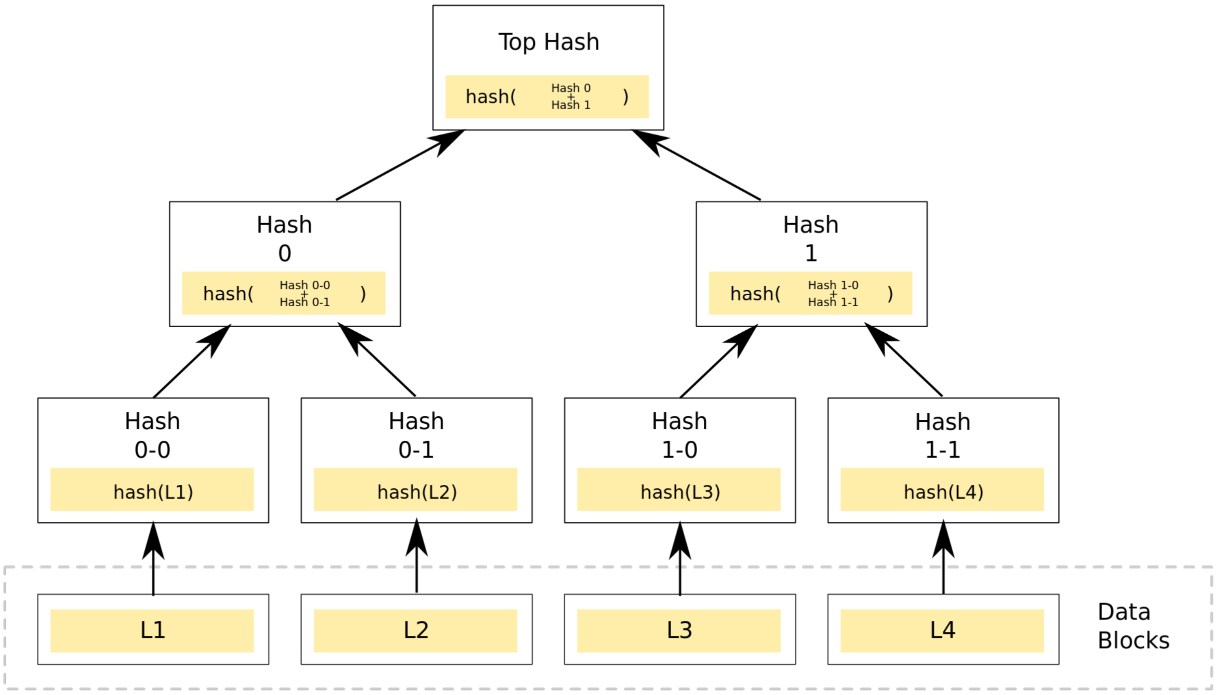
Using public-key cryptography, Alice and Bob can create a cryptographically secure channel and send encrypted messages to each other. They do this by generating a shared secret that only they know and then freely using this as a secret key in one of an infinite number of encryption schemes. Even if Eve is listening to Alice and Bob's messages she can't decrypt their communications.

A really bad, but working example encryption scheme using the secret key might be multiplying the message by the key to encrypt and dividing by the key to decrypt.

Appendix C contains the details of Diffie-Hellman. For a challenge try implementing the given formulas in the programming language of your choice.

1.7 Merkel Trees

These are a type of trie data structure where every non leaf element contains a hash of its children. There are no other constraints on its formation. Bitcoin uses these in multiple ways, for example: the entire blockchain forms a Merkle Tree with each block linking to the block that came before it enforcing block chronology.



