

[www.chainfrog.com](http://www.chainfrog.com)



THE COMPUTER SCIENCE BIT

Find us at [www.chainfrog.com](http://www.chainfrog.com)

# Hash Functions

- A mathematical function is a mapping between a range and a domain (strings connecting things in one box to things in another box)
- A computer function is a self-contained procedure of code that returns an output
- A hash function is a computer function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called:
  - hash values,
  - hash codes,
  - hash sums,
  - or simply hashes

# Hash Functions

- Some popular hash functions:
  - RIPEMD-160 (160 bit hashes)
  - SHA512 (512 bit hashes)
  - BLAKE (256 bit hashes)
- For blockchains we want these functions to be:
  - **Deterministic** (same output for same input, i.e. not time dependent or featuring randomness)
  - **Collision proof** (different data should give a different hash output)
  - **Unpredictable**/one way (we should not be able to engineer a specific hash output)



# Hash Functions

- <http://www.xorbin.com/tools/sha256-hash-calculator>

Input:

Hello, world!

Output:

315f5bdb76d078c43b8ac0064e4a0164612b1fce77c869345bfc94c75894edd3

Input:

hello, world!

Output:

68e656b251e67e8358bef8483ab0d51c6619f3e7a1a9f0e75838d41ff368f728

Input:

hello world

Output:

b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9

# Proof of Work

Based on Hashcash algorithm, invented by Adam Back in 1997.

- Take your data and add a nonce: data + nonce
- Hash it twice:  $\text{sha256}(\text{sha256}(\text{data} + \text{nonce}))$
- Check if your resulting hash output is smaller than a given target (the difficulty)
- If not, try a new nonce, and repeat, again and again.  
So it takes time...

# Proof of Work

For example, find a nonce to add to “hello world” that gives a  $\text{sha256}(\text{sha256}(\text{input}))$  hash that starts with 0:

Input:

hello world19

Output:

0b6bac8ac40b7b0acc7a217f8321c62f9742dc7d87da467eb1c82bd2f651769c

- This took 19 attempts.
- Any particular attempt has a 1 in 16 chance of succeeding.  $p = 1/16$
- Expected number of attempts to succeed:  
 $E(X) = 1/p = 16$ .
- How about finding a  $\text{hash}(\text{hash}(\text{input}))$  starting with 000?



# Proof of Work

- How about find a double hash starting with 000?
- Answer: 1 in 4096 chance. (1 in  $16 \times 16 \times 16$ )  
Expected time to find: 4096 goes.
- Finding a `hash(hash())` starting with  $k$  zeroes on average takes  $16^k$  goes
- But only 1 go to check a proposed block if someone else finds and submits it.
- Summary: finding a block is hard, checking a block is easy...

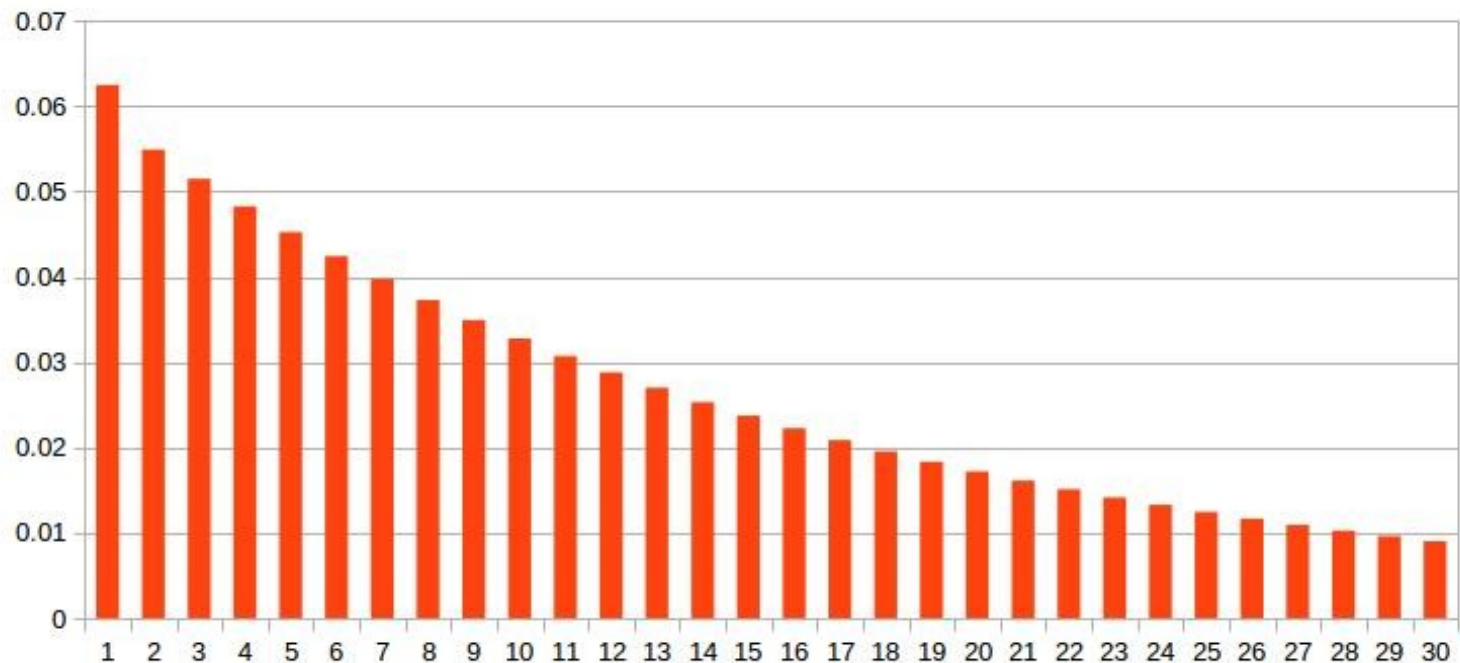
# Dynamic Adjustment of Difficulty

- If more peers (“miners”, or “validators”) join, a correct block is found more quickly.
- If some peers leave, a correct block is found more slowly.
- To compensate the target (the difficulty) is adjusted every N blocks to keep the expected time to generate a block about the same.



