FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's thesis

Jesper Borgstrup

Private, trustless and decentralized message consensus and voting schemes

Nicolas Pouillard, ITU
Andrzej Filinski, DIKU

November 23rd, 2014

Abstract

This thesis proposes a protocol to conduct anonymous, trustless, decentralized elections over the internet. Only registered voters can vote, multiple votes from the same voter are easily detected and discarded, and it is infeasible to determine the identity behind a given vote with a better probability than random guessing.

The voting protocol builds on top of a *decentralized deadline consensus protocol* which can form a consensus about which messages have been sent before a specific deadline. This consensus protocol can also be used to suit other purposes such as contests, auctions and applications in a decentralized manner.

The protocols use the Bitmessage protocol for communication. Bitcoin, blockchain-technology and Invertible Bloom Lookup Tables are used for defining deadlines and timestamping of messages. Linkable Ring Signatures provide a signature scheme suitable for signing votes.

A proof-of-concept client has been developed and implemented, where one can create and run elections with a basic ballot format.

Contents

1 Introduction

This thesis proposes a high-level design and working implementation of a **trustless, private and decentralized peer-to-peer voting scheme**.

In particular, the thesis has resulted in an electronic voting scheme that is fully *decentralized* — that is, there is no need and no use for a central authority; all users of the system are equal in rights and authority — as well as *trustless* — it is not necessary to have trust in any particular user, one only needs to trust the math underlying the system — and *private* — it is infeasible to determine the identity behind a specific vote with probability better than random guessing.

The voting scheme is based on a message deadline consensus protocol, which serves to achieve decentralized consensus on which messages have been sent before a specific deadline.

It is worth noting that the proposed design and protocols are relatively simple to both understand and implement; as an example, we do not need zero-knowledge proofs or secure multi-party computations for our scheme.

1.1 Possible use cases

One possible use case for this voting scheme is a group of people who are not necessarily physically close to each other and want to vote to decide the group's standing on a particular issue, without having to trust any administrator or third party to manage the voting process, while still being convinced that the election took place without any cheating.

Another example of using this scheme would be for executing polls or elections within corporations, where the voters want to be sure that their vote counts while no one will be able to link their vote to their identity.

While the basic protocol only realistically scales up to a few thousand voters, we can remedy this by creating many of these groups of voters — essentially just like having many polling places in real world elections — and then adding the intermediate tallies from these subgroups in order to get a final tally.

An example of where this scheme could be useful in a large scale, is for elections in countries where the population don't have trust in their government officials because of corruption or the like. The people voting would not have to trust anyone from the government in order for them to be convinced that the election was performed without foul play.

1.2 The decentralized deadline consensus problem

A central problem in this thesis is the *decentralized deadline consensus problem*, which is the problem of determining whether a message was sent before a certain deadline without relying on any centralized service, while ensuring that everybody arrives at the same result.

The motivation for solving this problem is for actors who want to arrive at a consensus of messages that were sent before the deadline, but who don't trust each other nor any central authority to make the decision for them.

A prime example of this is an election situation, where voters want to enforce a deadline for the election in order for it to end, and only have the timely votes to be counted towards a final tally.

The decentralized deadline consensus problem poses two interesting subproblems:

- We cannot trust any actor to correctly timestamp messages themselves, because their clock may be out of sync or they may be acting maliciously to force messages to appear as if they were sent before the deadline, even if they weren't.
- We need a canonical way to determine if a deadline has been reached or not, again without trusting any single entity.

Formal description

More formally, we want a mechanism with the following properties:

1. it must be able to determine whether a deadline has passed or not.
2. it must provide a mechanism for *timestamping* arbitrary messages, verifying these timestamps and deciding if a message was timestamped before or after a deadline.
3. it must be *canonical*, meaning that querying this service for the timestamp of a message or if a certain deadline has been reached always produces the same result regardless of who is querying.
4. it must be *decentralized*, i.e., no central authority will have absolute control over it.

A number of timestamping services already exist that are *either* canonical or decentralized, but not both:

Canonical timestamping Countless services, such as Twitter, Reddit, and many other social interaction services, allow for posting a message and timestamping it with the service's current time in the process. This produces a canonical timestamp, but isn't decentralized since we have to trust a central authority.

Decentralized timestamping The trivial example of decentralized timestamping is having each individual timestamp a message with their own time. This obviously isn't canonical because different individuals may have different clocks.

1.3 Structure of the thesis

Section 2 describes and analyzes the theory we need to solve the above problems. In Section 3 I propose a protocol to solve the decentralized deadline consensus problem, and in Section 4 we build a voting protocol on top of the proposed consensus protocol. Section 5 describes the implementation of these two protocols, and Section 6 evaluates the protocols and their implementations.

2 Theory

The following sections will describe the necessary theory used in this thesis.

The first five Sections (2.1 through 2.5) will provide a foundation for solving the decentralized deadline consensus problem in Section 3, while the last two Sections 2.6 and 2.7 will be used for building a trustless, private and decentralized peer-to-peer voting scheme in Section 4:

Section 2.1 explains the mathematical basis and advantages of elliptic curve cryptography. The concept of Proof-of-work (PoW) is explained in Section 2.2. Section 2.3 examines the cryptocurrency Bitcoin and its associated blockchain technology. In Section 2.4 we look at Bitmessage, a peer-to-peer protocol supporting encrypted and anonymous person-to-person and broadcast communication. Lastly, in Section 2.5 we look into Invertible Bloom Lookup Tables, a relatively new (2011) data structure for reconciliating sets between different actors, using very little space.

Section 2.6 describes basic theory about voting schemes and their properties, and in Section 2.7 we examine a special type of cryptographic signature, the linkable ring signature, which we will use to distinguish votes from registered voters.

2.1 Elliptic curve cryptography

Elliptic curve cryptography (ECC) is gaining ground in the world of cryptography, being used increasingly instead of well-known and widely-used schemes as RSA and Diffie-Hellman (DH). ECC offers improvements over RSA and DH with smaller key sizes and more efficient implementations while providing the same level of security [12].

The key sizes for elliptic curves range from being 6 to almost 30 times smaller than RSA/DH keys providing the same security, as shown in the following Table 1. For example, to provide the same level of security as a 3,072 bit RSA/DH key, only a 256 bit key is needed for ECC, which is a 12 times smaller key and computes roughly 10 times faster:

ECC key size	**RSA/DH key size**	**RSA:ECC key size ratio**	**RSA:ECC computation ratio**
160 bits	1024 bits	6.4	3
224 bits	2048 bits	9.1	6
256 bits	3072 bits	12.0	10
384 bits	7680 bits	20.0	32
521 bits	15360 bits	29.5	64

Table 1: Key sizes in RSA and ECC providing the same level of security along with the ratio in key size and computational cost [1]

It is interesting to note that the security of the two "first generation" asymmetric encryption schemes, RSA and Diffie-Hellman, rely on two different, but closely related, problems; factoring the product of two large primes and the discrete logarithm for finite groups, respectively.

On the other hand, the "next generation" scheme, elliptic curve cryptography, is based on the mathematical properties of certain well-chosen elliptic curves over finite fields, where it is assumed to be infeasible to find the discrete logarithm of a given element on the curve given a publicly known base point — the *elliptic curve discrete logarithm problem* or ECDLP, which is similar to the discrete logarithm problem in finite groups.

The reason why RSA/DH keys must be larger than ECC keys to provide the same level of security, is that RSA and Diffie-Hellman have slowly but steadily been compromised by stronger and stronger attacks, while ECC has had no known compromising attacks since it was introduced in 1985 [1].

