Politecnico di Milano
School of Industrial and Information Engineering
Master of Science in Mathematical Engineering

Blockchain Notarization: extensions to the OpenTimestamps protocol

Supervisors: Prof. Daniele Marazzina
Prof. Ferdinando M. Ametrano

Master thesis by:
Andrea Brandoli
ID: 842282

Academic year 2018-2019

The secret to happiness is freedom
and the secret to freedom is courage.

Thucydides

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Chapter 2

Distributed Consensus

2.1 Distributed Systems

With the persistent expansion of technology in the digital age we are going through, distributed systems are becoming more and more widespread.

On one side big companies operate on a global scale with thousands of machines deployed all over the world, big data are stored in various data centers and computational power is shared on multi-core processors or computing clusters.

On the other side, every day thousands of people withdraw money in a ATM point, a perfect example of distributed system. But simply, just think about a modern smartphone: it can share multiple data on the cloud and contain multiple co-working processing devices.

However, there is no a unique formal definition of distributed system. The one that fits better our interest is:

Definition 2.1.1. (distributed system)[1]. *A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another. The components interact with one another in order to achieve a common goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components.*

Therefore, the main reasons for using distributed systems are:

[1] `https://en.wikipedia.org/wiki/Distributed_computing`

- Scalability: Distributed systems should be scalable with respect to geography, administration or size.
- Performance: Compared to other models, distributed models are expected to give a higher boost to performance.
- Fault Tolerance: A cluster of multiple machines is inherently more fault-tolerant than a single source of failure.
- Reliability: Data is replicated on different machines in order to prevent loss.
- Availability: Data is replicated on different machines to minimize latency and provide faster access.

However, beautiful features like these often have a downside, bringing to light some challenging problems:

- Security: Especially when networks are public.
- Coordination[2]: In public distributed systems coordination problems are prevalent if proper protocols or policies are not in place; agents could be malevolent or malicious.

Thus, it can be easily inferred that one of the main challenges of distributed systems is to achieve overall system reliability in the presence of a number of faulty processes. Examples of applications of consensus include whether to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, clock synchronization, PageRank, load balancing and many others. Let us get in a detailed study of what consensus is and how consensus can be achieved in a distributed system.

2.2 The Consensus Problem

Networks are composed by many agents, called *nodes*. Think of a computer network, nodes are either *honest*, executing programs faithfully, or *byzantine* [11], exhibiting arbitrary behavior. We will also define them *correct* and *faulty*, but not as alternatives to honest and byzantine. A correct node is an honest node that always eventually makes progress. A faulty node is a

[2]In the fin-tech industry, coordination problems come with various names: consistency, agreement, consensus or Blockchain.

Byzantine node or an honest node that has crashed or will eventually crash. Note that honest and Byzantine are mutually exclusive, as are correct and faulty. However, a node can be both honest and faulty.

Definition 2.2.1. (node). *We call a single actor in the system node. In a computer network the computers are the nodes, in the classical client-server model both the server and the client are nodes, and so on.*

Definition 2.2.2. (byzantine node)[3]. *A node which can have arbitrary behaviour is called byzantine. This includes "anything imaginable", e.g., not sending any massage at all, or sending different and wrong messages to different neighbors, or lying about the input value.*

We assume that each pair of nodes is connected by a link, which is a bi-directional reliable virtual circuit and therefore messages sent on this link are delivered, eventually, and in the order in which they were sent (i.e., an honest sender keeps retransmitting a message until it receives an acknowledgment or crashes). A receiver can tell who sent a message (e.g., using MACs), so a Byzantine node cannot forge a message so it is indistinguishable from a message sent by an honest node.

In the consensus problem nodes run *actors*, that are either *proposers*, each of which proposes a *proposal*, or *deciders*, each of which *decides* one of the proposals. Assuming there exists at least one correct proposer (i.e., a proposer on a correct node), the goal of a consensus protocol is to ensure each correct decider decides the same proposal, even in the face of faulty proposers. A node may run both a proposer and a decider, in practice a proposer often would like to learn the outcome of the agreement.

Definition 2.2.3. (consensus). *There are n nodes, of which at most f might crash, i.e., at least $n - f$ nodes are correct. Node i starts with a proposed value v_i. The nodes must decide for one of those values, satisfying the following properties:*

- *Agreement: All correct nodes decide for the same proposal.*
- *Termination: All correct nodes terminate in finite time.*
- *Validity: The decision value must be the proposal of some proposer.*

[3]Before the term *byzantine* was coined, the terms Albanian Generals or Chinese Generals were used in order to describe malicious behavior. When the involved researchers met people from these countries they moved, for obvious reasons, to the historic term byzantine.

What is really interesting is to learn how to achieve distributed consensus in an asynchronous network in presence of byzantine nodes. Starting with the historical formulation of the problem and a first example of solved consensus in a model that is synchronous, we will finally face a famous impossibility result in the asynchronous case.

2.2.1 The Byzantine Generals' Problem

Distributed computer systems that want to be reliable must handle faulty components that give conflicting information to different parts of the system.

Figure 2.1: Conquest of Constantinople

Source: `https://medium.com/all-things-ledger/the-byzantine-generals-problem-168553f31480`

The "Byzantine Generals' Problem" is the abstraction of the aforementioned situation. We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals communicate with one another as well as with all lieutenants only by messenger. After observing the enemy, they must decide upon a common plan of action: the exact time to attack all at once or, if faced by fierce resistance, the time to retreat all at once. The army cannot hold on forever, if the attack or retreat is without full strength then brutal defeat is

the only possible outcome. However, some of the generals may be traitors, trying to prevent loyal generals from reaching agreement.

For simplicity, we can restrict ourselves to the case of a commanding general sending an order to his lieutenants, obtaining the following problem.

Definition 2.2.4. (byzantine generals' problem). *A commanding general must send an order to his $n-1$ lieutenant generals such that the following conditions*[4] *are satisfied:*

1. *All loyal lieutenants obey the same order.*
2. *If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.*

2.2.2 Byzantine Agreement

Achieving consensus (as in Definition 2.2.3) in a system with byzantine nodes is hard stuff. A careful reader immediately realizes that fulfilling agreement and termination is straight-forward, but what about validity? Reminding that a byzantine node can be malevolent lying about its input value, we must specify different types of validity:

Definition 2.2.5. (any-input validity)[5]. *The decision value must be the proposed value of any node.*

As we can see, this definition does not still make sense in presence of byzantine nodes; we would wish for a differentiation between byzantine and correct inputs.