1.8 Game Theory

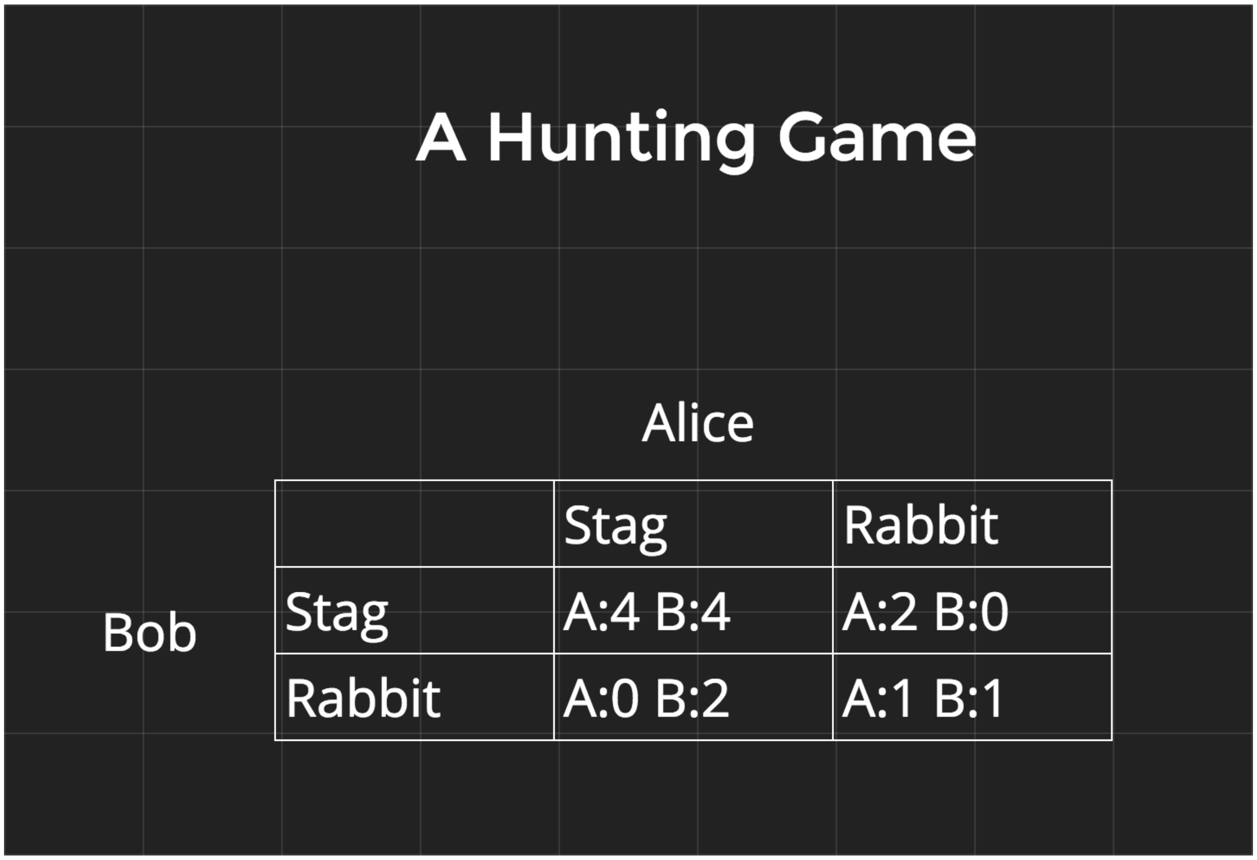
1.8.1 Nash Equilibrium

This is a set of actions in a non-cooperative game where every participant has nothing more to gain by choosing a different action. Even if the participants could telepathically see into the strategies of others. How this comes into play in Bitcoin will come up in the next section.

A Hunting Game Alice and Bob go out on a hunt. Each must individually choose to hunt a stag or hunt a rabbit before leaving home. Each player must choose an action without knowing the choice of the other. If an individual hunts a stag, they must have the cooperation of their partner in order to succeed. An individual can get a rabbit by themself, but a rabbit is worth less than a stag.

The first example gives the points diagram for this hunting game. In this configuration there are two Nash Equilibria at Stag:Stag and Rabbit:Rabbit.

In the second example the points have been psuedo-arbitrarily adjusted so that there is only one Nash Equilibrium.

