2023/2024 S2

Bitcoin I

Dr. Hui Gong (h.gong1@westminster.ac.uk)

UNIVERSITY OF WESTMINSTER

Cryptographic Hash Functions

Hash Function

The first cryptographic primitive that we need to understand is a ***cryptographic hash function***. A hash function is a mathematical function with the following three properties:

- Its input can be any string of any size.
- It produces a **fixed-sized output**. For the purpose of making the discussion here concrete, we will assume a 256-bit output size. However, our discussion holds true for any output size, as long as it is sufficiently large.
- It is efficiently computable. Intuitively this means that for a given input string, you can figure out what the output of the hash function is in a reasonable amount of time. More technically, computing the hash of an *n*-bit string should have a running time that is *O(n)*.

These properties define a general hash function, one that could be used to build a data structure, such as a hash table. We're going to focus exclusively on *cryptographic* hash functions. For a hash function to be cryptographically secure, we require that it has the following three additional properties:

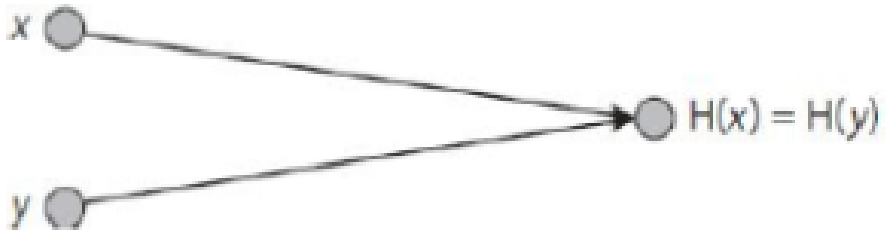
(1) collision resistance,

(2) hiding, and

(3) puzzle friendliness.

The puzzle friendliness property, in particular, is not a general requirement for cryptographic hash functions, but one that will be useful for **cryptocurrencies** specifically.

Collision Resistance



The first property that we need from a cryptographic hash function is that it is collision resistant. A collision occurs when two distinct inputs produce the same output. A hash function H(·) is collision resistant if nobody can find a collision. Formally:

Collision resistance. A hash function H is said to be collision resistant if it is infeasible to find two values, x and y, such that $x \neq y$, yet $H(x) = H(y)$.

Notice that we said "**nobody can find**" a collision, **but we did not say that no collisions exist**. Actually, collisions exist for any hash function, and we can prove this by a simple counting argument.

Now, to make things even worse, we said that it has to be impossible to find a collision. Yet there are methods that are guaranteed to find a collision. Consider the following simple method for finding a collision for a hash function with a 256-bit output size: pick $2^{256} + 1$ distinct values, compute the hashes of each of them, and check whether any two outputs are equal. Since we picked more inputs than possible outputs, some pair of them must collide when you apply the hash function. The method above is guaranteed to find a collision. But if we pick random inputs and compute the hash values, we'll find a collision with high probability long before examining $2^{256} + 1$ inputs.

Cryptographic Hash Functions

Hiding

The second property that we want from our hash functions is that it is hiding. The hiding property asserts that if we're given the output of the hash function $y = H(x)$, there's no feasible way to figure out what the input, x, was.

Hiding. A hash function H is said to be hiding if when a secret value r is chosen from a probability distribution that has *high min-entropy*, then, given $H(r||x)$, it is infeasible to find x.

In information theory, *min-entropy* is a measure of how predictable an outcome is, and *high min-entropy* captures the intuitive idea that the distribution (i.e., of a random variable) is very spread out.

So, for a concrete example, if r is chosen uniformly from among all strings that are 256 bits long, then any particular string is chosen with probability 1/ 2^{256}, which is an infinitesimally small value.

In particular, what we want to do is called a *commitment*.

Commitment scheme. A commitment scheme consists of two algorithms:

- $com := commit(msg, nonce)$ The commit function takes a message and secret random value, called a nonce, as input and returns a commitment.
- $verify(com, msg, nonce)$ The verify function takes a commitment, nonce, and message as input. It returns true if $com == commit(msg, nonce)$ and false otherwise.

We require that the following two security properties hold:

- Hiding: Given com, it is infeasible to find msg.
- Binding: It is infeasible to find two pairs $(msg, nonce)$ and $(msg', nonce')$ such that $msg \neq msg'$ and $commit(msg, nonce) == commit(msg', nonce')$.