Definition 2.2.6. (correct-input validity). *The decision value must be the proposed value of a correct node.*

Fulfilling this particular validity definition is not so simple, as byzantine nodes following correctly a specified protocol but lying about theirs input values are indistinguishable from correct nodes. An alternative could be:

Definition 2.2.7. (all-same validity). *If all correct nodes start with the same proposed value v, the decision value must be v.*

[4]Conditions 1 and 2 are called the *interactive consistency* conditions

[5]This is the validity definition we implicitly used for consensus, in Definition 2.2.3

Here, if the decision values are binary, then correct-input validity is induced by all-same validity. Else, if the input values are not binary, all-same validity is not useful anymore.

Now that we have clear in mind what are the ingredients for the consensus recipe, a question bales to us: which are the algorithms that solve byzantine agreement? What type of validity they fulfill? The King algorithm is one of the best examples, but we have to restrict ourselves to the so-called synchronous model.

Definition 2.2.8. (synchronous model). *In the synchronous model, nodes operate in synchronous rounds. In each round, each node may send a message to the other nodes, receive the message sent by the other nodes, and do some local computation.*

Algorithm 2.1 King Algorithm (for $f < n/3$)

1: $x =$ my input value
2: **for** phase $= 1$ to $f + 1$ **do**
Round 1
3: Broadcast $value(x)$
Round 2
4: **if** some $value(y)$ at least $n - f$ times **then**
5: Broadcast $propose(y)$
6: **end if**
7: **if** some $propose(z)$ received more than f times **then**
8: $x = z$
9: **end if**
Round 3
10: Let node v_i be the predefined king of this phase i
11: The king v_i broadcasts its current value w
12: **if** received strictly less than $n - f$ $propose(x)$ **then**
13: $x = w$
14: **end if**
15: **end for**

To be rigorous, we must state some useful lemmas in order to prove that Algorithm 2.1 solves byzantine agreement.

Lemma 2.2.1. *Algorithm 2.1 fulfills the all-same validity.*

Lemma 2.2.2. *There is at least one phase with a correct king.*

Lemma 2.2.3. *After a round with a correct king, the correct nodes will not change their values v anymore, if $n > 3f$.*

Theorem 2.2.1. *Algorithm 2.1 solves byzantine agreement.*

Proof. The king algorithm reaches agreement as either all correct nodes start with the same value, or they agree on the same value latest after the phase where a correct node was king according to Lemmas 2.2.2 and 2.2.3. Because of Lemma 2.2.1 we know that they will stick with this value. Termination is guaranteed after $3(f + 1)$ rounds, and all-same validity is proved in Lemma 2.2.1. □

However, this is not the end of the story about distributed consensus. In order to dig into the hearth of this work we must introduce consensus results in the asynchronous model.

Definition 2.2.9. (asynchronous model). *In the asynchronous model, algorithms are event based ("upon receiving message ..., do ..."). Nodes do not have access to a synchronized wall-clock. A message sent from one node to another will arrive in a finite but unbounded time.*

Definition 2.2.10. (asynchronous runtime). *For algorithms in the asynchronous model, the runtime is the number of time units from the start of the execution to its completion in the worst case (every legal input, every execution scenario), assuming that each message has a delay of at most one time unit.*

We will now present a famous algorithm (Algorithm 2.2) by Ben-Or [7], which tries to solve asynchronous byzantine agreement.

Unfortunately, Algorithm 2.2 is just a proof of concept that asynchronous byzantine agreement can be achieved, but practically it is unfeasible due to its exponential runtime.

Algorithm 2.2 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/9$)

1: $x_i \in \{0, 1\}$ ▷ input bit
2: $r = 1$ ▷ round
3: decided = false
4: Broadcast $propose(x_i, r)$
5: **repeat**
6: Wait until $n - f$ *propose* messages of current round r arrived
7: **if** at least $n - 2f$ *propose* messages contain the same value x **then**
8: $x_i = x$, decided = true
9: **else if** at least $n - 4f$ *propose* messages contain the same value x **then**
10: $x_i = x$
11: **else**
12: choose x_i randomly, with $Pr[x_i = 0] = Pr[x_i = 1] = 1/2$
13: **end if**
14: $r = r + 1$
15: Broadcast $propose(x_i, r)$
16: **until** decided (see Line 8)
17: decision = x_i

2.2.3 Impossibility Result

A solution to the Byzantine Generals' Problem 2.2.4 may seem a piece of cake. That's not true at all. Its difficulty arises from the fact that if the generals can send only oral messages, then no solution will work unless more than two-thirds of the generals are loyal. An oral message is one whose contents are completely under the control of the sender, so a traitorous sender can transmit any possible message. Such a message corresponds to the type of message that computers normally send to one another.

Let us now show that, if only oral messages are allowed, there is no solution for three generals with a single traitor. For simplicity, the only possible messages that generals can send are "attack" or "retreat". This restriction opens two possible scenarios. In the first scenario, shown in Figure 2.2, a loyal commander sends an "attack" order to his lieutenants, but lieutenant 2 is malevolent and report "retreat" to lieutenant 1. In this situation, in order to satisfy Condition 2 of problem 2.2.4, lieutenant 1 must obey the order to attack. In the second scenario, shown in Figure 2.3, the commander is a traitor and sends an "attack" command to lieutenant 1, and a "retreat" order to lieutenant 2. But, lieutenant 1 does not know who is the traitor and

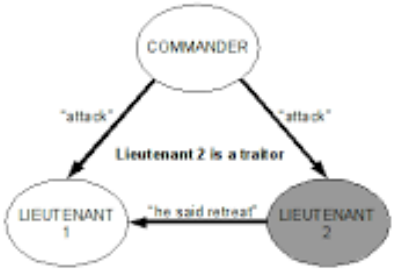

Figure 2.2: Lieutenant 2 is a traitor

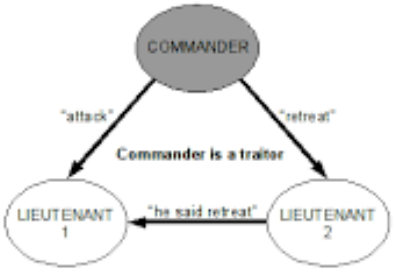

Figure 2.3: Commander is a traitor

Source: `https://www.researchgate.net/figure/Byzantine-Generals-Problem-Lamport82_fig4_263046309`

cannot tell the order the commander sent to lieutenant 2. Thus, lieutenant 1 cannot distinguish in which scenario he belongs, so he always obey to the "attack" order. With a similar argument, if lieutenant 2 receive a "retreat" order from the commander, he must follow him even if lieutenant 1 reports him to attack. This violates Condition 1 of problem 2.2.4.