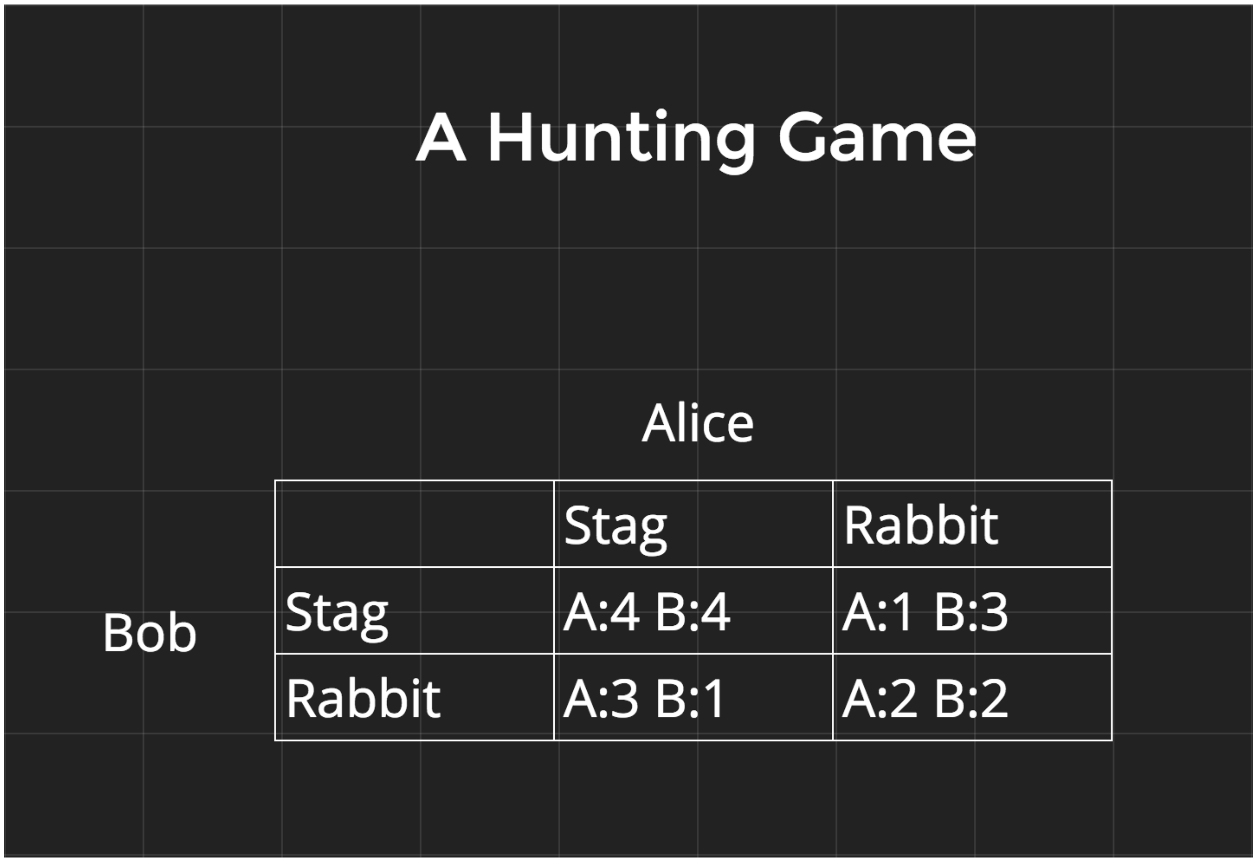


A Hunting Game

Alice

Bob

	Stag	Rabbit
Stag	A:4 B:4	A:2 B:0
Rabbit	A:0 B:2	A:1 B:1



A Hunting Game

Alice

Bob

	Stag	Rabbit
Stag	A:4 B:4	A:1 B:3
Rabbit	A:3 B:1	A:2 B:2

1.9 Decentralize all The Things

Here are three different type of networks to think about when discussing how blockchain based systems differ from those they replace.