2.1.1 What is an elliptic curve?

Traditionally, an elliptic curve E has been defined by a type of equation known as a *Weierstrass equation* which is of the form

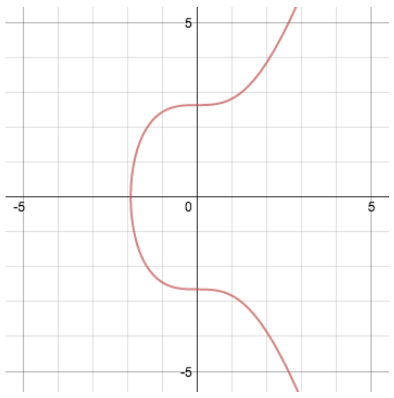
$$E:\ y^2 = x^3 + ax + b, \quad a, b \in \mathbb{R} \tag{1}$$

For the curve to usable, it has to be non-singular[1], which it is iff a and b satisfy the following equation:

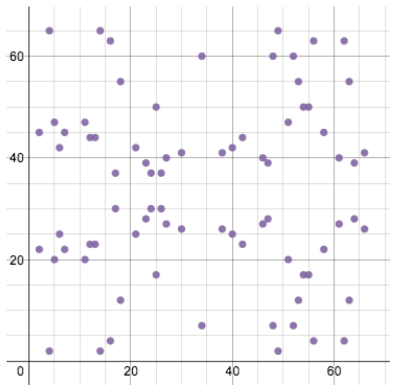
$$-16(4a^3 + 27b^2) \neq 0$$

In addition to the Weierstrass equation, other equation types — in particular the *Montgomery equation*[2] and the *Edwards equation*[3] — have been introduced, promising more efficient algorithms for performing mathematical operations on points on these curves [15].

Figure 1 shows a typical elliptic curve over $\mathbb{R}$ and a finite field. One can see that the points on the curve over $\mathbb{R}$ are continuous and smooth, but in a finite field the points are scattered across the entire field. Note how all points in both curves are mirrored around a horizontal line in the center[4].



(a) Elliptic curve over $\mathbb{R}$



(b) Elliptic curve over a finite field

Figure 1: A typical elliptic curve over $\mathbb{R}$ and a finite field. Images used with permission from chain.com[5]

[1]A singular elliptic curve has a special point in which the tangent is not regularly defined, and we need the tangent to be properly defined in all points on the curve.

[2]$by^2 = x^3 + ax^2 + x$, where $b(a^2 - 4) \neq 0$ in $\mathbb{F}_p$

[3]$x^2 + y^2 = 1 + dx^2y^2$, where $d(1 - d) \neq 0$ in $\mathbb{F}_p$

[4]Curiously, when storing elliptic curve points in a "compressed form", only the x-coordinate is used along with a single bit that determines which of the two points with that x-coordinate to use.

[5]`http://blog.chain.com/post/95218566791/the-math-behind-bitcoin`, visited 2014-10-17

In cryptography related applications, elliptic curves are usually[6] used over a prime field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$. For the remainder of this section on elliptic curves, I will only focus on prime fields as the underlying field. The size of this field defines the difficulty of the ECDLP.

Besides the already mentioned parameters — a, b and p — a curve consists of a *base point* G, an order n and cofactor h:

The base point (or *generator*) $G = (x_G, y_G)$ defines the cyclic subgroup of order n in which addition and multiplication (doubling) of points on the curve takes place. Finally, the cofactor $h = |\mathbb{F}_p|/n$ is normally 1 for curves over prime fields [28].

Care must be taken when selecting the domain parameters (a, b, p, G, n, h) for an elliptic curve in order for the curve to be considered secure. Several standard curve parameters believed to be secure have been proposed by SEC[7] [28] and SafeCurves [31].

2.1.2 Geometrical addition of points

The elements E in the group of points in an elliptic curve are

$$E : \{(x, y) \mid y^2 = x^3 + ax + b\} \cup \mathbf{0}$$

... where $\mathbf{0}$ is a special *point at infinity*, which also serves as the identity element with respect to addition (+).

The negation of a point $P = (x_P, y_P)$ is defined as negating the y-coordinate of the point: $-P = (x_P, -y_P)$. The negation of $\mathbf{0}$ is $\mathbf{0}$.

To understand the arithmetic behind elliptic curve point addition, it is useful to first look at a geometric interpretation of the operation, which is shown in Figure 2.

If $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ are two points satisfying the curve equation and $x_P \neq x_Q \Leftrightarrow P \neq Q \wedge P \neq -Q$, there is usually[9] a unique third point $-R$ where the line through P and Q intersects with the curve. Addition is defined as the negation of this unique point $P + Q = R$ as seen on Figure 2a. Figure 2c shows an addition over a finite field. Note that the line “wraps around” the edges until it hits a point.

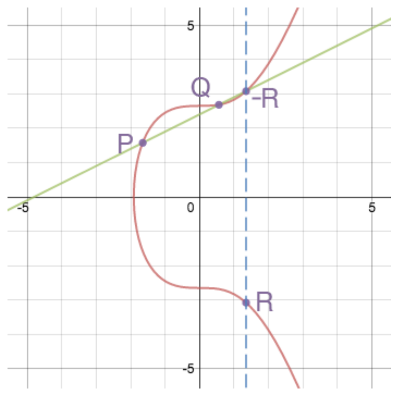
If, on the other hand $P = Q$, then we use the tangent line in the point P to find the other unique point and negate this to find $P + P = 2P = Q$ as shown in Figure 2b. This is known as *point doubling*.

[6]Sometimes, the field used is a binary field $\mathbb{F}_{2^m}$ instead of a prime field.

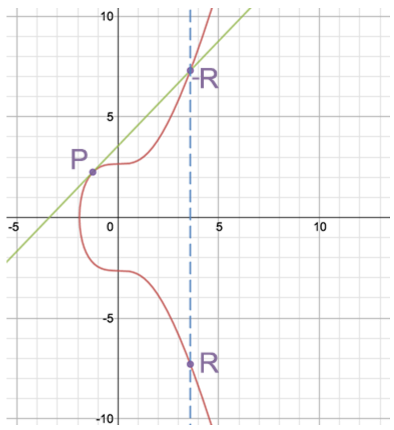
[7]Standards for Efficient Cryptography

[8]`http://blog.chain.com/post/95218566791/the-math-behind-bitcoin`, visited 2014-10-17

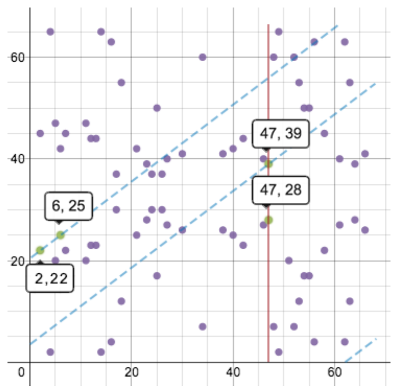
[9]If there is not a unique third point defined this way, it means that the line through the points P and Q is also the tangent for one of those points on the curve. Addition is then geometrically defined as the negation of the point in which the line is tangent.



(a) Addition of two points over $\mathbb{R}$
$P + Q = R$



(b) Doubling of a point over $\mathbb{R}$
$2P = P + P = R$



(c) Addition of two points over a finite field
$(2, 22) + (6, 25) = (47, 28)$

Figure 2: Addition and doubling of points on an elliptic curve over $\mathbb{R}$ and addition over a finite field. Images used with permission from chain.com[8]

Lastly, if $P = -Q$, the line between the two points would be completely vertical and the result of $P + -P = \mathbf{0}$.

2.1.3 Arithmetic addition of points

With a intuitive understanding of how to add and double points on an elliptic curve, we define the same thing arithmetically so we can compute the results of addition and doubling. We still differentiate between the three cases where $x_P \neq x_Q$, $P = Q$ and $P = -Q$. The equations given in this section can be found, among other places, in Standards for Efficient Cryptography (SEC) 1 [27].