A demanding reader could comply about this nonrigorous result. A famous result[6] of Fisher, Lynch and Paterson [9] formally proves impossibility of distributed consensus with one faulty process.

Due to this proven impossibility result, how Satoshi Nakamoto reaches consensus in Bitcoin, a decentralized, distributed, peer-to-peer network? In the next chapter we will delve into this argument.

[6]This result was awarded the 2001 PODC Influential Paper Award (now called Dijkstra Prize).

Chapter 3

Nakamoto Consensus in Bitcoin

Bitcoin is a distributed, decentralized, peer-to-peer electronic payment system based on cryptographic proof instead of trust, allowing transactions between two counterparts without the need for a trusted third party. However, in late 2008, when the white paper was published by his author Satoshi Nakamoto[1], it lacked a formalization of the protocol and of the guarantees it claimed to provide.

This chapter delves into the core innovation behind Bitcoin, i.e. *Nakamoto consensus*, term that is commonly used to refer to Bitcoin's novel consensus mechanism, which allows mutually distrusting pseudonymous identities to reach eventual agreement.

3.1 Eventual Consistency

By its very nature, the Bitcoin network is subject to a type of failure called *network partition*, where a network splits into at least two parts that cannot communicate with each other, often due to software bugs, incompatible protocol versions, or simply network disconnections. Thus, Bitcoin is inherently characterized by a trade-off between *consistency*, *availability* and *partition tolerance*. Let us be more precise.

[1]The name Satoshi Nakamoto is the pseudonym adopted by the creator of Bitcoin. While his identity remains a mystery, some information is known. He registered the domain bitcoin.org in August 2008 and in October 2008 he publicly released the famous white paper. In the Bitcoin's early days he participated extensively in forums and mailing lists maintaining the souce code. During the next two years other contributors slowly taken over the project maintenance and he stopped communicating, leaving a veil of mystery.

Definition 3.1.1. (consistency). *All nodes in the system agree on the current state of the system.*

Definition 3.1.2. (availability). *The system is operational and instantly processing incoming requests.*

Definition 3.1.3. (partition tolerance). *Partition tolerance is the ability of a distributed system to continue operating correctly even in the presence of a network partition.*

In practice, only two of these properties can be reached simultaneously; a theorem by Brewer proves this result.

Theorem 3.1.1. (CAP theorem). *It is impossible for a distributed system to simultaneously provide consistency, availability and partition tolerance. A distributed system can satisfy any two of these but not all three.*

Proof. Let us assume two nodes that share some state. The nodes belongs to different partitions, so they cannot communicate each other. Assume a request wants to update the state and contacts one of the two nodes. The node may either: 1) update its local state, resulting in a situation of inconsistent states, or 2) not update, resulting in a system no longer available for updates. □

Recalling the Definition 2.2.3 of consensus and the properties it must satisfy, a clever intuition of Nakamoto is to weaken the agreement property to hold probabilistically and not deterministically, in order to deal with an asynchronous network. In particular the aforementioned trade-off is in advantage of partition tolerance rather than consistency. In fact, state changes of the underlying transaction ledger (i.e. the blockchain), are rendered probabilistic and the decision on a specific value of the state reaches $Pr(1)$ when $\lim_{r\to\infty}$, where r is the number of rounds in the consensus protocol.

Therefore, we can look at Bitcoin as an example of *eventual consistency*.

Definition 3.1.4. (eventual consistency) *If no new updates to the shared state are issued, then eventually the system is in a quiescent state, i.e., no more messages need to be exchanged between nodes, and the shared state is consistent.*

Now that we have clear in mind the consensus problem from a computer science perspective, we must move forward and study how practically Nakamoto consensus works. Starting from a brief summary of some basic Bitcoin mechanics, we will finally show how cleverly Nakamoto solves the consensus problem in Bitcoin, going through cryptography tools and game theory aspects.

3.2 Bitcoin Design

We already defined Bitcoin as a distributed, decentralized, peer-to-peer electronic payment system, but we lacked about defining the underlying digital asset, i.e. *bitcoin*, the challenges in building it, and also the fundamental bricks of the Bitcoin wall. Combining all these aspects is crucial to fully understand Nakamoto consensus.

3.2.1 Constructing a Decentralized Digital Currency

Ownership is the first challenge of constructing a virtual currency. Imagine units of fiat currencies, their ownership depends on their form. If these units are in the form of paper notes or metal coins, ownership is simply determined by physical possession. Else, if these units are digitally stored in a bank account, ownership is determined by possession of the credentials to spend them. But what about ownership for a decentralized virtual currency? Who maintains the ledger of credentials? How can we guarantee the integrity of that ledger? In a network like Bitcoin nodes are free to leave at any time, thus a single source of trust (a node or a group of them) cannot be given this responsibility, even if one-way functions protects credentials. A solution could be that all nodes maintain a copy of that ledger, but what about validation? Can I feasible query all the nodes? Integrity is also crucial: usually digital signatures ensure it, but reminds the nodes could act maliciously and so, can I trust the signer?

Another big challenge is the so called *double spending* problem. This refers to the fact that the owner of some units of digital currency could spend them more than once. With fiat currencies, granted by a central authority, this problem cannot occur. In fact the bank database is the single source of truth with regard to the deposited amount of currency. Thus, spending with f.e. an online transfer, would immediately result in the rebalancing of that account to reflect the spend. However, in a peer-to-peer decentralized network some

malicious agent could try to perform a double spend taking advantage of the delay in updating the shared ledger. An example will explain better.