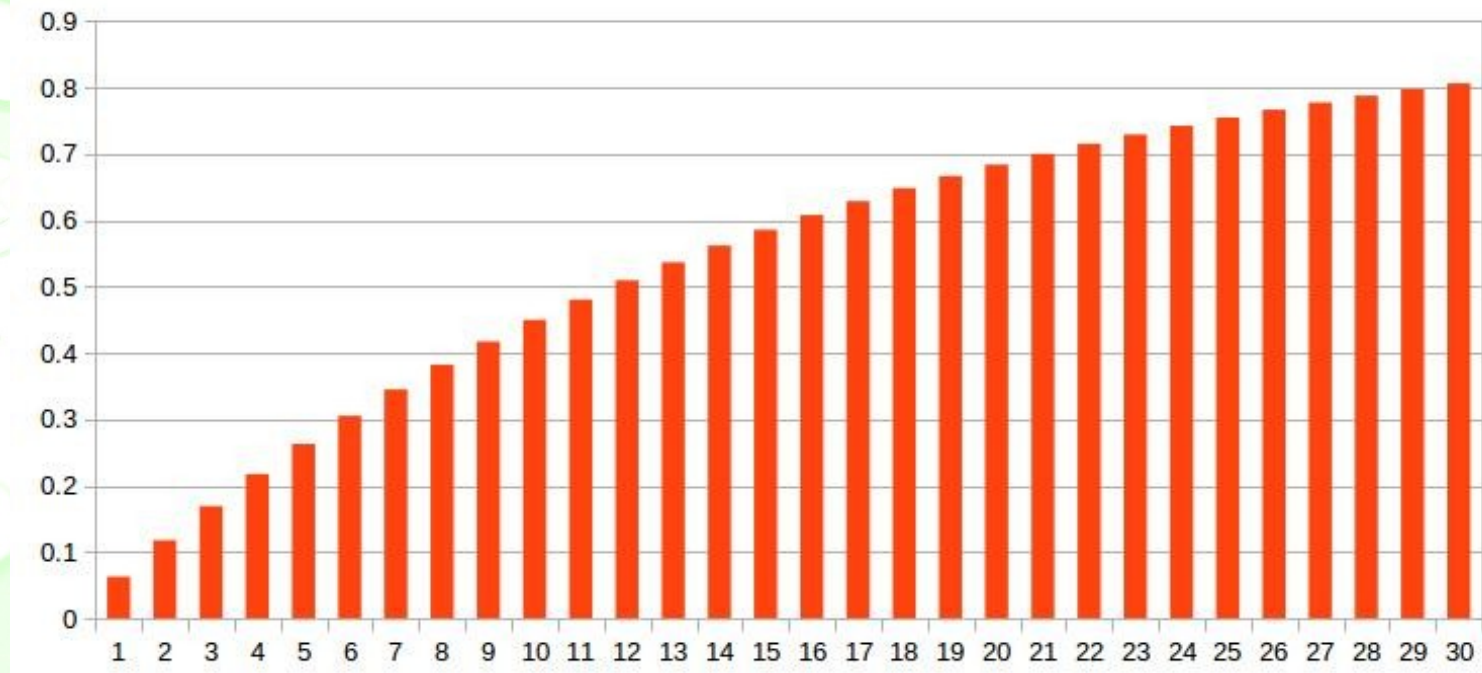
# Dynamic Adjustment of Difficulty

- For Bitcoin  $N = 2016$ .
- Note: expected time is 10 minutes, but individual blocks can be found within any time between 1s and a couple of hours.
- Graph of geometric distribution for  $p=1/16$ :



# Dynamic Adjustment of Difficulty

- Other view: cumulative probability of finding a solution (this is for  $p=1/16$ ):



# But Why Do All This Work?

- Good question. Doing all these hashes to find blocks is burning a lot of energy and heating up the planet!
- It's a probabilistic solution to the Byzantine Generals Problem: provided over half the computing power on the blockchain network is "good", the "evil" power probably can't subvert the system (for long).
- In other words, it's a way (but not the only way) for a distributed system to reach a consensus over time.
- Other, less power-intensive solutions are being investigated (proof-of-stake, proof-of-elapsed-time, practical Byzantine fault toleration).



# Byzantine Generals Problem

Here's Nakamoto's recasting of the original BGP:

- A number of Byzantine Generals each have a computer and want to attack the King's wi-fi by brute forcing the password, which they've learned is a certain number of characters in length. Once they stimulate the network to generate a packet, they must crack the password within a limited time to break in and erase the logs, otherwise they will be discovered and get in trouble (executed for treason!).
- They only have enough CPU power to crack it fast enough if a majority of them attack at the same time.

# Byzantine Generals Problem

- They don't particularly care when the attack will be, just that they all agree, and no minority accidentally attacks early.
- It has been decided that anyone who feels like it will announce a time, and whatever time is heard first will be the official attack time.
- The problem is that the general's network is not instantaneous, and if two generals announce different attack times at close to the same time, some may hear one first and others hear the other first.
- If that happens they are in trouble...



# Byzantine Generals Problem

How do they coordinate their attack time?

Let's say  
tomorrow at 2?