Normal addition ($P + Q,\ x_P \neq x_Q$) We consider two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$, $x_P \neq x_Q$. As with the geometrical addition above, we first find the line between the two points — in fact, we only need the slope of the line

$$s = \frac{y_P - y_Q}{x_P - x_Q}$$

From here, we can calculate

$$P + Q = R = (x_R, y_R)$$
$$x_R = s^2 - x_P - x_Q$$
$$y_R = -y_P + s(x_P - x_R)$$

Doubling ($2P = P + P$) When doubling, we need to check that the Y-coordinate $y_P \neq 0$. If $y_P = 0$, the result of the doubling is $\mathbf{0}$.

Otherwise, if $y_P \neq 0$, we calculate the slope differently than normal addition (because we need the slope of the tangent in the point P):

$$s = \frac{3{x_P}^2 + a}{2y_P}$$

Once we have the tangent slope, the calculation of the resulting point is conceptually the same as in addition:

$$2P = P + P = R = (x_R, y_R)$$
$$x_R = s^2 - 2x_P$$
$$y_R = -y_P + s(x_P - x_R)$$

Adding the inverse ($P + (-P)$) In the case that we want to add a point to its inverse, the result is $\mathbf{0}$.

Multiplication with scalar values When addition of two points and doubling of a single point has been defined, we can use that definition to define multiplication of a point with an integer value. This multiplication can be decomposed into a series of additions and doublings. As an example, we'll multiply a point P with 27:

$$\begin{aligned}27P &= 26P + P \\ &= 2(13P) + P \\ &= 2(12P + P) + P \\ &= 2(2(6P) + P) + P \\ &= 2(2(2(3P)) + P) + P \\ &= 2(2(2(2P + P)) + P) + P\end{aligned}$$

This method of computing the multiplication of a point with a scalar value is commonly called *double-and-add*[10].

It is worth noting that $nG = \mathbf{0}$, where n is the order of the cyclic group. $\mathbf{0}$ being the neutral element in the group, this also implicates that $(n+1)G = nG + G = \mathbf{0} + G = G$.

2.1.4 Number of points in an elliptic curve over a finite field

Although it is computationally very hard[11] to discover the exact number of points in an elliptic curve over a finite field, a useful approximate of this value is provided by *Hasse's theorem on elliptic curves* [34] which states that

$$|N - (p + 1)| \leq 2\sqrt{p}$$

This can be interpreted to say that the number of points N is "very close" to $p + 1$ (p being the size of the underlying field). Very close meaning that N and $p + 1$ differ by at most $2\sqrt{p}$.

This next section will explain why Hasse's theorem is useful for our purpose.

2.1.5 A hash function for elliptic curves

For certain applications we need an injective function that maps an arbitrary bitstring into the point space of a curve. One might think of this function a "hash function for elliptic curves" in the sense that it is fairly easy to map the bitstring into the point space, but very difficult to reverse the direction of the function and retrieve the bitstring from the point.

[10]As a mathematical curiosity, double-and-add is conceptually very similar to the *square-and-multiply* method for computing large positive integer powers of a number.

[11]Basically, one would have to go through all points one after one and count them

Several functions of this kind exist — the *try-and-increment* method, the "Twisted" Curves and the Shallue-Woestijne Algorithm among others [22], and I will in this paper consider the try-and-increment method because of its simplicity.

The try-and-increment method As hinted at, the try-and-increment method is conceptually very simple and it maps a positive integer $i \in \mathbb{F}_p$ into the point space. It is implemented in the following way:

1. Set $x \leftarrow i$
2. Calculate $r = x^3 + ax + b$
3. If r is a quadratic residue[12] in $\mathbb{F}_p$, then return point $P = (x, \sqrt{r})$
4. Else, increment x and restart at step 2

While it isn't possible to determine exactly how many times this algorithm will loop and try again for a specific starting value of x, Hasse's theorem above tells us that the number of valid curve points over a finite field is close to the size of the field. This is important because we can interpret the result as saying that roughly one out of two x-coordinates will be valid and terminate in step 3 [22].

Another way to put this is to say that the probability of the algorithm not returning after k steps is 2^{-k}; in average the algorithm returns on its first try in every other attempt, its second try in one out of four attempts, and so on.

Hashing with the try-and-increment method Now that we have a function $\mathbb{F}_p \to E$ to turn an element of $\mathbb{F}_p$ into a point on an elliptic curve over that field, we just need a function to irreversibly turn an arbitrary bitstring into an element of the prime field, which then in turn can be transformed to a point on the curve.

A straightforward obvious way to do this is to find a cryptographic hash function with an output space (roughly) equal to the size[13] of the prime field. One could then hash the bitstring into an element of the prime field, which is then again used as input to the try-and-increment method to create an irreversible injection from an arbitrary bitstring to a point on an elliptic curve over a prime field.

[12]An integer r is called a *quadratic residue* in $\mathbb{F}_p$ if there exists an integer x such that $x^2 \equiv r \mod p$; we then define $\sqrt{r} = x$. If not, then r is called *quadratic nonresidue.* To decide whether or not r is a quadratic residue in $\mathbb{F}_p$, one could use the Tonelli-Shanks algorithm [35], which also computes the needed $\sqrt{r}$. Note that the Tonelli-Shanks algorithm only works with prime fields.

[13]Alternatively, the hash function could have an output space that is close to a multiple of the field size, and we could use the remainder from division with p as input to the try-and-increment method

2.1.6 Using elliptic curves for cryptography

With the mathematical basis in place for elliptic curves, we will look at how we can use it to provide cryptographic primitives which have lower key sizes and in turn more efficient implementations than the first generation RSA/Diffie-Hellman schemes.

The elliptic curve discrete logarithm problem (or ECDLP), which is the basis of cryptography with elliptic curves can be stated as follows:

> Let E be an elliptic curve defined over a finite field $\mathbb{F}_p$.
>
> $$E : y^2 = x^3 + ax + b \qquad a, b \in \mathbb{F}_p$$
>
> Let S and T be points in E. Find an integer m so that $T = mS$.

Note that although the problem looks like a division, $T = mS \Rightarrow m = \frac{T}{S}$, because we use additive notation, as is convention, we could just as well have used multiplicative notation, $T = S^m \Rightarrow m = \log_S(T)$ to make it look like a logarithm.

The essence of the problem is that it is easy to compute T if you have m and S, but impractical to compute m if you have S and T.

As mentioned in the beginning of Section 2.1, elliptic curve cryptography provides more efficient alternatives to already widely-used schemes, such as Diffie-Hellman key exchange (DH), Digital signature algorithm (DSA), and Diffie-Hellman Integrated encryption scheme (DHIES). The elliptic curve counterparts to these schemes are known as ECDH, ECDSA, and ECIES, respectively [27].

Public and private keys with elliptic curves Two of the schemes, ECDSA and ECIES, requires some kind of private and public keys, and for elliptic curves, these key pairs are generated in the following way:

1. Select a random integer $k \in [1, n-1]$. This integer is the private key.
2. The public key is calculated as $K = kG$.

It is important to note that although the private key k is an integer just as with, e.g., RSA keys, the public key K is not an integer but instead a point on the curve.

When someone receives a public key $K = (x_K, y_K)$, they can check that it is a valid public key point by checking the following three things as described by Antipa et al. [2]:

- Check that $K \neq \mathbf{0}$ and that $x_K \in [1, n-1]$ and $y_K \in [1, n-1]$.

- Check that K lies on the curve, i.e., satisfies the curve equation.
- Check that $nK = \mathbf{0}$, where n is the order of the cyclic group.

The first two checks ensure that the point is on the curve, while the third check serves to ensure that the point isn't in a small subgroup of the curve [2].

2.2 Proof-of-work

A proof-of-work is a solution to a mathematical puzzle that on average takes a certain amount of resources to compute. The concept of proof-of-work originated with Hashcash by Adam Back in 1997 [4].

Hashcash is a system originally proposed as a way to combat email spam, and introduces the idea of CPU cost-functions, which are parameterisably expensive to compute, but relatively cheap to verify.