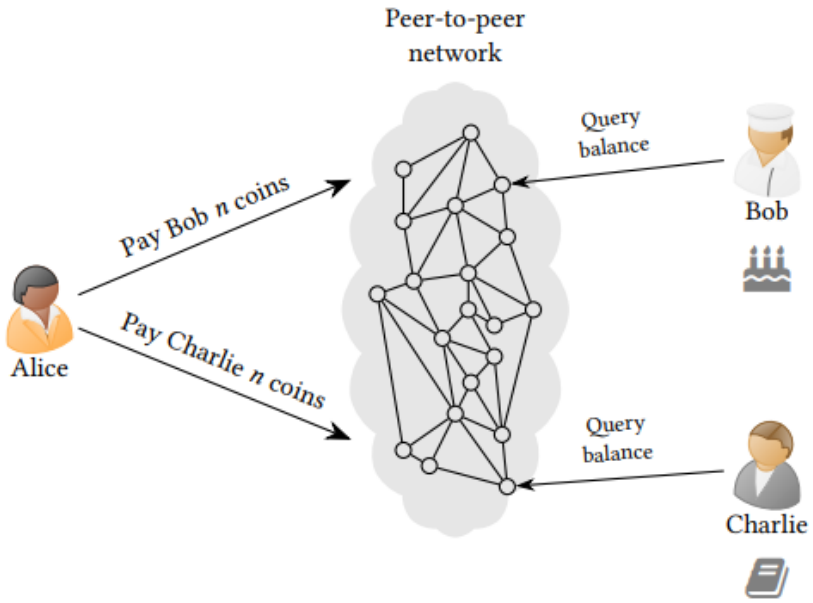

Figure 3.1: Illustration of the double spending problem

Consider the scenario described in Figure 3.1 where Alice tries to perform a double spend using some digital coins in her wallet. She would purchase a cake from Bob and a book from Charlie spending the same n coins. For simplicity, let us assume that the two items cost the same, precisely n units of currency. Thus, in order to finalize the purchase, both Bob and Charlie will provide Alice with their wallet addresses. Then Alice creates two different transactions, both spending the same n coins, one paying Bob for the cake and the other paying Charlie for the book. The network will only accept one of them because the two transactions conflict. But there are several nodes in the network, some of which can be made to accept one of the transactions and the rest to accept the other. So Alice can transmit the transaction paying Bob to the portion of the network which Bob is connected to and the same argument holds for Charlie's transaction. Therefore, when Bob and Charlie query the balances in their private wallets they both will find a valid payment. If they provide Alice with their goods before being aware that the two transactions are conflicting, the result is that Alice has successfully performed a double spend.

Nakamoto solves brilliantly the aforementioned challenges in Bitcoin combining cryptography and social incentive engineering. But first we need a brief summary of the mechanics of the protocol.

3.2.2 Transactions and Blockchain Structure

In Bitcoin, a coin or a *bitcoin* is defined as a chain of digital signatures. Coins cannot be combined, subdivided or transferred, they can only be entirely consumed as transaction inputs (TxIn) in order to create new output coins (TxOut). Thus, a transaction output can only be in two states, *spent* or *unspent*. The set of current unspent coins takes the name of UTXO (Unspent Transaction Outputs). So, a *transaction* assumes the role of the fundamental unit of state in the Bitcoin protocol.

More precisely, each Tx[2] output consists of an amount of bitcoins and a cryptographic puzzle, the so-called *locking script*, which determines the spending conditions for that amount. An example of such a puzzle could be a request for a digital signature which can be validated with a public key. On the other side, a Tx input is characterized by a pointer referencing the UTXO that it consumes and an *unlocking script*, the solution to the aforementioned locking script, which proves that funds in the considered transaction can be honestly spent.

All the transactions, starting from the time the protocol was launched[3], are securely recorded in a distributed ledger, shaped as a sequence of blocks, the *blockchain*. In a technical cryptographic perspective, the *blockchain* is a hash pointer linked list, whose aim is to be a *tamper-evident* log. That is, a log data structure that stores some data (mainly transactions) allowing only to append new data onto the end of the log but, if somebody alters some earlier data then it would be easily detected. Figure 3.2 provides a graphic idea of the structure.

[2]It's a common practice in Bitcoin community to abbreviate the word *transaction* as Tx or sometimes TX.

[3]The first block of the chain, called the *genesis* block, was generated in January 2009

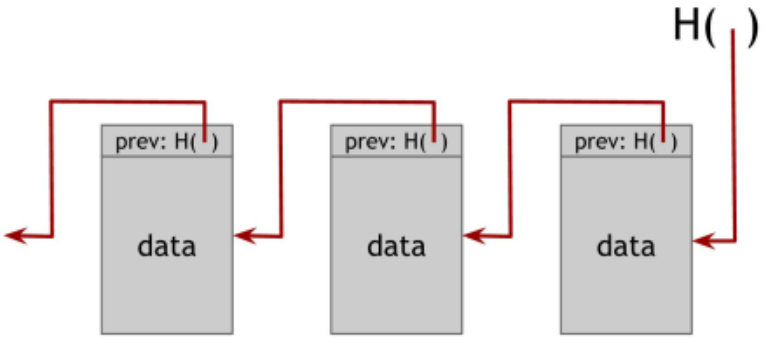

Figure 3.2: Blockchain as a hash pointer linked list

Source: [13]

The aforementioned *tamper-evidence* property derives from cryptography, so we must introduce some primitives. But first a bit of history. The ideas behind the blockchain trace back to a paper by Haber and Stornetta in 1991 [10]. They proposed a method to *timestamp* digital documents, that is a method to give an idea of when a document came into exixtance. To achieve this goal, a timestamping server receives client documents to timestamp. Then it signs the documents (one by one) together with the current time, and a special type of pointer that links to a piece of data instead of a location, pointing to the previous document. Thus, if some data is tampered, that pointer turns to be invalid. Finally the timestamping server issues a certificate containing such information. Therefore it is clear that each document's certificate ensures the integrity of the previous document.

3.2.3 Hash Functions and Pointers

The more the world becomes digital, the more we need information security. Cryptography mixes advanced mathematical and computer science techniques to provide confidentiality, data integrity, authentication and non-repudiation.

A *hash function* is a particular mathematical function that maps an arbitrary length input message into a short, fixed-length bit string, usually called *digest* or *hash value*. Hash functions can be seen as a digital fingerprint of a message, i.e. a unique representation of a message.

Definition 3.2.1. *A function* $h : \{0,1\}^* \to \{0,1\}^n$ *is a hash function if it is computable in polynomial time in the length of the input.*

This definition lacks of security arguments. There are three main properties that hash functions need to possess in order to provide security:

Definition 3.2.2. *Let h be a hash function, the following properties may hold:*

- *preimage resistance (one-wayness): given $h(x)$ it is not feasible to compute x;*
- *second-preimage resistance (weak collision resistance): given x it is not feasible to compute y s.t. $x \neq y$, $h(x) = h(y)$;*
- *collision resistance (strong collision resistance): it is not feasible to find x, y s.t. $x \neq y$, $h(x) = h(y)$.*

A hash function that fulfills the preimage resistance property *hides* the input, so one cannot derive a matching message from a hash value. A second-preimage resistant hash function prevents that, given a message, one can find another message that produces the same output once hashed. Unfortunately such a property cannot exist in theory, due to the so-called *pigeonhole principle*[4]. Suppose to be the owner of 100 pigeons but in your pigeon loop are only 99 holes, at least one pigeonhole will be occupied by 2 birds. The same argument holds for hash functions. Since the output of a hash function has a fixed bit length, say n bits, there exist only 2^n possible output values. However the length of input values is arbitrary, resulting in an infinite number of possible inputs. Thus, weak collision can occur. Fortunately, given today's computers, an output length of $n = 80$ bits is sufficient to prevent collisions. To sum up, collisions do exist, but they are *unfeasible*. Finally, the strong collision resistance property refers to the problem of finding two messages that generate the same hash value. With an output length of $n = 80$ bits as before, producing 2^{80} possible values, one may think that finding two inputs generating the same output would require around $\frac{2^{80}}{2}$, a half of the possible values. That's completely wrong, due to the *birthday paradox*, a powerful cryptographic tool which deserves a formal definition.