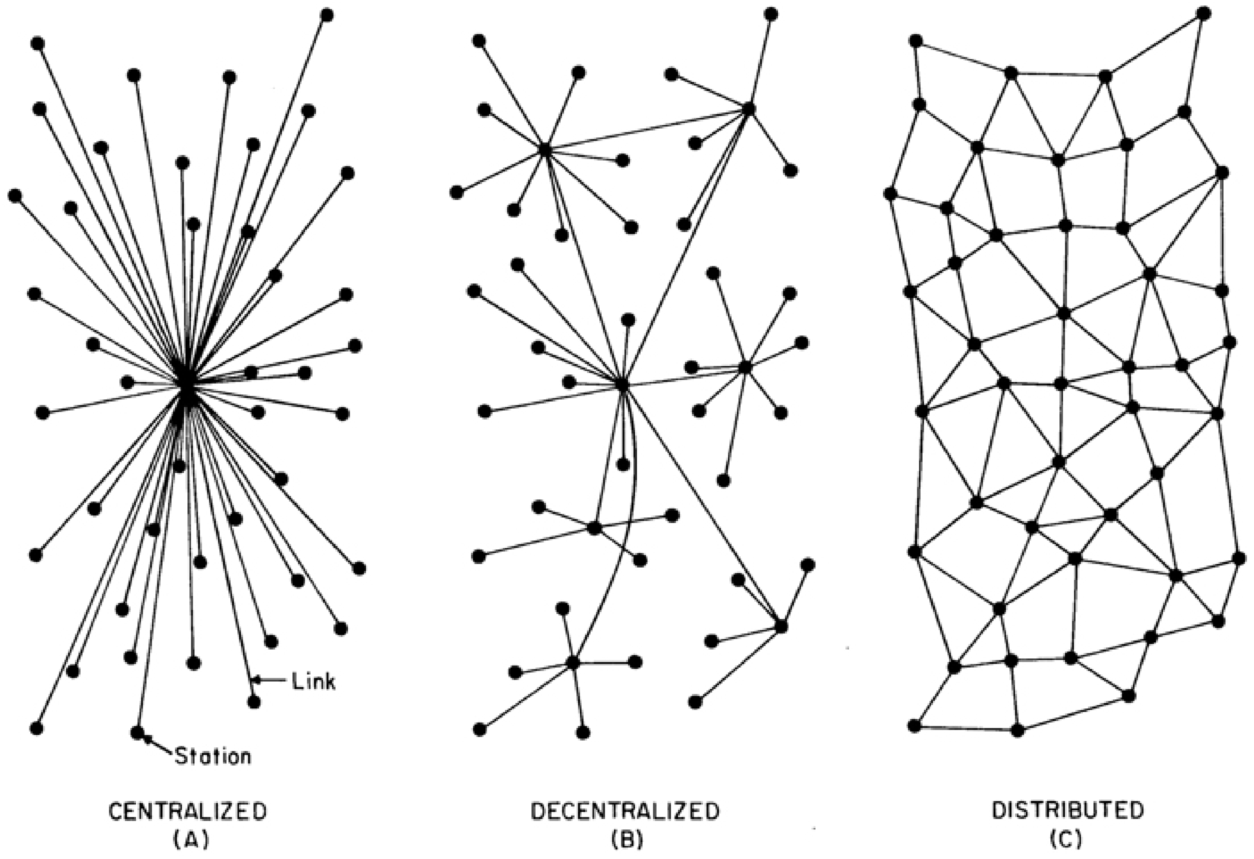


FIG. I — Centralized, Decentralized and Distributed Networks

1.9.1 Appendix

1.10 A) Schnorr Signatures

Schnorr Signatures are special because they are one of the simplest and efficient signature schemes to be proven secure.

define:

public generator G
Alice's private_key: a
Alice's public_key: $A = G^a \ (mod\ G)$
random signing number: k
generated signing key: $r = G^k \ (mod\ G)$
a message: M

concatenation symbol: $||$

Note: Remember that everything defined above, including the message can and must be expressed as a number. |

Signing With these Alice can compute a signature: S

$$E = H(r||M)$$
$$S = k - E * a$$

Alice publishes (M, E, S)