The idea is that a client who wants to send an email would have to spend CPU resources to complete a challenge from an email server in order to identify the client as an honest email-sender. The problem for email-spammers is then that they have to spend CPU resources for every email they want to send, and this will essentially make mass-emailing a very expensive task.

In particular, the PoW schemes discussed in this thesis are known as *partial hash inversions*m which build on a security property of cryptographic hash functions; they are designed to be substantially easier to compute than to invert: Given a hash function H, it is easy to compute $y = H(x)$, but it is very hard to find x given only the hash y.

The partial hash inversion basically works like this:

1. You need to have a message m that you want to compute a proof-of-work for and the required *target* value t for the proof-of-work.
2. Choose a candidate integer[14] i
3. Compute the hash of m concatenated with i: $h = H(m \mid\mid i)$
4. If $h < t$, (h, i) is the proof-of-work for the message m.
   If $h \geq t$, repeat from step 2.

The difficulty of the challenge can be adjusted to increase or decrease the average time required to solve it by making the target value t lower or higher. Specifically, the challenge is to find a nonce that, when hashed along with the payload yields a bit string that starts with a certain number of zeros. Adjusting the number of zeros required adjusts the difficulty by making it twice as hard for each additional zero bit.

Both Bitcoin and Bitmessage[15], which we examine in the following sections, use a proof-of-work scheme. The former to establish consensus and the latter to protect the network against flooding, respectively.

[14] In practice, this integer is usually selected by starting with 0, then 1, and so on.

[15] When doing the proof-of-work, Bitmessage computes the hash h as $h = H(i \mid\mid H(m))$, `https://bitmessage.org/wiki/Proof_of_work`, visited 2014-11-20

2.3 Bitcoin and blockchains

Bitcoin is a so called *cryptocurrency*, the first of its kind that has received widespread adoption, and the underlying technology, the *blockchain*, which is a central part of Bitcoin, has presented a way for many entities to agree on a single truth.

In the Bitcoin protocol, clients connect and communicate with each other without any kind of central repository or administrator. Instead, the clients collectively decide on what to record in the blockchain, a public ledger of all accepted transactions with the unit of currency, which is also called bitcoin[16]. The blockchain is publicly visible to everyone interested, and contains a complete record of all transactions since the first release of Bitcoin in 2009.

The original Bitcoin white paper [25] was published in 2008 by "Satoshi Nakamoto", a pseudonym for a person or a group of people; the true identity of Nakamoto is unknown, although many attempts to uncover it have been made [21].

The double-spending problem

Bitcoin and the blockchain is also a practical solution to the *double-spending problem* [25], which is the problem of ensuring that an individual cannot duplicate his money and use it as payment more than once. Historically, this task has been undertaken by a central authority such as a clearing house, but with Bitcoin this responsibility has been taken away from any single entity and instead distributed among the clients in the Bitcoin network, who are able to collectively reach a consensus about whether or not transactions are valid and should be included in the blockchain.

The details about how this problem is solved is described in the following sections.

Bitcoin addresses and keys

In Bitcoin, value is transferred between *addresses*, a combination of 26-34 alphanumeric characters starting with 1 or 3[17], which look like this: `13u9opV1XxoTtvpjuhjw32jmkr63UwpH6F`. Anyone can free-of-charge create as many addresses as they desire.

An address is a Base58Check[18] encoded 160-bit hash of the public key of an elliptic curve keypair, meaning that an address is essentially defined by its private key. The private and public keys in Bitcoin are standard elliptic curve keys as described in Section 2.1.6, where the private key is an integer

[16]Note that Bitcoin with a capital B refers to the Bitcoin protocol and the Bitcoin network, while bitcoin with a lowercase b refers to the currency unit.

[17]`https://en.bitcoin.it/wiki/Address`, visited 2014-10-16

[18]`https://en.bitcoin.it/wiki/Base58Check_encoding`, visited 2014-11-10, lists the reasons for using a custom encoding instead of standard base-64 encoding.

$k \in [1, p-1]$ and the public key is a point $K = kG$. The constants p and G used by Bitcoin are described in the following section.

The exact recipe for computing the Bitcoin address corresponding to a specific EC keypair can be found on the Bitcoin wiki[19].

Elliptic curve cryptography in Bitcoin

Bitcoin uses a specific elliptic curve called `secp256k1`, which has been noted as not being completely safe[20] by `http://safecurves.cr.yp.to`. It is important to note that this doesn't mean that this specific curve has been compromised, but instead that the curve has some properties that could eventually pose a security risk for everyone that uses it as a base for their cryptography.

The domain parameters (p, a, b, G, n, h) for the `secp256k1` curve are specified in SEC's Recommended Elliptic Curve Domain Parameters [28] and are as follows[21]:

$$\begin{aligned} p &= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1 \\ a &= 0 \\ b &= 7 \\ n &\approx 2^{256} \\ h &= 1 \end{aligned}$$

2.3.1 Transactions

One of the fundamental building blocks in the Bitcoin protocol is that of the *transaction*[22], or *tx* for short. A transaction is a transfer of bitcoins between two or more Bitcoin addresses. A transaction must be signed with the private key of the address that sends the bitcoins. This essentially ensures that only the holder of a private key can authorize transfers of bitcoins from the address associated with that private key, and, as such, it is infeasible to forge transactions if one does not know the appropriate private key.

Transactions are identified by their *transaction hash*, which is computed as a double SHA-256 hash[23].

A transaction consists of a number of *inputs* and a number of *outputs*:

[19]`https://en.bitcoin.it/wiki/Technical_background_of_Bitcoin_addresses`, visited 2014-10-16

[20]Not completely safe means that the curve may produce incorrect results for some rare curve points and/or occasionally leak secret data.

[21]The generator point G is not included here for brevity.

[22]`https://en.bitcoin.it/wiki/Transaction`, visited 2014-10-16

[23]To clarify, in order to compute the hash of a transaction, one has to first compute the SHA-256 hash of the transaction, and then compute the SHA-256 of that intermediary hash.

Input An input is comprised of a reference to a previous output and a *script* that proves that the transaction creator has knowledge of the private key yielding the address that the previous output refers to, the *source address*.

In particular, the reference to the previous output consists of the hash of the transaction that the output is contained in, as well as the output *index*, i.e., the 0-indexed output number inside the referenced transaction.

Output An output is the combination of another script that defines the *destination address*, i.e., the address to which the bitcoins are to be sent, and a *value* that specifies how many bitcoins to send to that address. The value is denominated in *satoshis*, the smallest possible unit of bitcoins, where 1 bitcoin = 100,000,000 satoshis.

If the total value of all outputs exceed the value of the referenced inputs, the transaction will be rejected as invalid. If, on the other hand the output total value is less the the input values, the excess value will be regarded as a *transaction* fee to the *miner* that includes the transaction in the blockchain. Miners will be explained later.

Transaction scripts Both inputs and outputs contain scripts[24] that serve different purposes. As mentioned, output scripts define the address that will receive the value, and input scripts serve as proofs that the transaction is approved by someone who has knowledge of the private keys of the addresses which contains the value to send. The input scripts are called `scriptSig`'s and output scripts `scriptPubKey`'s.

Although the scripts are usually used as described here, they can allow for more complex behaviours, such as requiring more than one private key, or a combination of several private keys to "unlock" the value contained in the transaction. These scripts can be used to form *smart contracts* and the like[25]

Address balance It is important to point out that the current "balance" of any Bitcoin address, i.e., the amount of bitcoins available for spending by the holder of the private key of the address, isn't an inherent property of the address itself. Instead, the current balance is computed on the basis of *unspent* transactions to that address, i.e., transactions with outputs that have not been used as inputs for other transactions.