Definition 3.2.3. (birthday paradox). *The birthday paradox concerns the probability that, given a set of n randomly chosen people, some pair of them will have the same birthday.*

[4]Formally this principle takes the name of *Dirichlet's drawer principle*, but it is commonly referred to as the *pigeonhole principle*.

The correct approach to solve this problem is to compute the probability of two people *not* having the same birthday:

$$P(\text{2 people not colliding}) = \left(1 - \frac{1}{365}\right)$$

If a third person is joining:

$$P(\text{3 people not colliding}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right)$$

Hence, the probability for t people having no birthday collision is given by:

$$P(t \text{ people not colliding}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{t-1}{365}\right)$$

Then, making some simple calculations, it turns out that it only requires 23 people to have a probability of about 0.5 for a birthday collision:

$$\begin{aligned} P(\text{at least one collision}) &= 1 - P(\text{no collision}) \\ &= \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{23-1}{365}\right) \\ &= 0.507 \approx 50\% \end{aligned}$$

The search of a collision for a hash function is exactly the same problem as the birthday problem, obviously with different parameters. Again, let n be the output length in bits, thus resulting in 2^n possible output values. Here, the probability of no collisions among t hash values is:

$$\begin{aligned} P(\text{no collision}) &= \left(1 - \frac{1}{2^n}\right) \cdot \left(1 - \frac{2}{2^n}\right) \cdots \left(1 - \frac{t-1}{2^n}\right) \\ &= \prod_{i=1}^{t-1} \left(1 - \frac{i}{2^n}\right) \\ &\approx \prod_{i=1}^{t-1} e^{-\frac{i}{2^n}} \\ &\approx e^{-\frac{t(t-1)}{2 \cdot 2^n}} \end{aligned}$$

Remind that we want to find how many messages are required to find a collision. Let λ be the probability of at least one collision:

$$\lambda \approx 1 - e^{-\frac{t(t-1)}{2^{n+1}}}$$

Solving for t and observing that, since in practice $t >> 1$, it holds that $t^2 \approx t(t-1)$ we obtain:

$$t \approx 2^{(n+1)/2} \sqrt{\ln\left(\frac{1}{1-\lambda}\right)}$$

The main takeaway of these computations is that the number of messages needed to find a collision of hash values is proportional to the square root of the number of possible output values.

Riportare qui una tabellina coi valori, prendi da Pelzl and Paar (vedere per il titoletto delle sole colonne).

Example 3.2.1. *SHA256 belongs to the family SHA2, the generation of hash functions following SHA1*[5]*. It outputs strings 256 bit long and, at the moment, it is considered to satisfy all the properties in Definition 3.2.2. Several systems are built upon this assumption. In addition, as other hash functions, SHA256 could be modeled as a random oracle, a fixed input will provide always the same output, since hash functions are deterministic, but the outputs corresponding to new inputs will give results that are indistinguishable from a uniform distribution. Note how a little change in the input produces outputs very dissimilar in both hash functions, this feature helps to spot alterations in the inputs.*

```
SHA256(b'Hello World!\n') = 03ba204e50d126e4674c005e04d82e84c21
                            366780af1f43bd54a37816b6ab340
SHA256(b'Hello World\n')  = d2a84f4b8b650937ec8f73cd8be2c74add5
                            a911ba64df27458ed8229da804a26
```

Since SHA256 is available, the use of SHA1 should be avoided unless there is a particular motivation.

Sistema l'esempio ed aggiungi hash pointers, poi la parte del mining, pow, network behaviour, supply e forks. Spiega dunque come Nakamoto risolve le main challenges nel costruire una moneta decentralizzata.

[5]Currently, it is available SHA3, which contains, among others, KECCAK256.

Chapter 4

Cryptographic Background

The aim of this chapter is to provide all the cryptographic primitives to deeply understand distributed consensus and blockchain immutability in Bitcoin. Furthermore, a cryptographic background accompanies us to the hearth of this work: blockchain notarization.

The more the world becomes digital, the more we need information security. Cryptography mixes advanced mathematical and computer science techniques to provide confidentiality, data integrity, authentication and non-repudiation. In order to build security protocols we must introduce cryptographic primitives.

4.1 Hash Functions

A hash function is a particular mathematical function that maps an arbitrary length input message into a short, fixed-length bit string, usually called *digest* or *hash value*. Hash functions can be seen as a digital fingerprint of a message, i.e. a unique representation of a message.

Chapter 5

OpenTimestamps User Guide

Chapter 6

Proposed Extensions to the Protocol

Chapter 7

Conclusions

Appendix A

Bitcoin from the command line

This appendix is aimed at providing a technical walkthrough the Bitcoin network from the command line point of view, supposing a Unix-like operating system. Initially, we will configure a machine to launch a Bitcoin node; then we will introduce some of the most important RPC (taken from... metti la fonte) in order to perform basic operations. Lastly the OpenTimestamps protocol comes into play: we will practically see how to timestamp a document using the Python client and how to set up a calendar server.

A.1 Running a Bitcoin Core node

To get started with Bitcoin, you first need to join the network running a node. Following these simple steps will make the first experience a lot easier.

- Setup variables (for an easy installation):

```
$ export BITCOIN=bitcoin-core-0.17.0
```

- Download relevant files (Remark: every time you find username replace it with your personal one):

```
$ wget https://bitcoin.org/bin/$BITCOIN/$BITCOINPLAIN-
  x86_64-linux-gnu.tar.gz -O ~username/$BITCOINPLAIN-
  x86_64-linux-gnu.tar.gz
$ wget https://bitcoin.org/bin/$BITCOIN/SHA256SUMS.asc -O ~
  username/SHA256SUMS.asc
$ wget https://bitcoin.org/laanwj-releases.asc -O ~username
  /laanwj-releases.asc
```

- Verify Bitcoin signature (in order to verify that your Bitcoin setup is authentic):

```
$ /usr/bin/gpg --import ~username/laanwj-releases.asc
$ /usr/bin/gpg --verify ~username/SHA256SUMS.asc
```

  Amongst the info you get back from the last command should be a line telling "Good signature", do not care the rest.