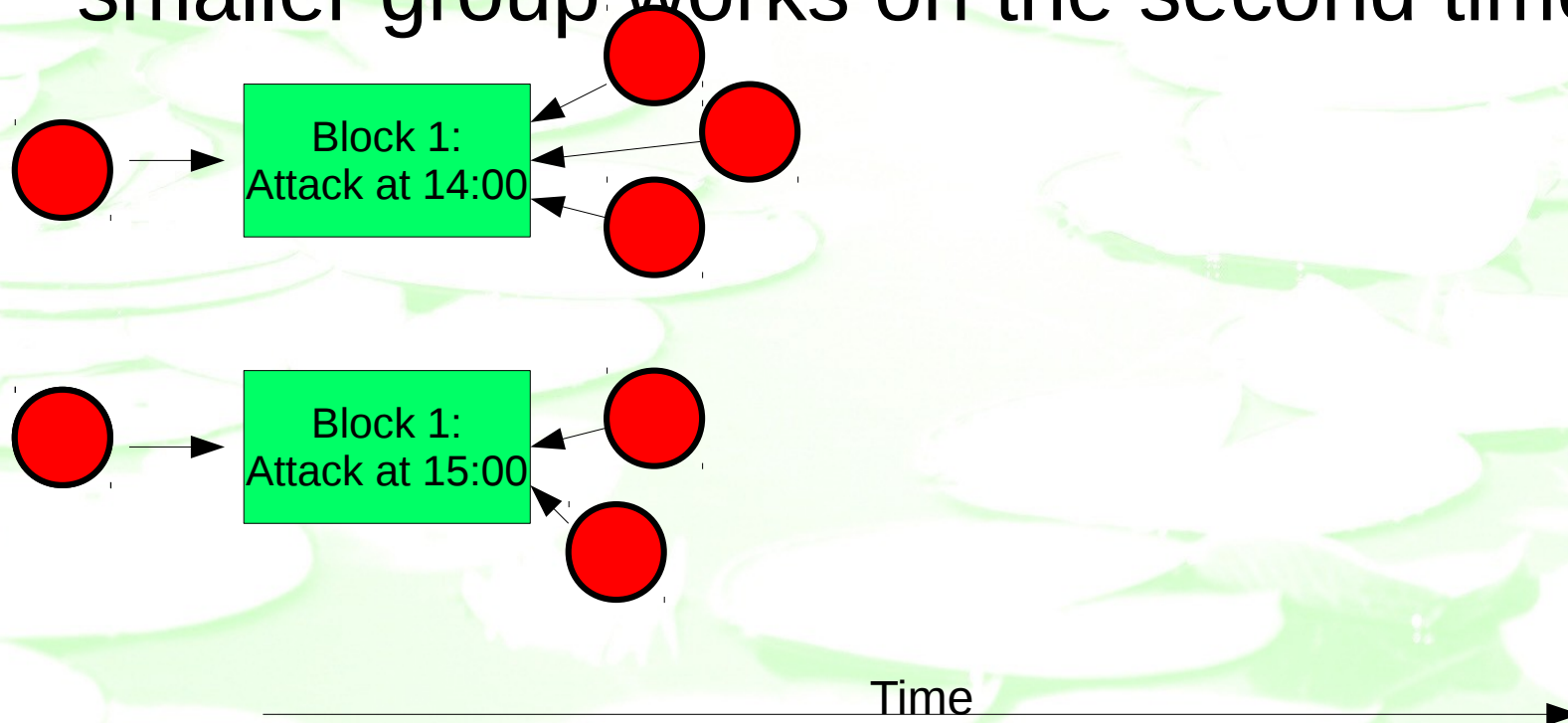


# Consensus

- The generals need to reach a consensus:
- They use a proof-of-work chain to solve the problem. Once each general receives whatever attack time he hears first, he sets his computer to solve the proof-of-work problem that includes the attack time in its hash.
- Each time a general creates a block he announces it on the network, and all the generals receiving it switch to generating the next block.
- If there are two competing blocks, some of the generals will work on one “fork” of the blockchain and some will work on the other.
- However, eventually one fork will be longer, and everyone will switch to working on that one.

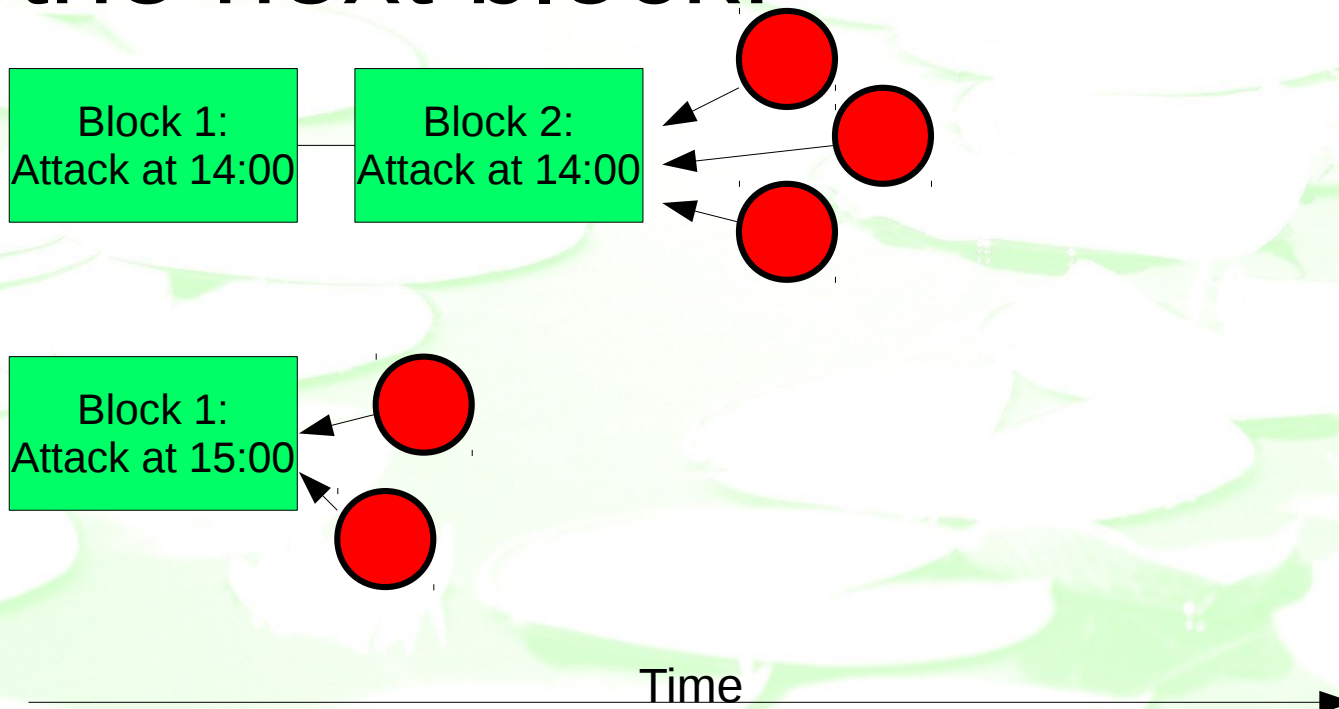
# Consensus

- Two generals announce a proposed attack schedule at about the same time. One group of generals start hashing the first time, a second smaller group works on the second time.



# Consensus

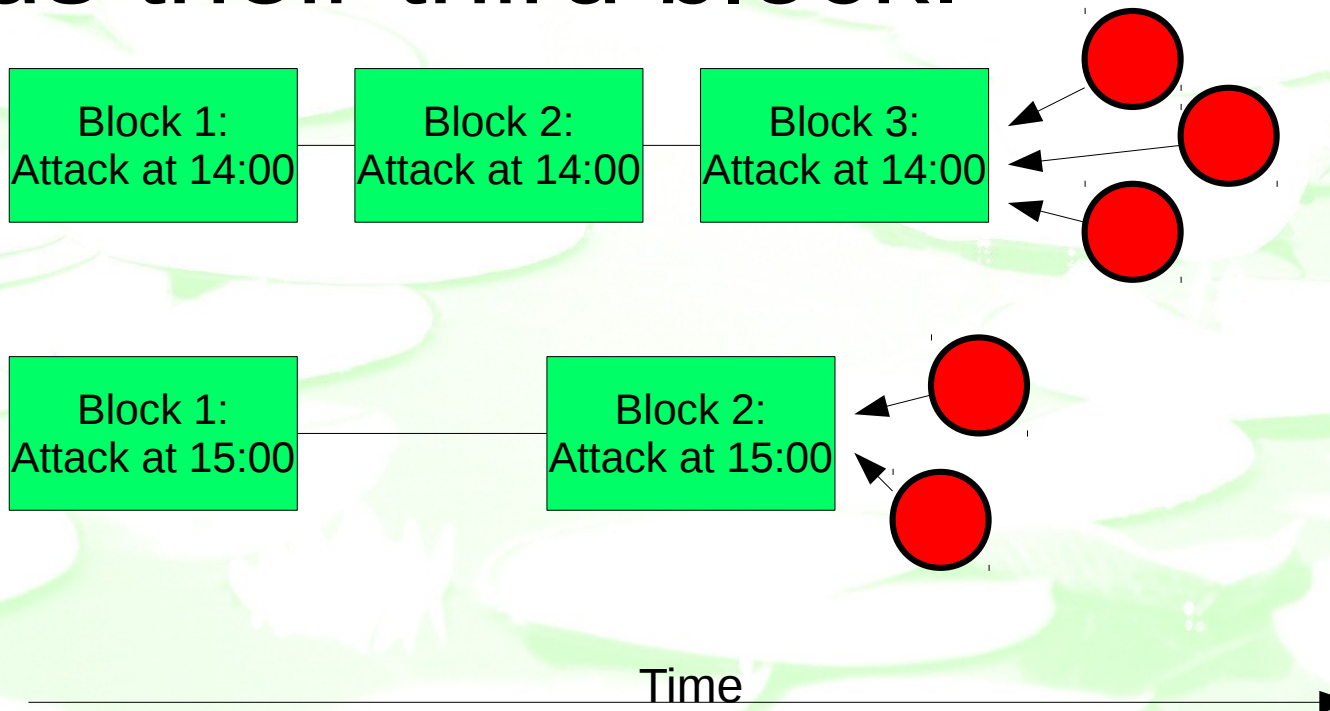
- Chances are the larger group will find a block first. They start working on the next block:





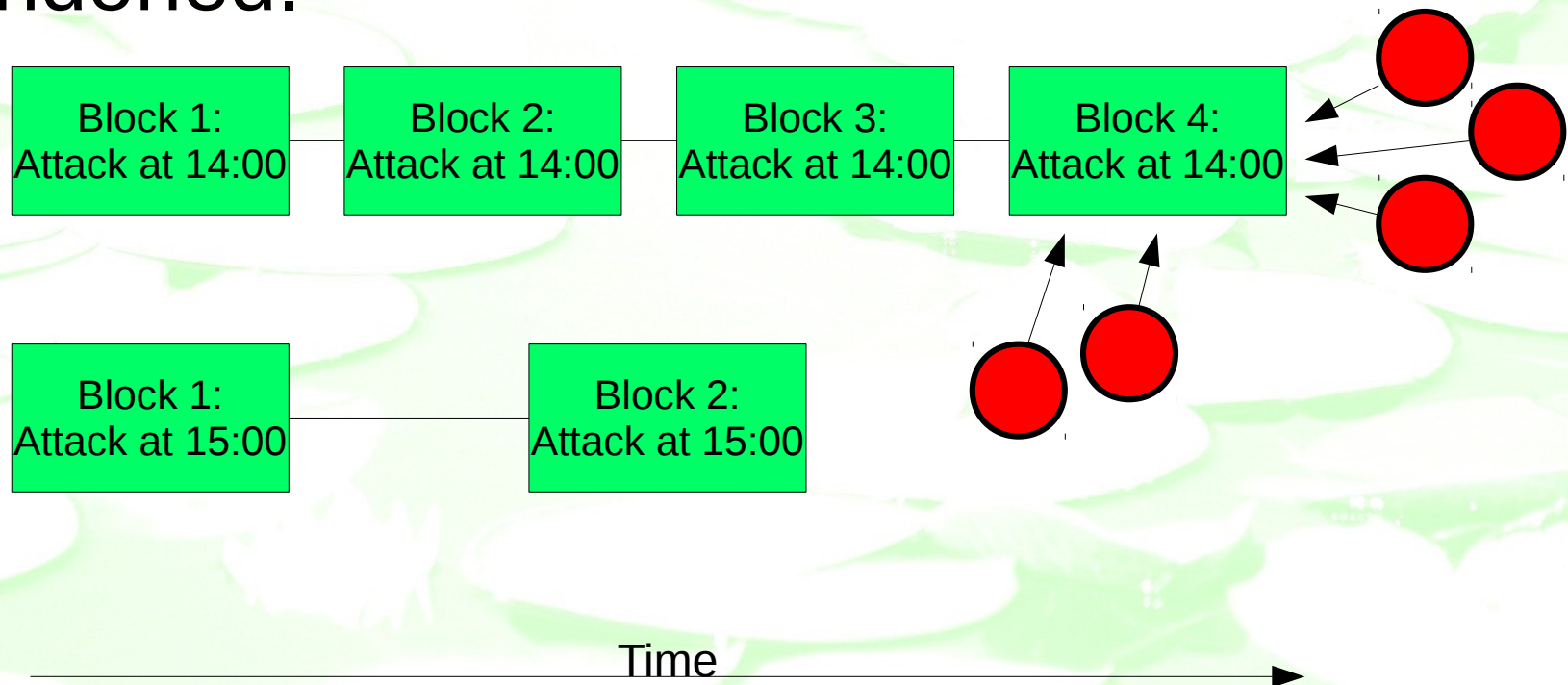
# Consensus

- The second group eventually finds a block, but by now the first group finds their third block.



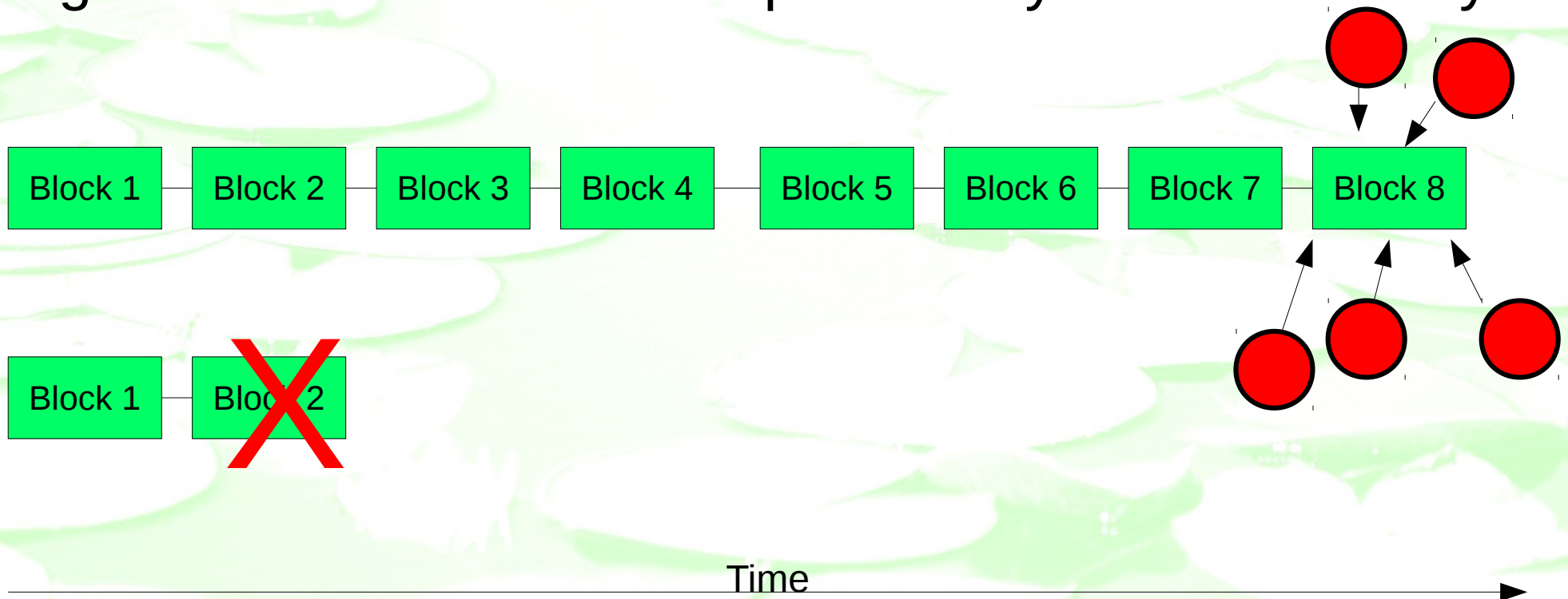
# Consensus

- And a fourth block. Obviously the first chain is extending more quickly. The remaining generals switch to working on it, and the second chain is abandoned.