Verification

$$r_v = G^S * A^E$$
$$e_v = H(r_v||M)$$

check that $e_v = E$

Proof

$$r_v = G^S * A^E = G^{k-E*a} * G^{E*a} = G^k = r$$
$$e_v = H(r_v||M) = H(r||M) = E$$

1.11 B) RSA

Named after it's creators, RSA is extremely widely used. Even though more efficient algorithms have since been discovered RSA is still useful as an encryption scheme that's relatively simple to teach.

define:

two large primes: p, q
the totient function: Φ
secret encryption key: e
message: m
decryption key: D
encrypted message or "ciphertext": C

Encryption Alice can then compute:

$$N = p * q$$
$$\Phi(N) = \Phi(p) * \Phi(q) = (p-1) * (q-1)$$
$$D = \frac{1}{e}\ (mod\ \Phi(N))$$
$$C = m^e\ (mod\ N)$$

Alice then publishes the (C, D, N)

Decryption

$$m = C^D\ (mod\ N) = m^{e*D}\ (mod\ N) = m^1\ (mod\ N) = m$$

Proof This is harder to explain, but it is detailed on the RSA Wikipedia Page.

1.12 C) Diffie-Hellman Key Exchange

define:

> public generator G
> Alice's private_key: a
> Alice's public_key: $A = G^a \ (mod\ G)$
> Bob's private_key: b
> Bob's public_key: $B = G^b \ (mod\ G)$

Then Alice and bob can trade their public keys and independently construct a shared secret for encrypting their messages.

$$Alice: \ (a, B) - > B^a = G^{ab} \ (mod\ G)$$
$$Bob: \ (b, A) - > A^b = G^{ab} \ (mod\ G)$$

Once they both have a shared secret that only they know there is an infinite number of ways they can use the secret to encrypt and decrypt messages to each other. The unsolved Discreet Log Problem ensures that even though Eve learns the public keys G^a and G^b she cannot use these to compute G^{ab} and decrypt messages.

CHAPTER 2

Bitcoin and the Blockchain

Bitcoin is the O.G. of cryptocurrencies because every other cryptocurrency is fork of or directly borrows fundamental elements. That is why other cyrptocurrencies are collectively referred to as *altcoins* (alternative currencies).

This section is in the form of a presentation.

Video link

Slides

CHAPTER 3

In depth about Blockchain Technology

3.1 Trust

The only way to have objective trust in the state of the blockchain is to download and verify the entire thing from the genesis block to the head.

3.2 Chain Reorgs

A blockchain will *fork* into two chains regularly as it's possible for multiple people to release valid blocks simultaneously. These forks resolve quickly as eventually one of the forks gains more hashpower support and dominates. As nodes switch over to the fork with more hashpower any transactions that were confirmed in the orphaned fork but not in the majority fork will no longer be valid. This is why it's important to never accept a transaction as valid until it has been included in a block; has one confirmation. Every succeeding block after that increases the transactions confirmation count.

3.3 Types of Nodes

Depending on the blockchain technology used there are multiple type of nodes a user might run.

Full nodes are the default type and are required for the network to operate as they store the entire history of the blockchain. A full node can verify the validity of any transaction by walking the blockchain and verifying the validity of all previous transactions. Any new node to the network that wants to sync to the current block must contact a full node.

Pruned nodes are full nodes configured to only keep a fixed number of blocks in history. This allows users who have space constraints to make a trade of between space and security. It is inherently less secure to use a pruned node, consider a node that only keeps one block at a time. That node won't be able to detect forks starting a blocks earlier than the one that it is holding. Additionally if the node accepts a block that doesn't end up being accepted by the network, then it will have to resync from the genesis block all over again.

Light Nodes are not secure as they only store block headers, or don't store the blockchain at all and instead make calls to other full nodes to verify blocks and transactions. A light node cannot verify transactions by walking the blockchain, instead it generates a probability that a transaction is valid based off of how many confirmations it has. At the same time, light nodes are necessary to interact with the blockchain using mobile apps, browser extensions, and other platforms which don't provide much storage space.