- Verify Bitcoin SHA (Secure Hash Algorithm) of the .tar file against the expected one:

```
$ /usr/bin/sha256sum ~username/$BITCOINPLAIN-x86_64-linux-
  gnu.tar.gz | awk '{print $1}'
$ cat ~username/SHA256SUMS.asc | grep $BITCOINPLAIN-x86_64-
  linux-gnu.tar.gz | awk '{print $1}'
```

  From this verification you must obtain the same number.

- Install Bitcoin:

```
$ /bin/tar xzf ~username/$BITCOINPLAIN-x86_64-linux-gnu.tar
  .gz -C ~username
$ sudo /usr/bin/install -m 0755 -o root -g root -t /usr/
  local/bin ~username/$BITCOINPLAIN/bin/*
$ /bin/rm -rf ~username/$BITCOINPLAIN/
```

- Create the configuration file:

  First, create the directory you want to put your data into:

```
$ /bin/mkdir ~username/.bitcoin
```

  Then create the configuration file (as you need it):

```
$ cat >> ~username/.bitcoin/bitcoin.conf << EOF
# Accept command line and JSON-RPC commands
server=1
# Username for JSON-RPC connections
rpcuser=invent_a_username_here
# Password for JSON-RPC connections
rpcpassword=invent_a_long_pwd_here
# Enable Testnet chain mode
testnet=1
EOF
```

The *bitcoin.conf* file is the default file which Bitcoin daemon reads when launched, and it contains instructions to configure the node (e.g. mainnet, testnet or regtest mode). In this example we define a *rpcuser* and *rpcpassword* to enable RPC, and we tell the node to synchronize with the Bitcoin Testnet chain. (metti una nota qui per il sito dei conf file).

Finally, limit the permissions to your configuration file (not really needed, but more secure):

```
$ /bin/chmod 600 ~username/.bitcoin/bitcoin.conf
```

- Start the daemon[1]:

```
$ bitcoind -daemon
```

We are now ready to be part of the Bitcoin network !
In the next section we will learn how to manage the Bitcoin command line interface.

A.2 Bitcoin-cli: command line interface

In this section we will answer the following question: "*What is RPC and which calls are the most used in Bitcoin?*".

Without further going into details regarding computer science[2], a RPC (Remote Procedure Call) is a powerful technique for constructing distributed, client-server based applications. It is based on the extension of the conventional procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. Our interest is for the `bitcoin-cli`, a handy interface that lets the user send commands to the `bitcoind`. More specifically, it lets you send RPC commands to the `bitcoind`. Generally, the `bitcoin-cli` interface

[1] In multitasking computer operating systems, a *daemon* is a computer program that runs as a background process, rather than being under direct control of an interactive user. Traditionally, the process names of a daemon end with the letter *d*; `bitcoind` is the one which implements the Bitcoin protocol for remote procedure call (RPC) use.

[2] For more details, see https://en.wikipedia.org/wiki/Remote_procedure_call

is much more clean and user friendly than trying to send RPC commands by hand, using `curl` or some other method.

The first thing to do for a beginner user is to look for the full list of calls invoking the `help` command:

```
$ bitcoin-cli help

== Blockchain ==
getbestblockhash
getblock "blockhash" ( verbosity )
getblockchaininfo
getblockcount
getblockhash height
getblockheader "hash" ( verbose )
getblockstats hash_or_height ( stats )
getchaintips
getchaintxstats ( nblocks blockhash )
getdifficulty
getmempoolancestors txid (verbose)
getmempooldescendants txid (verbose)
getmempoolentry txid
getmempoolinfo
getrawmempool ( verbose )
gettxout "txid" n ( include_mempool )
gettxoutproof ["txid",...] ( blockhash )
gettxoutsetinfo
preciousblock "blockhash"
pruneblockchain
savemempool
scantxoutset <action> ( <scanobjects> )
verifychain ( checklevel nblocks )
verifytxoutproof "proof"

== Control ==
getmemoryinfo ("mode")
help ( "command" )
logging ( <include> <exclude> )
stop
uptime

== Generating ==
generate nblocks ( maxtries )
generatetoaddress nblocks address (maxtries)

== Mining ==
getblocktemplate ( TemplateRequest )
getmininginfo
```

```
getnetworkhashps ( nblocks height )
prioritisetransaction <txid> <dummy value> <fee delta>
submitblock "hexdata"  ( "dummy" )

== Network ==
addnode "node" "add|remove|onetry"
clearbanned
disconnectnode "[address]" [nodeid]
getaddednodeinfo ( "node" )
getconnectioncount
getnettotals
getnetworkinfo
getpeerinfo
listbanned
ping
setban "subnet" "add|remove" (bantime) (absolute)
setnetworkactive true|false

== Rawtransactions ==
combinepsbt ["psbt",...]
combinerawtransaction ["hexstring",...]
converttopsbt "hexstring" ( permitsigdata iswitness )
createpsbt [{"txid":"id","vout":n},...] [{"address":amount},{"
   data":"hex"},...] ( locktime ) ( replaceable )
createrawtransaction [{"txid":"id","vout":n},...] [{"address":
   amount},{"data":"hex"},...] ( locktime ) ( replaceable )
decodepsbt "psbt"
decoderawtransaction "hexstring" ( iswitness )
decodescript "hexstring"
finalizepsbt "psbt" ( extract )
fundrawtransaction "hexstring" ( options iswitness )
getrawtransaction "txid" ( verbose "blockhash" )
sendrawtransaction "hexstring" ( allowhighfees )
signrawtransaction "hexstring" ( [{"txid":"id","vout":n,"
   scriptPubKey":"hex","redeemScript":"hex"},...] ["privatekey1
   ",...] sighashtype )
signrawtransactionwithkey "hexstring" ["privatekey1",...] ( [{"
   txid":"id","vout":n,"scriptPubKey":"hex","redeemScript":"hex
   "},...] sighashtype )
testmempoolaccept ["rawtxs"] ( allowhighfees )

== Util ==
createmultisig nrequired ["key",...] ( "address_type" )
estimatesmartfee conf_target ("estimate_mode")
signmessagewithprivkey "privkey" "message"
validateaddress "address"
verifymessage "address" "signature" "message"

== Wallet ==
```

```
abandontransaction "txid"
abortrescan
addmultisigaddress nrequired ["key",...] ( "label" "
  address_type" )
backupwallet "destination"
bumpfee "txid" ( options )
createwallet "wallet_name" ( disable_private_keys )
dumpprivkey "address"
dumpwallet "filename"
encryptwallet "passphrase"
getaccount (Deprecated, will be removed in V0.18. To use this
  command, start bitcoind with -deprecatedrpc=accounts)
getaccountaddress (Deprecated, will be removed in V0.18. To use
   this command, start bitcoind with -deprecatedrpc=accounts)
getaddressbyaccount (Deprecated, will be removed in V0.18. To
  use this command, start bitcoind with -deprecatedrpc=accounts
  )
getaddressesbylabel "label"
getaddressinfo "address"
getbalance ( "(dummy)" minconf include_watchonly )
getnewaddress ( "label" "address_type" )
getrawchangeaddress ( "address_type" )
getreceivedbyaccount (Deprecated, will be removed in V0.18. To
  use this command, start bitcoind with -deprecatedrpc=accounts
  )
getreceivedbyaddress "address" ( minconf )
gettransaction "txid" ( include_watchonly )
getunconfirmedbalance
getwalletinfo
importaddress "address" ( "label" rescan p2sh )
importmulti "requests" ( "options" )
importprivkey "privkey" ( "label" ) ( rescan )
importprunedfunds
importpubkey "pubkey" ( "label" rescan )
importwallet "filename"
keypoolrefill ( newsize )
listaccounts (Deprecated, will be removed in V0.18. To use this
   command, start bitcoind with -deprecatedrpc=accounts)
listaddressgroupings
listlabels ( "purpose" )
listlockunspent
listreceivedbyaccount (Deprecated, will be removed in V0.18. To
   use this command, start bitcoind with -deprecatedrpc=
  accounts)
listreceivedbyaddress ( minconf include_empty include_watchonly
   address_filter )
listsinceblock ( "blockhash" target_confirmations
  include_watchonly include_removed )
listtransactions (dummy count skip include_watchonly)
```

```
listunspent ( minconf maxconf  ["addresses",...] [
  include_unsafe] [query_options])
listwallets
loadwallet "filename"
lockunspent unlock ([{"txid":"txid","vout":n},...])
move (Deprecated, will be removed in V0.18. To use this command
  , start bitcoind with -deprecatedrpc=accounts)
removeprunedfunds "txid"
rescanblockchain ("start_height") ("stop_height")
sendfrom (Deprecated, will be removed in V0.18. To use this
  command, start bitcoind with -deprecatedrpc=accounts)
sendmany "" {"address":amount,...} ( minconf "comment" ["
  address",...] replaceable conf_target "estimate_mode")
sendtoaddress "address" amount ( "comment" "comment_to"
  subtractfeefromamount replaceable conf_target "estimate_mode
  ")
setaccount (Deprecated, will be removed in V0.18. To use this
  command, start bitcoind with -deprecatedrpc=accounts)
sethdseed ( "newkeypool" "seed" )
settxfee amount
signmessage "address" "message"
signrawtransactionwithwallet "hexstring" ( [{"txid":"id","vout
  ":n,"scriptPubKey":"hex","redeemScript":"hex"},...]
  sighashtype )
unloadwallet ( "wallet_name" )
walletcreatefundedpsbt [{"txid":"id","vout":n},...] [{"address
  ":amount},{"data":"hex"},...] ( locktime ) ( replaceable ) (
  options bip32derivs )
walletlock
walletpassphrase "passphrase" timeout
walletpassphrasechange "oldpassphrase" "newpassphrase"
walletprocesspsbt "psbt" ( sign "sighashtype" bip32derivs )

== Zmq ==
getzmqnotifications
```