Publishing transactions When someone has created a valid transaction and wants to include it into the blockchain, they broadcast the transaction

[24]A comprehensive documentation of the transaction scripts can be found at `https://en.bitcoin.it/wiki/Script`, visited 2014-10-16

[25]`https://en.bitcoin.it/wiki/Contracts`, visited 2014-11-17

to the their connected peers, which will in turn propagate the transaction to every peer on the network. Note that whoever broadcasts it is irrelevant, because the transaction is self-contained.

This means that you can create the transaction on a computer that holds the private keys and is not connected to the internet, and then broadcast that transaction from a completely different computer, if you are concerned about the security of your keys and don't want to potentially expose them to the dangers of the internet.

Dust In order to keep the network from being flooded with really small transactions (e.g., just sending 1 satoshi), the network considers small outputs to be *dust* and discourages transactions that only have dust as outputs by either requiring a transaction fee or simply not including the transaction.

In the Bitcoin 0.8.2 update[26], dust was defined as transaction outputs less than 5430 satoshi (0.0000543 BTC), and at the same time the default transaction fee was set to 10000 satoshi (0.0001 BTC). This means that the smallest possible transaction requires 15430 satoshi (0.0001543 BTC, USD 0.06 at this writing[27]).

2.3.2 Blocks

When transactions are broadcast to the network, they are collected into *blocks*, which are then placed on the blockchain. Each block contains a reference to the previous block, so they together form a chain of blocks which can be traced all the way back to the very first block, the *genesis block*[28], mined by Satoshi Nakamoto.

Blocks are essentially collections of recent transactions from when the block was *mined* and put on the blockchain. A block also contains a proof-of-work:

Proof-of-work (PoW) Just as in Section 2.2, the proof-of-work is to find an integer such that the block concatenated with the integer yields a sufficiently small number d when hashed[29]. Note that since each block must contain the hash of the previous block, the solution to the PoW cannot be computed before that block exists.

[26]`https://github.com/bitcoin/bitcoin/blob/master/doc/release-notes/release-notes-0.8.2.md`, visited 2014-10-25

[27]1 BTC could be exchanged for USD 358.20 on 2014-11-10

[28]The genesis block was created on January 3rd 2009, `https://en.bitcoin.it/wiki/Genesis_block`, visited 2014-10-16

[29]Just as with the transactions, the PoW uses double SHA-256 as its hashing mechanism. The reason for using double SHA-256 instead of a single round has been conjectured (`http://crypto.stackexchange.com/a/884/12371`, visited 2014-11-17) to be to make SHA-256 invulnerable to a *length extension* attack [19]

The number d defines the difficulty of the problem; if the hashing function has an output space of size 2^{256}, then a difficulty value of $d = 2^{255}$ would mean that a hash would be below d approximately half of the time, and with $d = 2^{254}$ a quarter of the time, and so on. If we also know how many hashes a single computer, or conversely the entire network, can compute in a given amount of time, we can calculate a difficulty value that makes the PoW take a given time *on average.* In Bitcoin, this average time is set to be 10 minutes.

As the network's hashing power inevitably fluctuates with clients connecting and disconnecting, as well as the increase in the performance of hardware over time, the difficulty is adjusted every 2 weeks[30] to ensure that the average block time remains close to the desired 10 minutes.

Mining When a client on the network has found a valid solution to the PoW, he broadcasts the PoW and the corresponding block to everyone else, and, if the other clients can verify the PoW as well as the transactions in the block, that block is added to the top of the blockchain, and clients begin a new block with new transactions on top of the newly added block.

This process of computing a PoW for a block is called *mining*[31] and the clients who participate in it are called *miners.*

Coinbase transaction When miners assemble a new block to begin mining on, they add a special transaction as the first transaction in the block, called the *coinbase* transaction. This transaction serves two purposes; to issue new bitcoins into existence and to collect transaction fees as described in the previous section on transaction outputs.

The miner is allowed (and expected) to transfer these newly issued coins and transaction fees to their own address, thus creating an incentive to use CPU time and electricity; they are paid in bitcoin for their efforts.

Bitcoins are issued at a predictable rate such that the supply of bitcoins never exceeds 21 million[32]: The coinbase transaction of the first 210,000 blocks issue 50 bitcoin to the currency supply, the next 210,000 blocks issue 25 bitcoins, and so on, halving the amount of new bitcoins for every 210,000 blocks. With the 10 minute average block time, this means that the issuance is halved approximately every 4 years[33].

[30]Technically, the difficulty is adjusted every 2,016 blocks, which is equal to 2 weeks if every block takes an average of 10 minutes.

[31]"The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation. In our case, it is CPU time and electricity that is expended." [25]

[32]The total amount of issued bitcoins can be calculated as $\sum_{i=0}^{\infty} 210,000 \cdot 50 \cdot 2^{-i}$

[33]If we assume that every year contains 365.25 days, four years equate to $4 \cdot 365.25 \cdot 24 \cdot 6 = 210,384$ blocks.

Note that these constants — 10 minute average block time, 2,016 blocks before adjusting difficulty, multiplying the issuance with 0.5 every 210,000 blocks, and 50 bitcoins as the start issuance — are properties of the Bitcoin protocol and not inherent to the blockchain technology; other coins with their own blockchain exist with completely different constants. As an example, Dogecoin, a less popular cryptocurrency, has an average block time of 1 minute, adjusts the difficulty after every block, and continues to issue 5,200,000,000 units of currency every year forever.

Blockchain branches Inevitably, two miners come up with valid blocks with different solution to the PoW, and possibly different transactions. Broadcasting these blocks will result in a *branching* of the blockchain, meaning that some miners will begin mining on top of block A whereas others will mine on top of block B. All miners will however remember both blocks in case they want to switch to the other branch.

In these cases, there is no immediate consensus on which block to use, and the network will have to wait until another block is mined, either A' on top of block A or B' on top of block B. Whichever of the A' or B' blocks are mined first is then accepted by the network to be the "correct" branch, and the blocks in the losing fork are then called *orphan* blocks. Of course, if both A' and B' are mined at around the same time, a consensus still haven't been reached and we must wait for either A'' or B''.

Branches in the blockchain don't happen often, and when they do they usually just result in a single orphaned block on the then-discarded branch. However, it has happened before that branches with up to four separate blocks have been orphaned[34].

Confirmations When we have seen that up to four blocks can be discarded in favor of another blockchain branch, we have to be aware of the opportunity of double-spending in two concurrent branches; say that Alice transfers 1 bitcoin to Bob on branch A and transfers the same bitcoin to Charlie on branch B. If Bob watches branch A and Charlie branch B, they would each think that the transactions transferring money to themselves were valid, although a conflicting transaction was on the other branch.

This risk is usually mitigated by requiring transactions to have a certain number of *confirmations* before it is considered completed; if the transaction in question is included in the latest block, it is said to have 1 confirmation. If the transaction is included in the next-latest block, it is said to have 2 confirmations, and so on.

The defacto standard for Bitcoin merchants is to only consider a transaction as confirmed when it has 6 confirmations[35], but depending on the

[34]`http://bitcoin.stackexchange.com/a/4638`, visited 2014-10-17
[35]`https://en.bitcoin.it/wiki/Confirmation`, visited 2014-10-17

purpose of the transaction it could be set higher or lower, e.g., if the transaction is a very small amount and it is more important that the transaction is approved quickly, for example paying for a cup of coffee, the merchant could deem it acceptable to just have seen the transaction broadcast[36] to the network in order to accept it quickly.

The more confirmations a merchant waits for before accepting a transaction as completed, decides how vulnerable the merchant is to a double-spending attack; if the transaction in question is included in block n and they wait for the standard 6 confirmations, an attacker would have to branch out from block $n - 1$ and mine his own blocks without that transaction in them. He would then have to mine blocks n through $n + 5$ faster than the rest of the network. This attack is sometimes called a 51% attack because in order to succeed, the attacker is required to control more hashing power than the rest of the honest network.