To use a commitment scheme, we first need to generate a **random nonce**. We then apply the *commit* function to this nonce together with *msg*, the value being committed to, and we publish the commitment *com*.

Hiding

At a later point, if we want to reveal the value that we committed to earlier, we publish the **random nonce** that we used to create this commitment, and the message, *msg*. Now anybody can verify that *msg* was indeed the message committed to earlier.

Every time you commit to a value, it is important that you choose a new random value *nonce*. In cryptography, the term nonce is used to refer to a value that can only be used once.

The two security properties dictate that the algorithms actually behave like sealing and opening an envelope. First, given *com*, the commitment, someone looking at the envelope can't figure out what the message is. The second property is that it's binding. This ensures that when you commit to what's in the envelope, you can't change your mind later. That is, it's infeasible to find two different messages, such that you can commit to one message and then later claim that you committed to another.

So how do we know that these two properties hold? Before we can answer this, we need to discuss how we're going to actually implement a commitment scheme. We can do so using a cryptographic hash function. Consider the following commitment scheme:

$$commit(msg, nonce) := H(nonce \,||\, msg),$$

where *nonce* is a random 256-bit value

To commit to a message, we generate a random 256-bit nonce. Then we concatenate the nonce and the message and return the hash of this concatenated value as the commitment. To verify, someone will compute this same hash of the nonce they were given concatenated with the message. And they will check whether the result is equal to the commitment that they saw.

Therefore, if H is a hash function that is both collision resistant and hiding, this commitment scheme will work, in the sense that it will have the necessary security properties.

Puzzle Friendliness

The third security property we're going to need from hash functions is that they are puzzle friendly.

Puzzle friendliness. A hash function H is said to be puzzle friendly if for every possible n-bit output value y, if k is chosen from a distribution with high min-entropy, then it is infeasible to find x such that H(k||x) = y in time significantly less than 2^n.

In this application, we're going to build a search puzzle, a mathematical problem that requires searching a very large space to find the solution.

Search puzzle. A search puzzle consists of

- a hash function, H,
- a value, id (which we call the puzzle-ID), chosen from a high min-entropy distribution, and
- a target set Y.

A solution to this puzzle is a value, x, such that

$$H(id||x) \in Y$$

The intuition is this: if H has an n-bit output, then it can take any of 2^n values. Solving the puzzle requires finding an input such that the output falls within the set Y, which is typically much smaller than the set of all outputs. The size of Y determines how hard the puzzle is. If Y has only one element, then the puzzle is trivial, whereas if Y is the set of all n-bit strings, then the puzzle is maximally hard.

If a hash function is puzzle friendly, then there's no solving strategy for this puzzle that is much better than just trying random values of x. And so, if we want to pose a puzzle that's difficult to solve, we can do it this way as long as we can generate puzzle-IDs in a suitably random way.

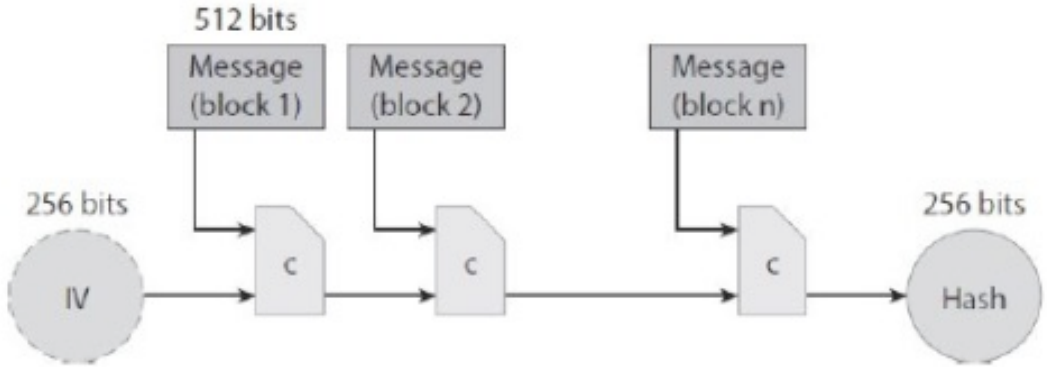
Cryptographic Hash Functions

SHA-256

Many hash functions exist, but this is the one Bitcoin uses primarily, and it's a pretty good one to use. It's called SHA-256.