3.4 Private vs Public Blockchains

The line between a blockchain and a database is not clear. Blockchain based systems are less efficient than databases, but in exchange have stronger consistency guarantees and work in environments with mutual disinterested (untrustworthy) participants.

3.5 Failure Modes

3.5.1 51% Attack

If a single entity has more than 50% of the hashpower on a Proof of Work based system then they are statistically more likely to produce blocks than the rest of the network. This allows them to do other things, such as rewrite the blockchain to erase certain transactions from history. This is a risk even for entities that have large but below 50% hashpower.

3.5.2 Contentious Hardforks

Forks can be done deliberately, but without overwhelming community support the two resulting blockchains will diverge and become irreconcilable.

3.6 Consensus Algorithms

Alternative systems exist, but most blockchains are based on Proof of Work (PoW) to ensure the network can come to consensus on one true version of history and deter attackers. All versions of PoW involve hashing some combination of random data and data from the desired block to be added to the blockchain, but exact implementation details vary.

- Bitcoin: SHA256
- Ethereum: Ethash
- Dogecoin: Scrypt
- Sia: Blake2b

CHAPTER 4

Limitations of the Blockchain

4.1 Virtual Machines

Code has to run somewhere- code in the cloud is actually running on Google or Amazon's servers and code on the blockchain is actually running on some miner's servers. This is especially salient for blockchain technologies like Ethereum which boast Turing complete languages, but also applies to more basic languages like Bitcoin Script. The computational power of the network is limited by how fast computers on the network can do your operations.

4.2 Size

- the size of the blockchain trends upwards
- some portion of the network must store everything

Don't contribute to blockchain bloat. Data should be deleted from storage as soon as possible to mitigate contributing to the inexorable increase in the size of the blockchain. Some blockchain technologies, such as Ethereum, give a discount to smart contract operations that involve deleting data to incentivize self-regulation of data storage.

4.3 Latency and Consistency

In general I/O operations take a long time, but the time it takes for a transaction to get included into a block after being sent into the network can be orders of magnitude longer. After a transaction is forwarded into the network it gets held in memory by nodes and miners until it eventually gets dropped or included in a block. There is a *non zero* chance of a transaction getting lost needing to be resent. This has implications for user interfaces which need to display some sort of loading animation until the transaction has been confirmed and potentially timeout and resend the transaction. When using a permissioned blockchain the confirmation time will likely be much lower than a public blockchain, but due to the decreased number of nodes the risk of nodes failing and dropping transactions may be higher.

The point to remember is transactions are not final until they get included into a block.

4.4 Legality

Code is not law, and don't let anyone tell you otherwise. Smart contracts are merely a reflection of some soft, squishy, malleable social process. For example: if you write some code that is supposed to release 25% of funds to party A and 75% to party B, but a bug causes the two parties to be flipped, that does not mean done is done and the transfer is complete. Likely there will be some attempt by party B to execute legal action to rectify the situation. Smart contracts and the blockchain in general are an additional resource, but not a replacement for other social and legal constructs like courts.

4.5 The World Outside the Blockchain

Only data that is stored on the blockchain can be manipulated by smart contracts. The blockchain cannot hold the entire contents of the world so data must be brought into the scope of smart contracts in a credible way. In the context of smart contracts entities that can be trusted to inject data about the outside world into the blockchain are called oracles. For example: Nasdaq might act as a price oracle and manage a smart contract that provides conversion rates between different stocks.

4.6 Identity

Identity management is hard, although not impossible. By design there are an effectively infinite number of addresses that a person can generate to use as their public key, giving them the ability to assume as many identities as they would like. Even in permissioned blockchains where the identity of all the participants is known identity management may be difficult. There are two scenarios.

Determining Identities of consenting individuals: Eventually there will likely be one or more organizations who provide identity oracle services on various blockchains. Multiple oracles should be used to attest to a persons identity. Oracles can be used to attest to any of the components of an individual's identity that are currently being used: age, phone number, home address, driver's license, etc. The difficulty of that organizations providing most of these services do not exist yet.