# Consensus

- Eventually enough generals have expended enough computing power on a chain that determines a specific attack time, so they know they have the resources and agreed time to execute the plan safely and successfully.





# Identity

- We need to be able to identify agents or participants within the blockchain infrastructure:
- Who paid what to whom
- Which item was manufactured by who, and sold to whom
- Who repaired what, and when, with what components
- Who invented/wrote/coded what and when
- Who worked where and for how long

# Asymmetric Key Cryptography

- We use asymmetric key cryptography (also known as public key cryptography)
- Your public key is your public identity, or your “face” on the network
- No one can pretend to be you, because only you have the private key associated with the public key, or your “soul”...
- Keeping your private key secret is vitally important...

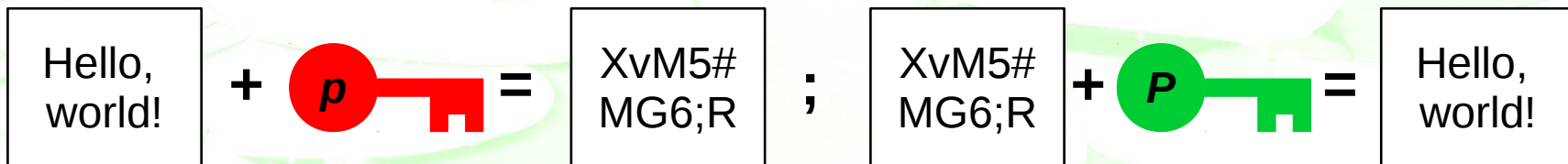
# Asymmetric Key Cryptography

- Simple summary:

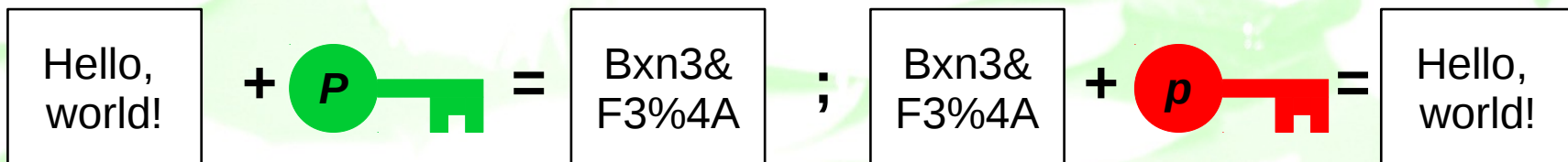
- You create a private key  $p$  and use it to generate a public key  $P$



- Anything encrypted with  $p$  can only be decrypted with  $P$



- Anything encrypted with  $P$  can only be decrypted with  $p$





# Asymmetric Key Cryptography

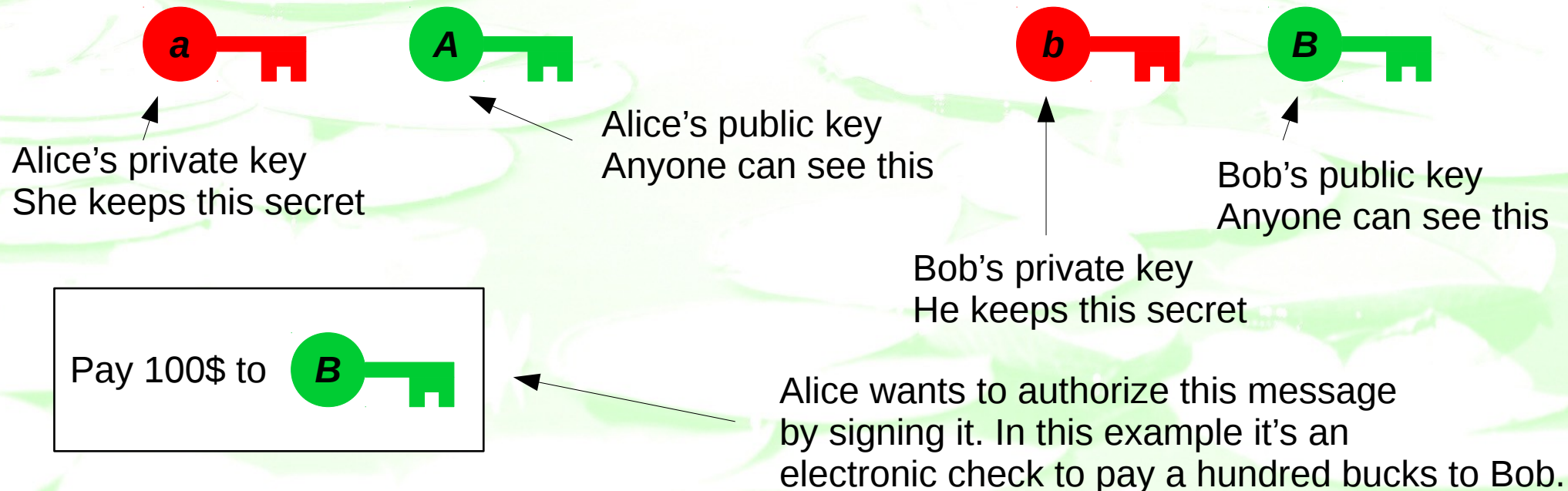
- Some maths (feel free to sleep for the next few minutes)
- Asymmetric key systems rely on one-way functions: it's easy to “do” the calculation, but hard to “undo” it.
- For example multiplying/factorisation (RSA):
  - What are the factors of 1643? (almost impossible to do in your head)
  - What is  $31 * 53$ ? (can be done in your head)
- Other one-way functions include:
  - Discrete exponential and logarithm:  $2^x/p$  (ElGamal)
  - Modular squaring:  $x^2 \bmod N$  where  $N = p1 * p2$  (Rabin)
  - Elliptic curves:  $y^2 = x^3 + ax + b$  over finite field,  $R = kP$  (ECDSA)

# Asymmetric Key Cryptography

- Although it's interesting (to mathematicians), you don't need to understand the maths. All these asymmetric key schemes have been implemented in OpenSSL, Python, Ruby, etc., and you can just use library functions.
- Don't implement your own versions for production. You'll make mistakes.
- What you do need to understand is the practicalities behind the systems
- As keys are large numbers stored on computers, you'll also need to understand hex notation of numerals
- Then you pick a random private key, use the functions to create the public key, and start encrypting...