Recall that we require that our hash functions work on inputs of arbitrary length. Luckily, as long as we can build a hash function that works on fixedlength inputs, there's a generic method to convert it into a hash function that works on arbitrary-length inputs. It's called the *Merkle-Damgård transform*. SHA- 256 is one of a number of commonly used hash functions that make use of this method. In common terminology, the underlying fixed-length collision-resistant hash function is called the *compression function*.

SHA-256 uses a compression function that takes 768-bit input and produces 256-bit outputs. The block size is 512 bits.



SHA-256 hash function (simplified). SHA-256 uses the Merkle-Damgård transform to turn a fixed-length collision-resistant compression function into a hash function that accepts arbitrary-length inputs. The input is padded, so that its length is a multiple of 512 bits.

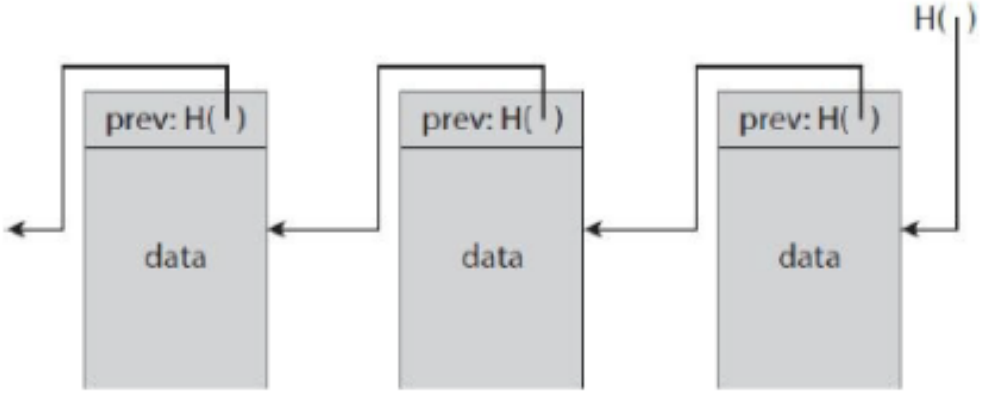
IV stands for initialization vector.

Hash Pointers and Data Structures

Block Chain

A hash pointer is a data structure that turns out to be useful in many of the systems that we consider. A hash pointer is simply a pointer to where some information is stored together with a cryptographic hash of the information.

Figure shows a linked list using hash pointers. We call this data structure a block chain. In a regular linked list where you have a series of blocks, each block has data as well as a pointer to the previous block in the list. But in a block chain, the previous-block pointer will be replaced with a hash pointer. So each block not only tells us where the value of the previous block was, but it also contains a digest of that value, which allows us to verify that the value hasn't been changed. We store the head of the list, which is just a regular hash-pointer that points to the most recent data block.



So we can build a block chain like this containing as many blocks as we want, going back to some special block at the beginning of the list, which we will call the *genesis* block.

You may have noticed that the block chain construction is similar to the Merkle-Damgård construction.

Digital Signatures

Digital Signatures Scheme

We desire two properties from digital signatures that correspond well to the handwritten signature analogy. First, only you can make your signature, but anyone who sees it can verify that it's valid. Second, we want the signature to be tied to a particular document, so that the signature cannot be used to indicate your agreement or endorsement of a different document.

Digital signature scheme. A digital signature scheme consists of the following three algorithms:

• $(sk, pk) := generateKeys(keysize)$ The generateKeys method takes a key size and generates a key pair. The secret key sk is kept privately and used to sign messages. pk is the public verification key that you give to everybody. Anyone with this key can verify your signature.

• $sig := sign(sk, message)$ The sign method takes a message and a secret key, sk, as input and outputs a signature for message under sk.

• $isValid := verify(pk, message, sig)$ The verify method takes a message, a signature, and a public key as input. It returns a boolean value, $isValid$, that will be true if sig is a valid signature for message under public key pk, and false otherwise.

We require that the following two properties hold:

• Valid signatures must verify: $verify(pk, message, sign(sk, message)) == true$.