Determining Identities of non-consenting individuals: Most cryptocurrencies are not anonymous. Notable exeptions are Monero and Zcash which use different schemes to obscure the identity of the participants in a transaction as well as the amount transacted.

Multiple identities are nice when testing smart contracts as it lets you represent multiple parties, but for production systems you'll need to write your own smart contract to handle identify or work with an outside company specialized in developing identity oracle solutions.

In Ethereum smart contracts are to be treated as first class citizens. By design the system tries to make it difficult to determine whether some address is a human account or another smart contract, and generally it shouldn't matter. For arbitrary addresses your code should treat both the same.

CHAPTER 5

Ethereum

An up to date technical description of Ethereum can be found in the Ethereum whitepaper. Read this first for an explanation of what Ethereum is and how it uses blockchain technology.

The following is a collections of useful resources and suggestions. Skim through the sections and use it as a reference. The next section will go more in depth about how to actually write smart contracts.

5.1 Wallets and Interacting with the Ethereum Blockchain

There are two broad categories of wallets for Ethereum.

Light Clients don't require you to download the blockchain. In exchange for this convenience is you have **zero anonymity**, someone is able to see and track all the data for every transaction you submit through them and directly link it to you.

- Jaxx- multi currency wallet available as a chrome extension, mobile and desktop app
- Metamask- chrome browser extension backed by Infura for managing ether

Full nodes let you manage coins, mine ether, and keep your own personal storage of the full blockchain. There are two main options

- Go-Ethereum (geth)- golang implementation of a full ethereum node
- Mist- desktop wallet developed by the developers at the Ethereum Foundation
- Parity- rust implementation of a full ethereum node

5.2 Setting up Development environments

You can play around with smart contracts using the online compiler but for actual development the following components are needed:

- A general Purpose or Ethereum Specific IDE

  - suggested: Atom.io, Sublime Text 3, Vim, Emacs, Notepad++
  - anti-suggested: Eclipse, IntelliJ, Visual Studio
- A blockchain
  - For testing use TestRPC, a blockchain simulator.
  - to submit your contracts to the actual Ethereum Network you will need a full node (covered above).
- A compiler, testing, and deployment utility
  - recommended: Truffle, a smart contract development framework
  - alternative: Embark

Once you start dealing with real money make sure you don't accidentally deploy to the real Ethereum blockchain when you meant to deploy to a testnet. Only run one blockchain at a time by only running one type of node at a time. Truffle can help you with managing multiple deployments.

5.3 Solidity Basics

Solidity is the primary language used for coding Ethereum smart contracts. A fair amount of progress in understanding the language syntax can be obtained by understanding knowing the following three constructs.

- Types
- Constructors and Functions
- Special Solidity Calls

There are many more features of solidity that are well documented in the Solidity Documentation. Although extensive, everything is useful and should be read at least once.

5.4 Interacting with your contract - Resources

The documentation for the following software is essential for developing smart contracts using the recommendations given above.

- Solidity
- Truffle
- TestRPC
- Web3
- Bignumber.js

5.5 Security

Hacks are forever (TheDAO was a unique exception). Be careful because unlike fiat currencies there is no bank you can call to save you if you mess up.

- Best practices
- Security Considerations
- Solidity Bugs

5.6 Testing

Truffle, Embark, and Chaithereum are developer frameworks for developing Solidity contracts. Tests for all three are written very similarly.

5.7 Extra Data

If you would like, you can go *even more* in depth into Ethereum by reading this community maintained wiki.

CHAPTER 6

Solidity

Beginner's Guide Smart Contract Programming Tutorial If you're having trouble getting started, this series of youtube videos will walk you through writing a variety of contracts, explaining syntax along the way.