Block timestamps Each mined block contains the miner's own timestamp at roughly the time when the block was mined. Given that different miners inevitably will have differing clocks, each client validating a block needs a mechanism for also validating the block's timestamp.

The Bitcoin Wiki on Block timestamps[37] defines the following rule to validate timestamps from other peers:

> A timestamp is accepted as valid if it is greater than the median timestamp of previous 11 blocks, and less than the network-adjusted time + 2 hours. "Network-adjusted time" is the median of the timestamps returned by all nodes connected to you.

In other words, this means that for any block n, the median timestamp of the previous 11 blocks will always be greater than the corresponding median for block $n - 1$ and less than for block $n + 1$. Thus, the median timestamp is always increasing and while the timestamps of the individual blocks can fluctuate somewhat, the median timestamp can be viewed as a pretty stable timestamping service.

[36]The merchant may decide the transaction may not even have to be confirmed inside a block, but its mere presence on the network, without any conflicting transactions, is enough to declare it valid.

[37]`https://en.bitcoin.it/wiki/Block_timestamp`, visited 2014-10-17

2.4 Bitmessage

Bitmessage is a peer-to-peer messaging protocol and network with a high level of privacy. It uses elliptic curve cryptography at its base and also hides message metadata like the sender and receiver of messages from network eavesdroppers.

It shares many similarities with Bitcoin: The “Bit” prefix, many aspects of the protocol are alike, the same elliptic curve is used, Bitmessage and Bitcoin keys are compatible, and more.

2.4.1 High-level overview of Bitmessage

From an external point of view, Bitmessage can essentially be viewed as an additional network transport layer that provides the following properties:

Pseudonymous messaging In Bitmessage, anybody can create a virtually unlimited number of *addresses*, or *identities*, which are used as source and destination for private messages between the owners of those addresses.

Decentralized, trustless messaging Bitmessage is a completely decentralized and trustless peer-to-peer messaging system, meaning that there are no central authorities or bottlenecks that the message data has to pass through. Thus, the users of the system don’t have to inherently trust any particular actor in the system, but only trust the system.

End-to-end encryption Every message sent is encrypted by the sender in such a way that only the intended receiver(s) will be able to open it. No one else will be able to decrypt it or even determine the sender and recipient.

State-of-the-art cryptography Bitmessage uses elliptic curve cryptography, which is considered among the strongest and most efficient asymmetric cryptosystems available today.

Broadcasting functionality Bitmessage allows for broadcasting messages to selected groups of people through two different forms of mailing list functionality. Just as with user-to-user messages, broadcast messages are end-to-end encrypted and it is infeasible for anyone who doesn’t know about the mailing list to determine the sender and receivers of a broadcast message. The mailing list functionalities are discussed more in depth in the following sections.

Flood protection Every message sent must compute a proof-of-work prior to sending to prove to the other nodes in the network that a not-insignificant amount of time and computer resources has been spent on this message, diminishing the economic incentive to send spam mail and flood the network.

The above properties mean that Bitmessage provides us with a trustless, decentralized, end-to-end encrypted message broadcasting service. We will use exactly this as a base for our implementation.

Note, however, that Bitmessage doesn't provide full anonymity, but rather only pseudonymity; if one can determine the true identity behind a Bitmessage address, the user behind the address is no longer anonymous. However, as we shall see shortly, Bitmessage provides a mailing list functionality that allows a sender to hide his own identity and identify simply as someone who knows of the mailing list.

The basic idea driving Bitmessage

Bitmessage is realized as a peer-to-peer network, where all nodes are equal and connected to a number of other nodes.

When someone wants to send a message, they first encrypt it[38] with the receiver's public key, and then advertise it to all their connected nodes, who will request the message. The nodes who have received the message will now advertise it to their connected nodes, who will also request the message (given that they don't already have it), and so on until the message has propagated through the entire network.

Once a client has received a new message, it tries to decode it with its known private keys; if decoding is successful, the client knows that the message was intended for them. On the other hand, if the message couldn't be decoded by any of the known private keys, the client can assume that the message wasn't bound for them.

In short, the idea is then that everybody gets every message and in trying to decode them, decides whether or not the message is bound for them or not. This method could arguably be described as a naïve form of Private Information Retrieval (PIR), where every user gets all data and then must decide if they need it.

This inevitably leads to a discussion of the scalability of Bitmessage; given that all clients need to read all messages passing through the network, at some point there will be enough messages to keep clients constantly trying to decode the incoming messages. The issue of scalability has been thought into the protocol from the beginning, and is remedied by the concept of *streams*, which is discussed in more detail later in this section.

In this system, where every node has to store every message with only a finite amount of memory available, a message expiry interval has been fixed

[38]As we will see later, even non-content meta-data is encrypted. The only data that isn't encrypted is the user's time for creating the message, the proof-of-work and the receiver's *stream number*. More on this later.

at $2\frac{1}{2}$ days, meaning that nodes only store messages for a few days before deleting them. This means that to be sure to receive all messages bound for you, you would have to connect to the network at least every other day to receive and try and decode all new messages before they expire.

The protocol also supports *acknowledgments*, or *receipts*, which allows for a sender to know that a sent message has been received. In Bitmessage, acknowledgments are hidden inside the encrypted payload of normal messages, and a receiver can, besides directly sending it to the sender, also use a third party to relay the acknowledgment. If a sender never receives an acknowledgment for a sent message, their client can choose to resend the message to the receiver, hoping that it will be delivered this time [37].

2.4.2 ECC keys and addresses

Bitmessage uses the exact same elliptic curve as Bitcoin (`secp256k1`).

Each Bitmessage client can create and own a virtually unlimited amount of addresses that are created on-the-fly as needed by the client.

In order to create an identity, a user needs two different private keys, which in turn can be used to compute two different public keys. One of these keys are used for encrypting data, and the other is used for signing data. These keys will be denoted the *encryption key* and the *signing key* for the rest of this paper.

A Bitmessage address starts with a `BM-` prefix which is followed by 32-34 characters in the Base58Check encoding[39] [5], and typically looks like this:

`BM-GtovgYdgs7qXPkoYaRgrLFuFKz1SFpsw`[40]

An address is a hash of a public key encoded along with a version and stream number. It is worth noting that given a public key and a version and stream number, you can compute the address for that public key[41], but you cannot go back and compute the corresponding public key from a given address.

This effectively means that in order to send a message to an address, the sender must first know the public key used to calculate the address. The sender can request this public key by broadcasting a `getpubkey` object to the network, expecting it to reach the client who controls the address, who in turn will reply with a `pubkey` object, which contains the public key along with some additional data. These two and more object types are examined in detail later in this section.

The Bitmessage client supports generating addresses in two ways:

[39] The base58 alphabet used in Bitmessage is similar to that of Bitcoin, but "many existing libraries do not use this ordering": `https://bitmessage.org/wiki/Public_key_to_bitmessage_address`

[40] This is the address of a centralized mailing list related to *Bitmessage System news/announcements* (`https://bitmessage.org/forum/index.php?topic=1689.0`)

[41] A brief description of the process can be found here: `https://bitmessage.org/wiki/Public_key_to_bitmessage_address` (visited 2014-08-22)

Deterministic address With a deterministic address, the user provides a passphrase which is then used to derive the private keys. Using the same passphrase yields the same address, which is useful for easy restoration of lost private keys, but one must take care to specify a unique passphrase; using a common word can lead to others generating the same address.

Random address With a random address, the private keys are generated from a randomly chosen number. It is not possible to regenerate the address without knowing this number. Random addresses are usually used for creating unique addresses [5]. When generating a random address, the Bitmessage client automatically broadcasts a `pubkey` object to the network.

Bitmessage uses ECIES to encrypt message payloads, which in turn uses ECDH to generate the encryption parameters for AES-256-CBC[42]. In addition to this, Bitmessage uses SHA512 as Key Derivation Function (KDF) and HMACSHA256 as Message Authentication Code (MAC) scheme [7].