# Signing

- Question: so how can you use your key to sign something?
- Answer: by using your private key to encrypt a hash of the message you want to sign, and attaching it to the message (usually along with your public key).
- For example, imagine Alice and Bob have their own private and public keys. Here they are:





# Signing

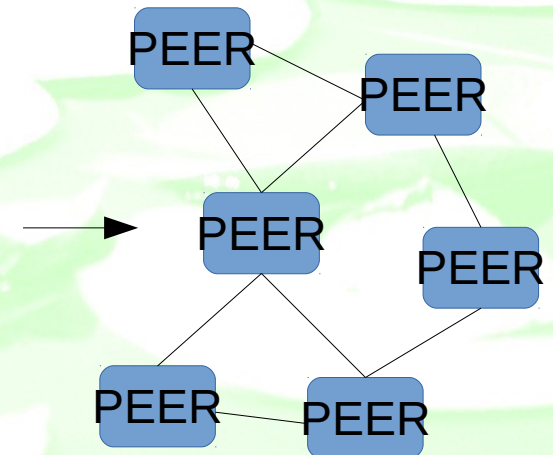
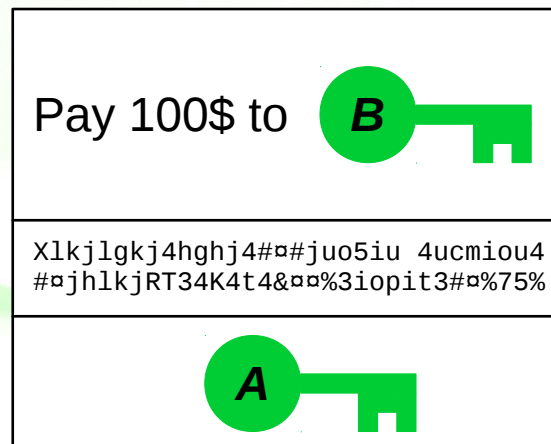
- Alice takes a hash of the message

$$\text{SHA256}(\text{Pay 100\$ to } \text{B} \text{ key}) = \begin{array}{|c|} \hline 61495e7d19001c89fd3a47bfd1f0f6a7 \\ 0765cd1f89d91e8c1b7014d51fa9a544 \\ \hline \end{array}$$

- She encrypts the hash using her private key

$$\begin{array}{|c|} \hline 61495e7d19001c89fd3a47bfd1f0f6a7 \\ 0765cd1f89d91e8c1b7014d51fa9a544 \\ \hline \end{array} + \text{a key} = \begin{array}{|c|} \hline \text{Xlkjlgkj4hghj4\#\#\juo5iu 4ucmiou4} \\ \text{\#ajhlkjRT34K4t4\&\#\%3iopit3\#\%75\%} \\ \hline \end{array}$$

- She attaches the encrypted hash and her public key to the message and transmits it to the blockchain network



# Signing

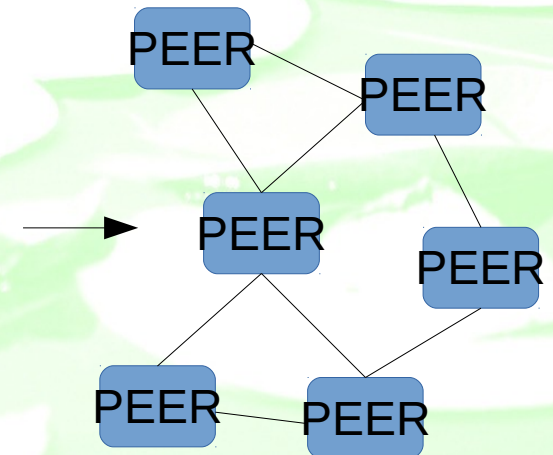
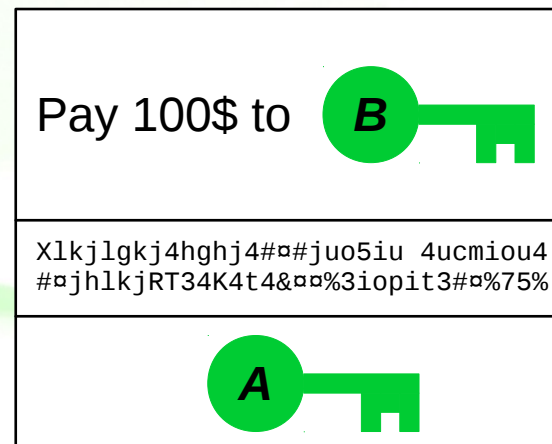
- Alice takes a hash of the message

$$\text{SHA256}(\text{Pay 100\$ to } \text{B} \text{ key}) = \begin{array}{|c|} \hline 61495e7d19001c89fd3a47bfd1f0f6a7 \\ 0765cd1f89d91e8c1b7014d51fa9a544 \\ \hline \end{array}$$

- She encrypts the hash using her private key

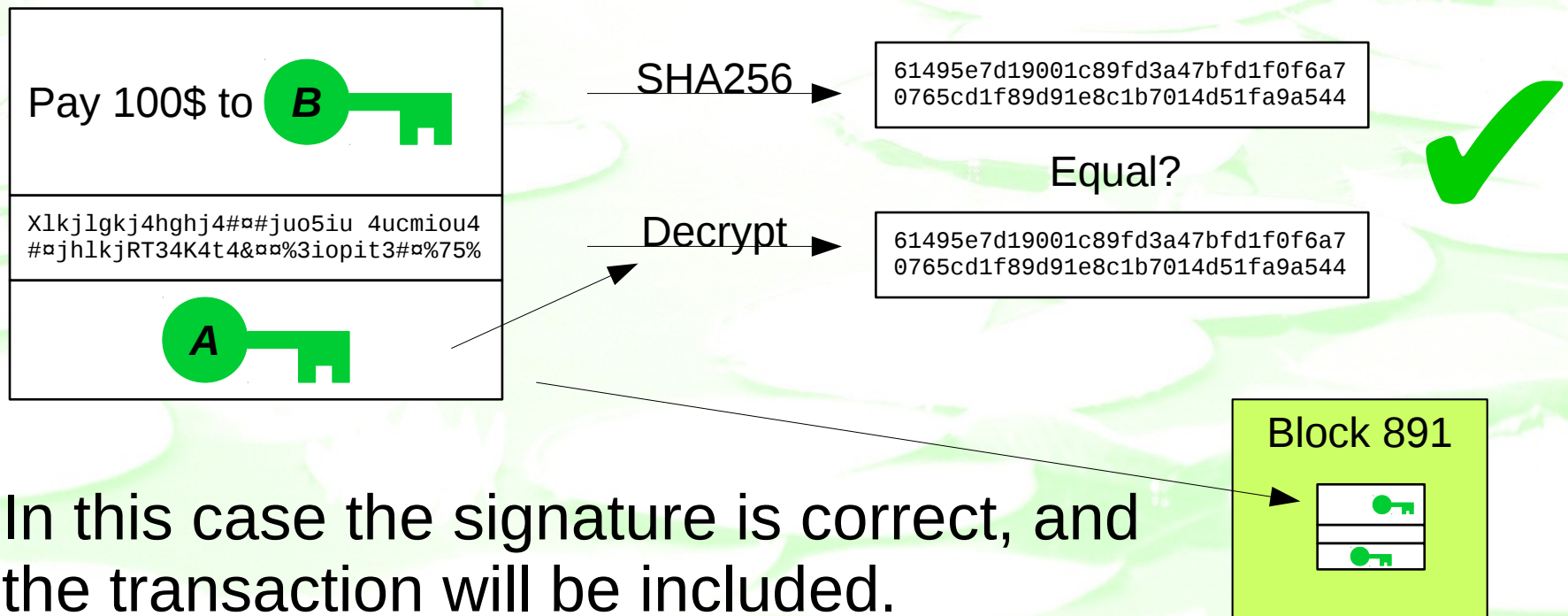
$$\begin{array}{|c|} \hline 61495e7d19001c89fd3a47bfd1f0f6a7 \\ 0765cd1f89d91e8c1b7014d51fa9a544 \\ \hline \end{array} + \text{a key} = \begin{array}{|c|} \hline \text{Xlkjlgkj4hghj4\#\#\juo5iu 4ucmiou4} \\ \text{\#ajhlkjRT34K4t4\&\#\%3iopit3\#\%75\%} \\ \hline \end{array}$$