You can also type `bitcoin-cli help [command]` to be even more extensive about that command. For example:

```
$ bitcoin-cli help getmininginfo

getmininginfo

Returns a json object containing mining-related information.
Result:
{
  "blocks": nnn,             (numeric) The current block
  "currentblockweight": nnn, (numeric) The last block weight
```

```
  "currentblocktx": nnn,      (numeric) The last block
   transaction
  "difficulty": xxx.xxxxx     (numeric) The current difficulty
  "networkhashps": nnn,       (numeric) The network hashes per
   second
  "pooledtx": n               (numeric) The size of the mempool
  "chain": "xxxx",            (string) current network name as
   defined in BIP70 (main, test, regtest)
  "warnings": "..."           (string) any network and
   blockchain warnings
}

Examples:
> bitcoin-cli getmininginfo
> curl --user myusername --data-binary '{"jsonrpc": "1.0", "id
   ":"curltest", "method": "getmininginfo", "params": [] }' -H '
   content-type: text/plain;' http://127.0.0.1:8332/
```

Here is a list of the most general commands to get additional information about bitcoin data:

```
$ bitcoin-cli getblockchaininfo
$ bitcoin-cli getmininginfo
$ bitcoin-cli getnetworkinfo
$ bitcoin-cli getnettotals
$ bitcoin-cli getwalletinfo
```

For example `bitcoin-cli getwalletinfo` gives the user precise information about the bitcoin wallet, like the confirmed and unconfirmed balance (nota a pie' pagina per spiegare cosa sia unconfirmed balance), and the total number of transactions in the wallet.

```
$ bitcoin-cli getwalletinfo
{
 "walletname": "",
 "walletversion": 169900,
 "balance": 0.07825304,
 "unconfirmed_balance": 0.00000000,
 "immature_balance": 0.00000000,
 "txcount": 1,
 "keypoololdest": 1544714806,
 "keypoolsize": 999,
 "keypoolsize_hd_internal": 1000,
 "paytxfee": 0.00000000,
 "hdseedid": "972e6ac8d0104fec0d0e6c052d3876ada965c745",
 "hdmasterkeyid": "972e6ac8d0104fec0d0e6c052d3876ada965c745",
```

```
  "private_keys_enabled": true
}
```

In the next sections we will deal with notarization, in particular we will get into technical details of *OpenTimestamps* protocol.

A.3 OpenTimestamps client

OpenTimestamps client is a command-line tool to perform stamping of documents and to verify *OpenTimestamps* proofs, using the Bitcoin blockchain as a timestamp notary.
After this step-by-step tutorial we will be able to fully manage the Python release of the *OpenTimestamps* client.

Before installing the *OpenTimestamps* client, we must remind that while you can *create* timestamps without a local Bitcoin Core node, to *verify* proofs you need one (a pruned node is fine too, metti la nota per spiegare perche').