2.4.3 Broadcasts/Mailing lists

In Bitmessage, two types of mailing lists exist; centralized and decentralized mailing lists. The following paragraphs describe the two types from a high-level perspective and then summarizes their key differences.

Centralized mailing list (CML) The centralized mailing list (CML) or *pseudo mailing list* is not really a mailing list internally. Instead it is a normal Bitmessage address that has been configured in a client to broadcast all messages it receives [8].

When a user wants to send a message to everyone on the CML, he or she sends a message to the CML address, which is received by the client administering that address, and the client then broadcasts the message to the subscribers of the mailing list.

This requires the client who operates the CML to be online in order to broadcast the messages, thus creating a single point of failure.

The private and public keys for decrypting and encrypting messages sent to the CML is derived from the CML address, which means that everyone who knows the address can both send messages to and receive messages from the CML.

Also, since the client operating the CML is responsible for re-sending the messages it receives, the client will have to perform a new proof-of-work for every message it broadcasts. This allows an attacker to flood the client with messages, which will slow down or even stop the flow of genuine messages.

[42]AES (Advanced Encryption Standard) with a 256 bit key that uses Cipher-Block Chaining.

Decentralized mailing list (DML) A decentralized mailing list (DML) or *chan*, on the other hand, does not require a central authority (client) as the centralized one does, and subscribers of the DML send messages directly to each other instead of using a relay as the CML does. [6]

When a user wants to send a message to everyone on the DML, he or she sends a message to the DML address, which every subscribed client then can decrypt and read.

Unlike with the CML, the private and public keys of a DML are computed from a passphrase (or chan name) in exactly the same way that the previously mentioned personal deterministic addresses are created[43]. Initially, the DML creator enters only a passphrase which is then used to generate an address for the chan. Users who wish to join the DML must know of both the passphrase as well as the generated address[44].

Also unlike the CML, the DML allows for different ways of sending messages:

DML to DML The DML is specified as both the sender and receiver of the message. “This is completely anonymous and is the default behavior. Some users however tend to block the DML address itself using the blacklist to reduce spam.” [6]

Person to DML The sender sends a message from one of his or her personal addresses to the DML address. “[U]seful if people block the DML address itself (see point above)” [6]

DML to Person The receivers are explicitly specified and only those receivers will be able to decrypt the message. “This can be used, if the user thinks the response might be useful for [specific] members of the given DML.” [6]

Table 2 summarizes the most important differences between CMLs and DMLs:

2.4.4 Scalability and streams

As mentioned earlier, Bitmessage is designed to remedy the inevitable problem of scalability arising from the fact that every client must read and try to decode all messages in order to decide if the message was bound for them or not.

[43]At the data level, personal and DML addresses are represented in exactly the same way, with the exception of a chan flag being set in DML addresses and unset in personal ones.

[44]From [6]: ”There is no technical difference between creation and joining a DML. Joining requires the address to make sure that you typed the DML name correctly and to make sure that you are using the correct stream number and address version which are included in the address and serves as an error check for the client.”

CML	**DML**
Centralized. Single point of failure.	Decentralized. No single point of failure.
Easily (centrally) moderated.	No moderation. Each user individually chooses what to read.
Spammer addresses can be blacklisted. (but spammers can generate new addresses easily)	Blacklisting spammers sending from the DML address also blacklists legitimate, anonymous messages. Spammers sending from another address can be blacklisted just like with CML.
Send messages to broadcast client who in turn broadcast them to subscribers.	Broadcast messages directly to subscribers.
Sender is pseudonymous.	Sender is anonymous when using the DML address as the sender address. Otherwise, sender is pseudonymous.
User must know CML address to send and receive messages.	User must know passphrase as well as DML address to send and receive messages.

Table 2: Important differences between centralized and decentralized mailing lists (CMLs and DMLs)

As the network and the amount of messages grow, at some point clients will start using unreasonably many resources on operating in the default mode. The creator of Bitmessage foresaw this issue, and introduced the concept of *streams* [36].

In a nutshell, once the amount of messages passing through the network reaches a certain threshold, clients begin to autonomously divide themselves into large clusters, or streams, of clients. Once divided in such streams, clients only broadcast messages to the stream which includes the intended receiver.

Two different facts make this possible:

- Encoded in a Bitmessage address is the stream number of the owner of the public key. Thus, the address dictates the stream to which messages should be directed.
- The objects sent around on the network always include an unencrypted stream number, so clients can decide whether or not to forward these objects based on if they are in the right stream or not.

As of this writing, handling of streams in Bitmessage is **not** implemented — there isn't even consensus on which algorithms to use to, e.g., subdivide the clients into streams. Although the whitepaper [36] outlines a trivial algorithm and data structure for streams, nothing has been implemented so far and is still being discussed in the Bitmessage forums[45]

As a result of this, the entire network is currently in stream number 1, and the scalability of Bitmessage is purely theoretical at this moment.

2.4.5 The Bitmessage network protocol

The Bitmessage network protocol is conceptually quite simple; it defines six *message types* and four *object types* [9, 10].

Messages Messages are used to initiate connections between peers, advertise known peers and objects, request objects, and inform about errors. All message types are sent unencrypted, and as such only objects employ encryption to hide the contents of the messages.

The six different *message types* are as follows:

`version` The `version` message is the first message exchanged when connecting to a new node, and contains information about the sender's client. Both ends of a new connection must advertise their version information before they can establish a valid link. In particular, a version message contains the following information:

[45]Among other topics, `https://bitmessage.org/forum/index.php?topic=2550.0` compares the different proposed topologies for streams.

- The version of the protocol used
- A number of flags to enable/disable features for the connection[46]
- The time when the message was generated and sent
- The network addresses of the sender and receiver of the message
- A random 64-bit integer used to detect connections to self
- The client's user agent[47]
- A list of streams that the client is interested in.

`verack` Once a client has received and accepted a `version` message from the other end of the new connection, he sends a `verack` message to acknowledge the other end's version. When both ends of the connection have received a `verack` message from the other part, the connection has been established and is ready for use.

`addr` The `addr` message informs about other known nodes on the network and is used to ensure the robustness of the network, in the sense that if a client decides that it has too few connections to other nodes, it can connect to one or more new nodes mentioned in the message. After the exchange of `verack` messages, nodes also exchange `addr` messages to help each other connect with the rest of the network.

`inv` Related to the `addr` message, the `inv` message is used to advertise known *objects* to peers. Other clients receiving this message check the list of objects against their own local list of objects to see if the peer knows of any "new" objects.

`getdata` When a client decides that one of its peers knows of one or more "new" objects, it sends a `getdata` message to the peer containing the identifier(s) of the needed object(s).

`error` This is a special error message that is used to inform other nodes about errors. It is a part of the newest protocol specification (version 3), and "may be silently ignored"[10].

Objects Objects can be considered as the real payloads being broadcast around the network. As mentioned before, peers can advertise knowledge of and request objects to and from each other. In general, clients always request advertised objects that they don't already know.

All objects, before being broadcast to the network, must include a header with a maximum time-to-live[48] (TTL) and a proof-of-work (PoW) and will

[46]As of this writing, there are no defined features to enable or disable, but 8 bytes are reserved for future uses.

[47]See `https://bitmessage.org/wiki/User_Agent` for the user agent format.

[48]The time-to-live indicates how for how long the object should be propagated around the network. The maximum time-to-live for an object is 28 days [10].

be rejected by the other peers if the PoW is not sufficient. The PoW is examined in more detail later in this section. In addition to this, all object header include a stream number[49].

The contents of the headers are always sent unencrypted. The headers are designed to leak the minimal amount of information while allowing the network to validate proof-of-work and provide support for streams.

All objects are currently restricted to a maximum size of around 256 kB [10]. There are four different types of objects:

`getpubkey` When a client wants to obtain the public key associated with a known address, it broadcasts a `getpubkey` object, and waits for the client controlling the known address to reply with a `pubkey` object. Nothing in the `getpubkey` object is encrypted.