- She attaches the encrypted hash and her public key to the message and transmits it to the blockchain network



# Signing

- Will peers include the transaction in their next block for adding to the blockchain? Yes, on two conditions:
  - Examining the blockchain shows that Alice has 100\$ to transfer
  - The signature on the transaction checks out

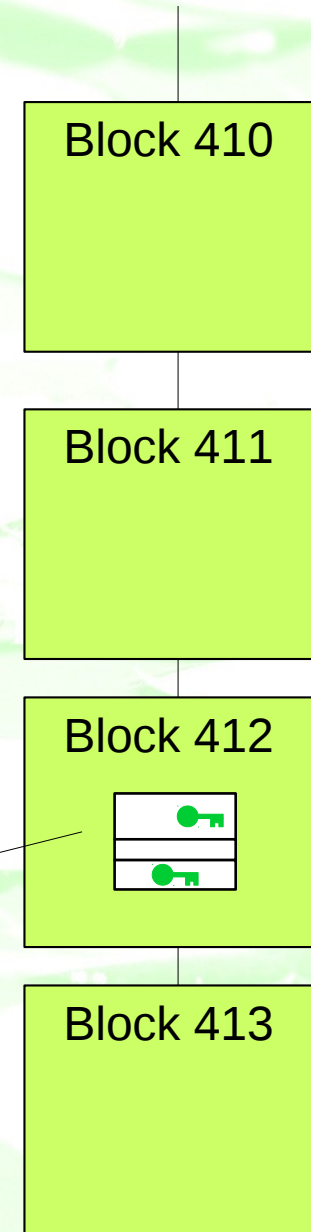
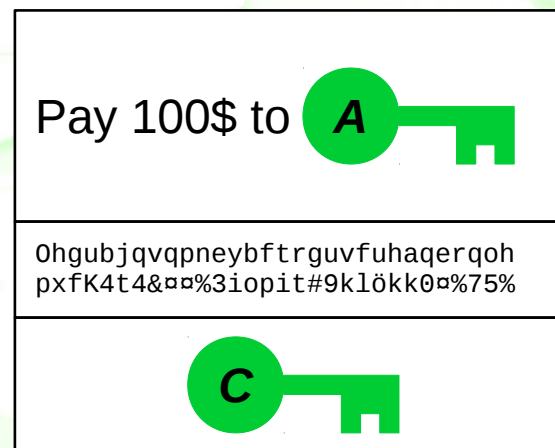


- In this case the signature is correct, and the transaction will be included.



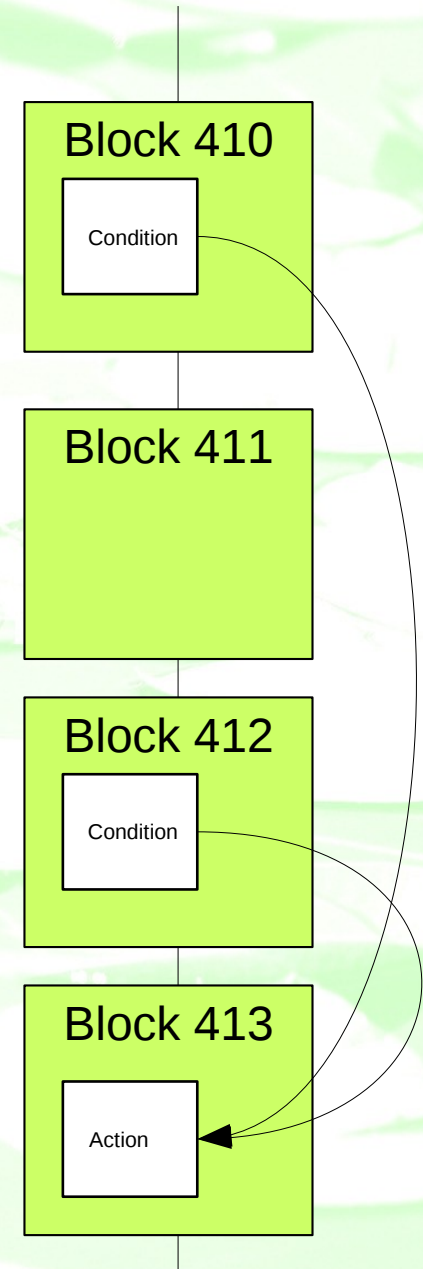
# Signing

- Hang on, how do we know Alice has 100\$
  - Examining the blockchain shows that Alice has 100\$ to transfer
  - She got it from Carlos in block 412



# Scripts/Smart Contracts

- Transactions are generally presented as code:
  - Bitcoin uses a simple Forth-like language and stack to code coin transactions
  - Ethereum uses Solidity, a Javascript-like language for coding contracts
  - Hyperledger Fabric uses Golang to produce something called Chaincode
  - Hyperledger Sawtooth Lake uses JSON data structures and SQL-like commands to trigger transactions based on code folded in to the system



# Smart Contracts

- Proposed by Nick Szabo in 1994
- An abstract concept relating to the automated execution of an already agreed contract
- Can be realised in blockchains through a contract scripting language

## Ricardian Contracts

- Based on the work of Ian Grigg in 1996
- A software design pattern to capture the intent of the agreement of the parties, before its execution
- Produces contracts that are both human and machine readable, and can be implemented with smart contracts