• Signatures are *existentially unforgeable*.

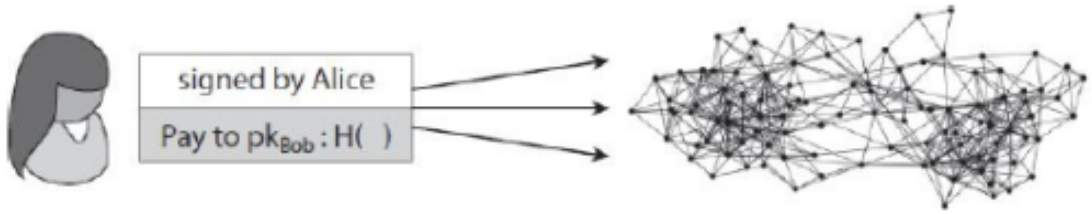
Distributed Consensus

Distributed Consensus Protocol

Distributed consensus protocol. There are n nodes that each have an input value. Some of these nodes are faulty or malicious. A distributed consensus protocol has the following two properties:

- It must terminate with all honest nodes in agreement on the value.
- The value must have been generated by an honest node.

What does this mean in the context of Bitcoin? To understand how distributed consensus works in Bitcoin, remember that Bitcoin is a peer-to-peer system. When Alice wants to pay Bob, what she actually does is **broadcast a transaction to all Bitcoin nodes that make up the peer-to-peer network.**



Incidentally, you may have noticed that Alice broadcasts the transaction to all Bitcoin peer-to-peer nodes, but Bob's computer is nowhere in this picture. It's of course possible that Bob is running one of the nodes in the peer-to-peer network. In fact, if he wants to be notified that this transaction did in fact happen and that he has been paid, running a node might be a good idea. Nevertheless, there is no requirement that Bob be listening on the network; running a node is not necessary for Bob to receive the funds. The bitcoins will be his regardless of whether he's operating a node on the network.

Incentives and Proof of Work (POW)

Incentives and Proof of Work (POW)

Incentive Mechanisms

Block Reward

According to the rules of Bitcoin, the node that creates a block gets to include a special transaction in that block.

As of 2015, the value of the block reward is fixed at 25 bitcoins. But it actually halves with every 210,000 blocks created. Based on the rate of block creation, the rate halves roughly every four years. We're now in the second period. For the first four years of Bitcoin's existence, the block reward was 50 bitcoins; now it's 25. And it's going to keep halving. This has some interesting consequences, which we address below.

Note that this is the only way in which new bitcoins can be created. There is no other coin-generation mechanism, which is why 21 million is a final and total number (as the rules stand now, at least) for how many bitcoins there can ever be. This block reward will run out in 2140, as things stand now.

Transaction Fees

The creator of any transaction can choose to make the total value of the transaction outputs less than the total value of its inputs. Whoever creates the block that first puts that transaction into the block chain gets to collect the difference, which acts a transaction fee. So if you're a node that is creating a block containing, say, 200 transactions, then the sum of those 200 transaction fees is paid to the address that you put into that block. The transaction fee is purely voluntary, but we expect, based on our understanding of the system, that as the block reward starts to run out, it will become more and more important, almost mandatory, for users to include transaction fees to maintain a reasonable quality of service.

Proof of Work (POW)

Problems

A few problems still remain with the consensus mechanism as described here.

1. The first major one is the leap of faith that we asked you to take that somehow, we can pick a random node.
2. Second, we've created a new problem by giving nodes these incentives for participation. The system can become unstable as the incentives cause a free-for-all, where everybody wants to run a Bitcoin node in the hope of capturing some of these rewards.
3. And a third one is an even trickier version of this problem: an adversary might create a large number of Sybil nodes to try and subvert the consensus process.

Solution

All these problems are related, and all have the same solution, which is called proof of work. The key idea behind ***proof of work*** is that we approximate the selection of a random node by instead selecting nodes in proportion to a resource that we hope that nobody can monopolize. If, for example, that resource is computing power, then it's a *proof-of-work system*.

Bitcoin achieves ***proof of work*** using hash puzzles. To create a block, the node that proposes that block is required to find a number (nonce), such that when you concatenate the nonce, the previous hash, and the list of transactions that make up the block and then take the hash of this whole string, then that hash output should be a number that falls in a target space that is quite small in relation to the much larger output space of that hash function.