Some things to be wary of when designing Ethereum smart contracts:

- In most cases you'll want to disallow users from transferring negative amounts of money.
- Make sure that updates to the int/uint types don't induce integer overflows
- Solidity cannot store floating point values (decimals). Solutions involve multiplying all values by 10^3 or 10^6 before division. Refer to the docs for more details.
- As a courtesy to users do not let them accidentally send money to the default address *address(0)*. Check for this specifically in functions dealing with transfers to accounts.
- For the actual Ethereum Network (not TestRPC) function calls, contract creation costs gas. If you don't send enough gas the transaction gets atomically rolled back and your gas is lost. If you send too much the extra gets refunded.
- Array deletions in solidity do not automatically left shift elements to fill the empty space, you have to write a function for that yourself.
- *tx.origin*- this function should never be used (except in rare specific cases). It breaks the paradigm of ethereum by distinguishing contracts from users and so disallows, for example, multisig wallet contracts from using your dapp.

CHAPTER 7

Solidity Challenges

These challenges are designed to help you become comfortable as a smart contract developer. They are loosely arranged in order of difficulty. These challenges are not scored, it's up to you to determine when you have sufficiently completed them.

You may need the resources from the previous section to complete these challenges.

1. download testrpc and go-ethereum and succefully install them
2. generate a new address using the geth console
3. check your account balance (web3.eth.getBalance(<your account>))
4. send 5ETH to the 0x0 address

1. Hint: if you use the right blockchain you'll have plenty of ETH

1. create a constructor for a contract that initializes a uint32 variable and create getter and setter functions for that variable.
2. install truffle and use its 'init' command to create a demo app
3. call a the getter and setter functions you created earlier by adding your contract to the Truffle dev environment and using the Truffle console
4. successfully initialize Web3 in the browser by hijacking the js console on https://ethereum.github.io/browser-solidity
5. use Truffle to get a contract abi and address and call a function of the contract using the web console
6. write and compile a very basic smart contract that forwards funds to another address
7. multi contract interaction

1. write a "Ticker" contract with functions that convert between KES and ETH
2. write from scratch or modify the ERC20 token standard to create a token that uses the Ticker contract to return a user's balance in KES

1. implement a function that can handle array deletions by shifting following elements to the left when an array element is deleted

2. create a web interface corresponding to a function in a contract, implementing an "ethereum faucet" allowing a user to input an address into a textbox and click a button to get free eth

CHAPTER 8

Failure By Example

These are some examples of Ethereum applications which serve as examples of both what not to do and how to recover from failure.

8.1 TheDAO

TheDAO has multiple problems with it's design, enough that multiple prominent members of the Ethereum community called for a moratorium until it could be redesigned. The biggest problem that led to its demise is called a "reentrancy attack". By design of Ethereum's Turing complete language contracts can call each other.

The attack involves exploiting this to make calls to a contract in a way the contract developer did not expect. This Vessenes article has a full explanation of how the attack works. For someone who is not used to thinking in terms of blockchain based systems the bug is an easy one to make.

TheDAO ultimately failed and led to a large rift in the Ethereum community, further devolving into a split of Ethereum into two different blockchains and cryptocurrencies. Ethereum and Ethereum Classic. This Coindesk article provides more details on the consequences of TheDAO's failure.

For a hands on example of how the DAO hack worked check out this github project.

8.2 Ethereum Naming Service (ENS) launch

The ENS is another project built on Ethereum that suffered problems after its first launch. Fortunately the developers took notes from the shortcomings of TheDAO incorporated a failsafes into their contracts.

The ENS incorporated smart contracts that could be upgraded or taken down. This was crucial in recovering from the bugs discovered after ENS was deployed. The root ENS smart contracts were under the control of seven keyholders, all public figures, four of whom were required to sign off on any upgrades and changes.

The ENS developers posted a detailed postmortem of the shortcoming of their smart contracts after the system was taken down.

After the solving the bugs discovered in the first launch the ENS team had security audits done by two independent and prominent Ethereum developers. Additionally the ENS team *made the results of the audits public*, contributing to the community knowledge pool on developing safe, secure decentralized applications.

8.3 Gnosis Token sale

Be diligent about promoting your smart contracts. There is a lot of hype in the blockchain space and people who will fund you just because you mention “blockchain”, “Ethereum”, or “Initial Coin Offering (ICO)”. Don’t succumb to the temptation! Use caps to only accept as much money into your smart contract as is necessary for your project to run.