# Bitcoin Transactions

Sample Bitcoin transaction script (pay to public key):

Input:

Previous tx: f5d8ee39a430901c91a5917b9f2dc19d6d1a0e9cea205b009ca73dd04470b9a6

Index: 0

scriptSig: 304502206e21798a42fae0e854281abd38bacd1aeed3ee3738d9e1446618c4571d10  
90db022100e2ac980643b0b82c0e88ffdfec6b64e3e6ba35e7ba5fdd7d5d6cc8d25c6b241501

Output:

Value: 5000000000

scriptPubKey: OP\_DUP OP\_HASH160 404371705fa9bd789a2fcd52d2c580b65d35549d OP\_EQUALVERIFY OP\_CHECKSIG

- The transaction requires an input, which is where the “money” comes from. The transaction is digitally signed. For the sender to claim this input (and send it as an output) they have to provide the full public key and sign the transaction with the associated private key.
- The amount from the input to send out to a new address is specified in the output. The destination address is not given: instead a RIPEMD-160 hash of the receiving address is provided. This is why the public key needs to be provided in the input scriptSig
- You can see that a Bitcoin transaction is a script: the last two lines are commands to duplicate onto the stack the RIPEMD-160 hash of the receiving address and the expected hash, compare them and then check the signature. If the result from this script is “True” the value can be redeemed.
- Bitcoin contains a stack-based language with commands to push items onto the stack, perform logical operations on them, do logical comparisons, perform arithmetic, and perform cryptographic actions, amongst others.

# Bitcoin Script Commands

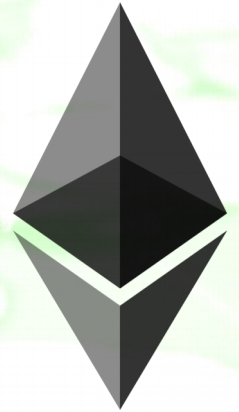
Word	Description
OP_TOALTSTACK	Puts the input onto the top of the alt stack. Removes it from the main stack.
OP_FROMALTSTACK	Puts the input onto the top of the main stack. Removes it from the alt stack.
OP_IFDUP	If the top stack value is not 0, duplicate it.
OP_DEPTH	Puts the number of stack items onto the stack.
OP_DROP	Removes the top stack item.
OP_DUP	Duplicates the top stack item.
OP_NIP	Removes the second-to-top stack item.
OP_OVER	Copies the second-to-top stack item to the top.
OP_PICK	The item n back in the stack is copied to the top.
OP_ROLL	The item n back in the stack is moved to the top.
OP_ROT	The top three items on the stack are rotated to the left.
OP_SWAP	The top two items on the stack are swapped.
OP_TUCK	The item at the top of the stack is copied and inserted before the second-to-top item.
OP_2DROP	Removes the top two stack items.
OP_2DUP	Duplicates the top two stack items.

# Bitcoin Script Commands

Word	Description
OP_1ADD	1 is added to the input.
OP_1SUB	1 is subtracted from the input.
OP_NEGATE	The sign of the input is flipped.
OP_ABS	The input is made positive.
OP_NOT	If the input is 0 or 1, it is flipped. Otherwise the output will be 0.
OP_0NOTEQUAL	Returns 0 if the input is 0. 1 otherwise.
OP_ADD	a is added to b.
OP_SUB	b is subtracted from a.
OP_BOOLAND	If both a and b are not 0, the output is 1. Otherwise 0.
OP_BOOLOR	If a or b is not 0, the output is 1. Otherwise 0.
OP_NUMEQUAL	Returns 1 if the numbers are equal, 0 otherwise.
OP_NUMEQUALVERIFY	Same as OP_NUMEQUAL, but runs OP_VERIFY afterward.
OP_NUMNOTEQUAL	Returns 1 if the numbers are not equal, 0 otherwise.
OP_LESSTHAN	Returns 1 if a is less than b, 0 otherwise.
OP_GREATERTHAN	Returns 1 if a is greater than b, 0 otherwise.
OP_EQUAL	Returns 1 if a is equal to b, 0 otherwise.
OP_LESSTHANOREQUAL	Returns 1 if a is less than or equal to b, 0 otherwise.

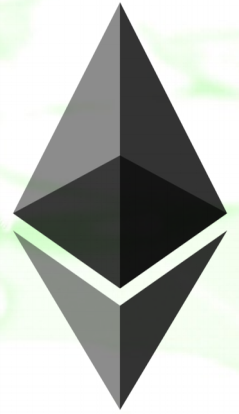


# Ethereum



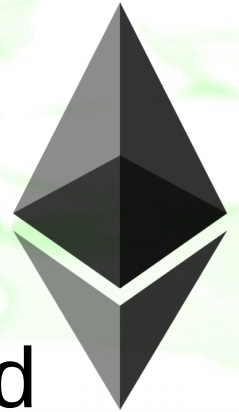
- Proposed by Vitalik Buterin in 2013 (when he was 19 years old)
- Development funded by crowdfunding in 2014
- Launched in 2015
- A public blockchain with an associated cryptocurrency called Ether (ETH), and a more complicated scripting language called Solidity.

# Ethereum



- To get the peers to run your program on the blockchain (the Ethereum Virtual Machine, or EVM) you have to pay ETH (for some reason when used like this it's called Gas)
- You set a Gas Limit to ensure your program doesn't use up all your ETH.

# Ethereum



- Solidity looks a lot like Javascript.
- There are all sorts of ways to compile and deploy your “contracts” on the Ethereum blockchain:
  - Ethereum GUI Wallet/Mist browser allows you to build and deploy contracts
  - Ethereum command line interface (called Geth) includes a Solidity compiler
  - Try it out in a web-browser at <https://ethereum.github.io/browser-solidity/>
  - More tools appearing all the time (e.g. Dapple, Node.JS developer’s harness)



# Hyperledger

- This is the “blockchain” project that the big companies are supporting.
  - Coordinated by the Linux Foundation
  - Two main projects are:
    - Fabric (IBM blockchain for smart contracts)
    - Sawtooth Lake (Intel blockchain using modular components)
- Yes, there are two different blockchain systems...



# HYPERLEDGER

# Fabric

- Tutorial on how to set up a development environment at <http://www.chainfrog.com/ibm-hyperledger-fabric/>
- It uses:
  - Docker, a software containerization platform, to ensure that if you run Fabric on a native platform, all the required packages and libraries are standardized. A Node.JS npm provides access to a Fabric SDK.
  - Or, a Virtual Machine running on VirtualBox, and Vagrant to ensure the VM is Ubuntu configured to ensure a standard development environment.

# Sawtooth Lake

- Also uses a VM and Vagrant to provide a standardized environment for execution and development
  - Core “sawtooth lake” code provides peers/validators
  - “MarketPlace” data structures and objects provide the means to create and execute transactions on the blockchain
- Tutorial available at  
<https://intelledger.github.io/tutorial.html>