Bitcoin Mining

Bitcoin Mining

The Task of Bitcoin Miners

1. ***Listen for transactions***. You listen for transactions on the network and validate them by checking that signatures are correct and that the outputs being spent haven't already been spent.

2. ***Maintain block chain and listen for new blocks***. You must maintain the block chain. You start by asking other nodes to give you all the historical blocks that are already part of the block chain before you joined the network. You then listen for new blocks that are being broadcast to the network. You must validate each block that you receive - by validating each transaction in the block and checking that the block contains a valid nonce.

3. ***Assemble a candidate block***. Once you have an up-to-date copy of the block chain, you can begin building your own blocks. To do this, you group transactions that you have heard about into a new block that extends the latest block you know about. You must make sure that each transaction included in your block is valid.

4. ***Find a nonce that makes your block valid***. This step requires the most work, and it poses all the real difficulties for miners.

5. ***Hope your block is accepted***. Even if you find a block, there's no guarantee that your block will become part of the consensus chain. There's a bit of luck here; you have to hope that other miners accept your block and start mining on top of it instead of some competitor's block.

6. ***Profit***. If all other miners do accept your block, then you profit! In 2015, the block reward is 25 bitcoins, which is currently worth about **$10,000**. (**NO. It currently worth about $875,000 if you hold it until now! It is equivalent to a 2-Bed flat in Central London.**) In addition, if any of the transactions in the block contained transaction fees, the miner collects those, too. So far transaction fees have been a modest source of additional income, only about 1 percent of block rewards.

Determining the Difficulty

Exactly how difficult is it to find a valid block? As of the end of 2015, the mining difficulty target (in hexadecimal) is:

0000000000000000a955000000000000000000000000000000000000000000

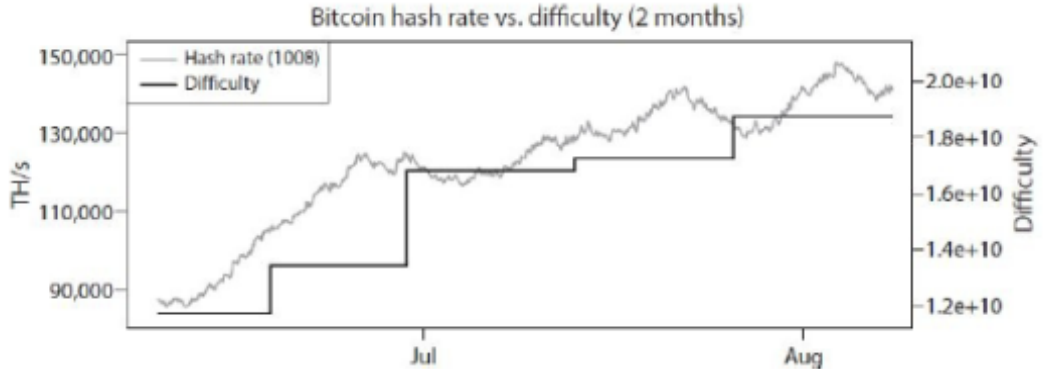
so the hash of any valid block has to be below this value. In other words, less than 1 in about 2^{68} nonces that you try will work, which is a really huge number. One approximation is that it's greater than the human population of Earth squared.

The mining difficulty changes every 2,016 blocks, which are found about once every 2 weeks. It is adjusted based on how efficient the miners were over the period of the previous 2,016 blocks according to this formula:

$$\textbf{next difficulty} = \frac{\text{previous difficulty} \cdot 2016 \cdot 10\ minutes}{\text{time to mine last 2016 blocks}}$$

Note that 2,016 · 10 minutes is exactly 2 weeks, so 2,016 blocks would take 2 weeks to mine if a block were created exactly every 10 minutes.