Let us get into a step-by-step guide to fully manage *OpenTimestamps* client in all its features:

- Install client (Python3 required):

```
$ pip3 install opentimestamps-client
```

- Install the necessary dependencies:

```
$ sudo apt-get install python3 python3-dev python3-pip
  python3-setuptools python3-wheel
```

- Pick a file or create a new one to timestamp:

```
$ cat > filename.txt
bla bla bla (write what you want)
CTRL+D (to close file and return to command prompt)
```

- Timestamp your example file:

```
$ ots stamp filename.txt
Submitting to remote calendar
  https://a.pool.opentimestamps.org
Submitting to remote calendar
  https://b.pool.opentimestamps.org
Submitting to remote calendar
  https://a.pool.eternitywall.com
Submitting to remote calendar
  https://ots.btc.catallaxy.com
```

Nice! Your timestamp request is submitted to the main public calendar servers.

Now in your current directory you must have a file named `filename.txt.ots` that is the proof of your timestamp[3].

- Verify the proof (local Bitcoin Core node needed):

```
$ ots verify filename.txt.ots
Assuming target filename is 'filename.txt'
Calendar https://bob.btc.calendar.opentimestamps.org:
  Pending confirmation in Bitcoin blockchain
Calendar https://alice.btc.calendar.opentimestamps.org:
  Pending confirmation in Bitcoin blockchain
Calendar https://btc.calendar.catallaxy.com: Pending
  confirmation in Bitcoin blockchain
Calendar https://finney.calendar.eternitywall.com: Pending
  confirmation in Bitcoin blockchain
```

Notice that you can't verify immediately the above-mentioned proof: it takes a few hours for the timestamp to get confirmed by the Bitcoin blockchain; *OpenTimestamps* is not doing one transaction per timestamp.

Now the proof is *incomplete* because it needs the assistance of a remote calendar to verify; the calendar provides the path to the Bitcoin block header. You can check for the current status of your proof file with the `info` command.

- Get detailed proof information:

[3]More specifically, an *OpenTimestamps* proof is the path up to merkleroot of the block which contains the relative transaction.

```
$ ots info filename.txt.ots
File sha256 hash: 13a1f687ddb7c1a0b12adeb708a2b464c9
  cfc24d5c9def4fa775cca81162feed
Timestamp:
append 5669aa818cc1c5bf8129d469f884fe79
sha256
 -> append 0f2a11ae083c66674d9d0d3da049079e
    sha256
    prepend f6836f41e86344eb7e4da03fe57c3e998d13594b2a27b4
  ccb86cef8ac19cf69e
    sha256
    prepend 5c502267
    append 2c767f767b02b45c
    verify PendingAttestation
  ('https://bob.btc.calendar.opentimestamps.org')
 -> append 675406ae56e9e40809ac587ee009067c
    sha256
    append 7690b939b5a663ea311992aadf2f182424cea544bbfb2a
  837ae366ba5cc1ecf4
    sha256
    prepend 5c502266
    append 1269e2007cc47092
    verify PendingAttestation
  ('https://alice.btc.calendar.opentimestamps.org')
 -> append 9b10d7ec7f0af842407d2d7cbebac9ad
    sha256
    prepend 5c502267
    append da1d41ee8803dff5
    verify PendingAttestation
  ('https://btc.calendar.catallaxy.com')
 -> append f923446e73e3a503121a56006723a6c1
    sha256
    append c35e379ef7097a65fdbfb2a594a93c75
    sha256
    prepend 5c502266
    append c47a3b08bfc2315e
    verify PendingAttestation
  ('https://finney.calendar.eternitywall.com')
```

Incomplete timestamps can be upgraded using the `upgrade` command which adds the path to the Bitcoin blockchain to the timestamp itself. Upgrading a proof isn't always available: there must be at least one completed attestation.

- Upgrade the proof:

```
$ ots upgrade filename.txt.ots
```

```
Got 1 attestation(s) from
  https://bob.btc.calendar.opentimestamps.org
Got 1 attestation(s) from
  https://alice.btc.calendar.opentimestamps.org
Got 1 attestation(s) from
  https://btc.calendar.catallaxy.com
Got 1 attestation(s) from
  https://finney.calendar.eternitywall.com
Success! Timestamp complete
```

In this case the timestamp is fully confirmed by Bitcoin Blockchain, so upgrading the proof succeeded.

Now that we know for sure that timestamping succeeded, you only left to verify the proof another time to check which block attested your timestamp (add the flag `-v` to be more verbose).

```
$ ots -v verify filename.txt.ots
Assuming target filename is 'filename.txt'
Hashing file, algorithm sha256
Got digest 13a1f687ddb7c1a0b12adeb708a2b464c9cfc24d5c9def
  4fa775cca81162feed
Attestation block hash: 000000000000000013e603482
  1d3d85e8a8ff0217482e4b2ab602ec7ab6ce7
Success! Bitcoin block 560604 attests existence as of
  2019-01-29 CET
```

Bibliography

[1] Bitcoin developer guide. `https://bitcoin.org/en/developer-reference`.

[2] Opentimestamps source code. `https://github.com/opentimestamps`.

[3] Opentimestamps website. `https://opentimestamps.org/`.

[4] Antonopoulos, A. M. *Mastering Bitcoin: Programming the Open Blockchain*, 2nd ed. O'Reilly Media, Inc., 2017.

[5] Back, A. Hashcash - a denial of service counter-measure. Tech. rep., 2002.

[6] Bayer, D., Haber, S., and Stornetta, W. S. Improving the efficiency and reliability of digital time-stamping. In *Sequences II: Methods in Communication, Security and Computer Science* (1993), Springer-Verlag, pp. 329–334.

[7] Ben-Or, M. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1983), PODC '83, ACM, pp. 27–30.

[8] Comandini, L. sign-to-contract: how to achieve trustless digital timestamping with zero marginal cost. Master's thesis, Politecnico di Milano, 2018.

[9] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr. 1985), 374–382.

[10] Haber, S., and Stornetta, W. S. How to time-stamp a digital document. *Journal of Cryptology 3* (1991), 99–111.

[11] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.

[12] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, 2009.

[13] Narayanan, A., Bonneau, J., Felten, E., Miller, A., and Goldfeder, S. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, Princeton, NJ, USA, 2016.

[14] Paar, C., and Pelzl, J. *Understanding Cryptography: A Textbook for Students and Practitioners*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[15] Todd, P. Interpreting ntime for the purpose of bitcoin-attested timestamps. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-September/013120.html`, 2016.

[16] Todd, P. Opentimestamps: Scalable, trust-minimized, distributed timestamping with bitcoin. `https://petertodd.org/2016/opentimestamps-announcement`, 2016.

[17] Todd, P. Sha1 is broken, but it's still good enough for opentimestamps. `https://petertodd.org/2017/sha1-and-opentimestamps-proofs`, 2017.

[18] Wattenhofer, R. *The Science of the Blockchain*, 1st ed. CreateSpace Independent Publishing Platform, USA, 2016.