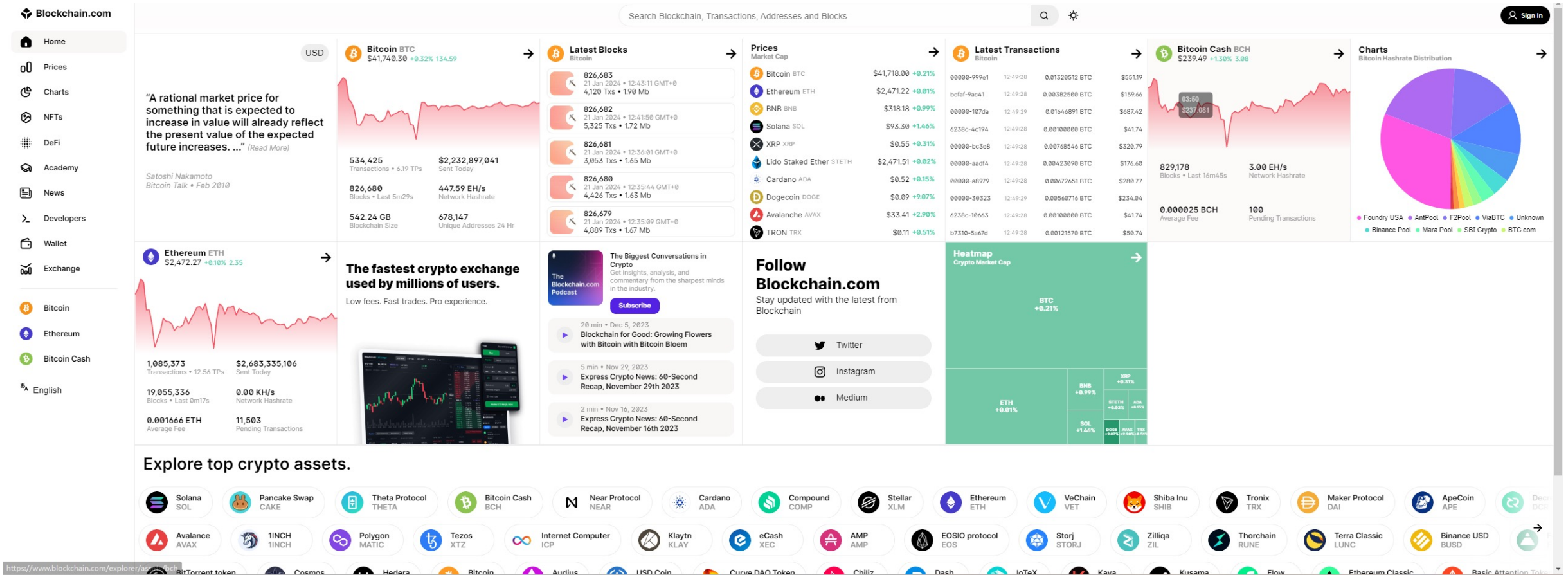
So the effect of this formula is to scale the difficulty to maintain the property that blocks should be found by the network on average about once every 10 minutes. There's nothing special about 2 weeks, but it's a good trade-off.



Network Difficulty: A relative measure of how difficult it is to mine a new block for the blockchain.

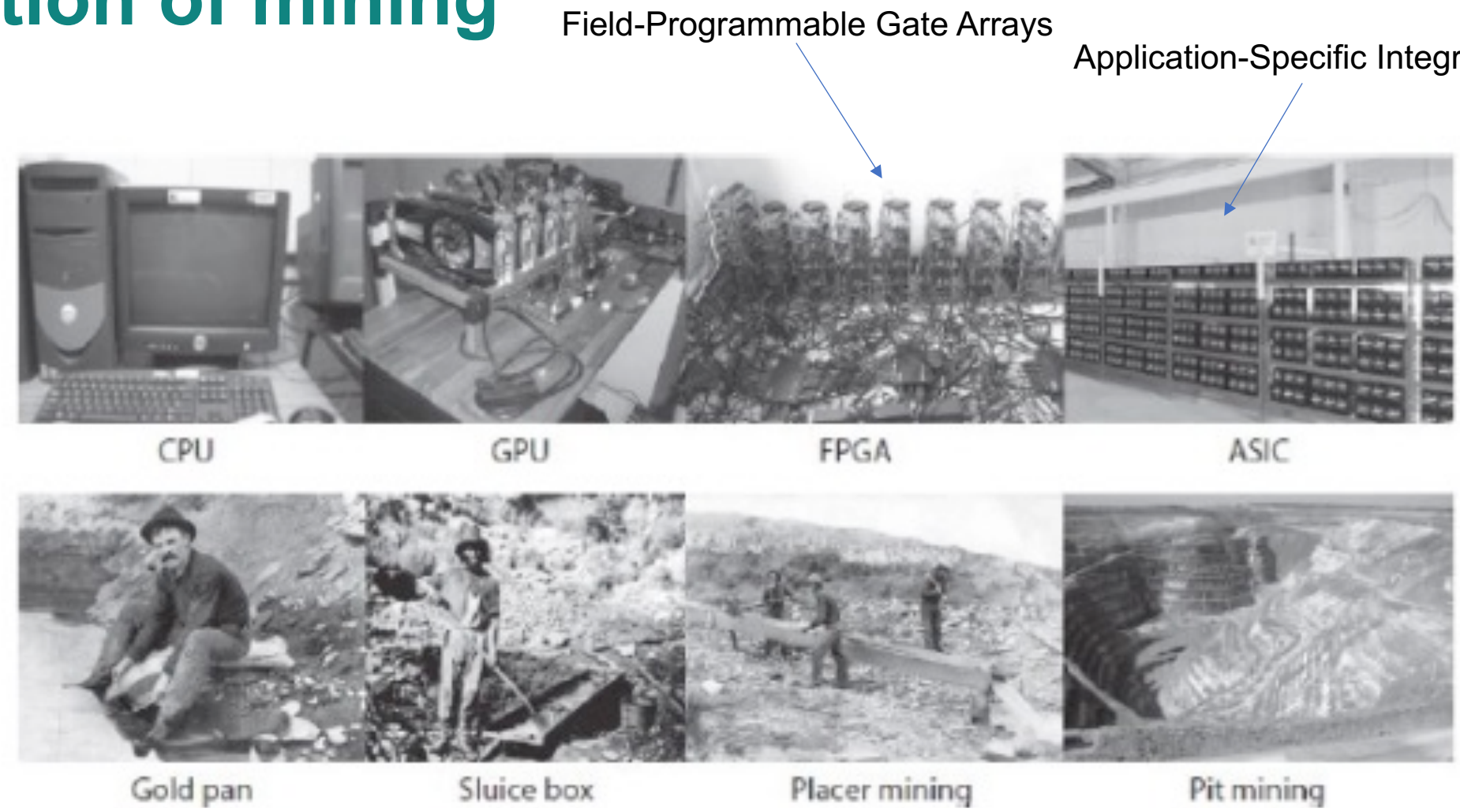
Bitcoin Mining

Bitcoin Explorer



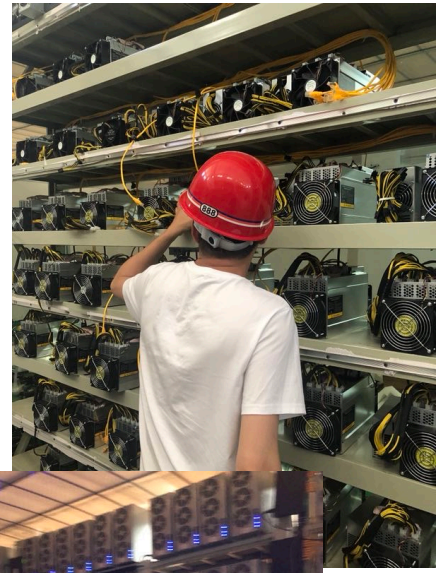
Bitcoin Mining

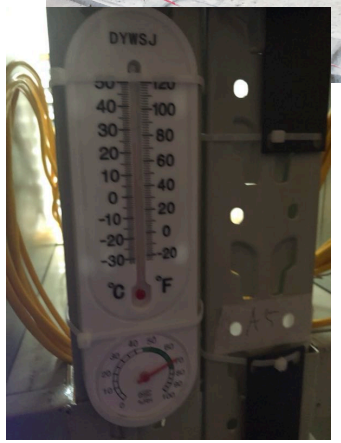
Evolution of mining



Bitcoin Mining

Hashrate Distribution





References to read

[1] Nakamoto, Satoshi (31 October 2008). "Bitcoin: A Peer-to-Peer Electronic Cash System" (PDF). bitcoin.org.

[2] Dai, W., 1998. b-money, 1998. http://www.weidai.com/bmoney.txt

[3] Back, A., 2002. Hashcash-a denial of service counter-measure.