`pubkey` The `pubkey` object contains the public signing and encryption keys as well as some short extra data for a given address and is usually[50] a response to the `getpubkey` object. As noted in the previous section on addresses and keys, the keys, address version and stream number can be combined to compute the address associated with the public key.

The `pubkey` object also contains two integers, $NonceTrialsPerByte$ and *ExtraBytes*, which are used to define the proof-of-work requirements of the public key and corresponding address. These two numbers will be examined later in the section on proof-of-work.

The public keys, $NonceTrialsPerBytes$, *ExtraBytes*, and the signature of the object are encrypted when sent over the network.

`msg` The `msg` object may arguably be considered as the most central object in Bitmessage. It is used for sending person-to-person messages as well as messages involving a DML. In a `msg` object, only the PoW nonce, timestamp and stream number are not encrypted — the rest, such as the sender's public key, the receiver's hashed public key, the actual message, an optional acknowledgment for another `msg` object and the signature, are all encrypted.

`broadcast` The `broadcast` object is quite similar to the `msg` object, but it differs in the fact that it is used for CML broadcasts instead of person-to-person or DML messages.

Note that the specification [9] of the `getpubkey` and `pubkey` objects indicate that multiple versions of the objects exist for backwards compatibility. For our purposes we will focus only on the newest version (4).

[49]The stream numbers included in object headers refer to either the sender's or the receiver's stream number, dependent on the object type

[50]When generating a random address, the client broadcasts the public key automatically

In contrast to earlier versions, version 4 encrypts most of the data in the `pubkey` objects so only someone who knows the address corresponding to that public key can decrypt it. This is made to prevent "people from gathering pubkeys sent around the network and using the data from them to create messages to be used in spam or in flooding attacks" [9].

For the curiously minded, I created a Wireshark dissector[51] for the unencrypted parts of the Bitmessage protocol, which can be found on GitHub: `https://github.com/jesperborgstrup/bitmessage-wireshark-dissector`

Message acknowledgments As hinted at, the `msg` object can carry an optional acknowledgment for a previous message, which is used to inform the sender of the previous message that this message has indeed reached the receiver.

While directly sending an acknowledgment back to the sender could be done, the protocol requires that acknowledgments are sent inside regular `msg` objects.

If, on the other hand, an attacker with capabilities to eavesdrop the traffic to and from an IP-address could send a message to Alice's Bitmessage-address, waiting for Alice's client to automatically broadcast an acknowledgment directly back to the sender of the original message, the attacker would be able to link the IP-address to Alice if it could be observed that the acknowledgment for the message to Alice originated at the specific IP-address.

The acknowledgment data packed in the message object is itself a Bitmessage protocol message, for which the PoW has already been completed. It is the responsibility of the acknowledger to prepare this acknowledgment message for delivery.

Once the acknowledgment message is completed, the user packs it in another message[52] (the outer message) and broadcasts this outer message. The recipient of the outer message must then extract the acknowledgment data (the inner message) and broadcast it to the network.

This way of packing acknowledgments inside regular messages can allow privacy-concerned individuals to have others relay their acknowledgments for them; say that Alice wants to inform Bob that she received his message. Alice prepares a `msg` object for Bob containing the acknowledgment data and does the Proof-of-Work for this message. She then takes this message, and packs it in a new `msg` object destined for Charlie, and sends it. Once Charlie receives this outer message, he discovers that the message contains acknowledgment data, extracts it from the message, and broadcasts it.

[51]Wireshark is an open-source network packet analyzer in which it is possible to write custom *dissectors*, which break down the binary contents of network packages into human-readable data segments.

[52]This message could be a "regular" message with actual content, or just an empty shell only containing the acknowledgment.

By having other clients relay your acknowledgments, the above mentioned attack is no longer possible since you can not prove that an acknowledgment actually originated from the IP-address which first broadcasts it — it could just as well be a naïve relay.

2.4.6 Proof-of-work (PoW)

As mentioned earler, Bitmessage employs a proof-of-work scheme intended to make spamming or flooding the network infeasible, or at least very expensive, for an attacker. This is done by requiring some work be done by the sender of a message to "prove" to the other clients that the message m is sent with honest intent.

The receiver's PoW target value t essentially dictates how "hard" the PoW should be, or more specifically how much time the average proof-of-work takes to calculate. The target value is calculated from two values in the receiver's `pubkey` object as well as the length L_p and time-to-live[53] TTL_p of the payload [10]:

$NTPB$ (or $NonceTrialsPerByte$) The amount of work that should be performed per byte of the payload. The network default value is $1,000$[54].

EB (or $ExtraBytes$) To make sending short messages a little more difficult, this value is added to the payload size for calculating the target value. The network default value is $1,000$[54].

$$t = \frac{2^{64}}{NTPB(L_p + EB) + \frac{TTL_P(L_p+EB)}{2^{16}}}$$

Thus, the target value t decides the difficulty of the problem; if the hashing function is good, its outputs are be distributed as uniformly as possible over the output space, and the amount of attempts spent finding a hash less than the target value is on average $2^{64}/t$

Bitmessage uses the SHA-512 algorithm as the hashing function, although 256 bits are only ever required, so the results of the SHA-512 are usually truncated to 32 bytes (256 bits).

When a client receives an object[55], they first check the PoW and only advertise and relay the object to their peers if the PoW is sufficient.

The PoW is checked by calculating the hash $h = H(i \,||\, m)$ and rejecting if $h \geq t$. Note that relaying nodes are not aware of the receiver's required tar-

[53]The time-to-live is given in seconds. `getpubkey`, `msg` and `broadcast` objects have a default time-to-live of 2.5 days, while `pubkey` objects have a default of 28 days.

[54]`https://bitmessage.org/wiki/Proof_of_work`, visited 2014-11-07

[55]All objects contain an attached proof-of-work

get value[56], so they use the default network values for $NonceTrialsPerByte$ and $ExtraBytes$.

2.4.7 Similarities with Bitcoin

Bitmessage shares a few similarities with Bitcoin, which, although they aren't critical for this thesis, are curiosities and reflect the influence of Bitcoin on the Bitmessage project [26]:

Network Both Bitcoin and Bitmessage are decentralized, trustless, peer-to-peer protocols.

Keys Since both Bitcoin and Bitmessage use the same elliptic curve domain parameters (`secp256k1`) their keys are compatible, meaning that Bitcoin keys can be used to receive send messages in Bitmessage and Bitmessage keys can be used to create transactions in the Bitcoin blockchain.

PoW As mentioned previously, the proof-of-work algorithm is essentially the same in Bitcoin as in Bitmessage. Only difference is that Bitmessage uses SHA-512 and truncates to 32 bytes, where Bitcoin uses SHA-256 which directly returns a 32-byte hash. The reason for using SHA-512 over SHA-256 is that Bitcoin uses SHA-256 everywhere, and "Bitmessage should use a different algorithm so that using Bitcoin mining hardware to do Bitmessage PoWs is at least not completely trivial." [37].

2.4.8 Possible vulnerabilities of Bitmessage

This section lists some vulnerabilities in the implementation of the reference client, PyBitmessage. Note that while there may be vulnerabilities in the implementation, it doesn't mean that the protocol is inherently flawed, but simply that the implementation of it is.

In addition to the following points, a more thorough security analysis of Bitmessage has been posted on the Bitmessage Forum[57] by user helpinghand.

Local storage of sensitive data In the current[58] implementation of the reference client, PyBitMessage, sensitive data such as keys[59] and decrypted

[56]Due to both the fact that only the receiver knows who the receiver is, and that the nodes may not even know the target value even if they knew who the receiver was

[57]`https://bitmessage.org/forum/index.php?topic=1666.0`

[58]Git commit `b02a5d31...7944f813` of November 13th 2014

[59]In Windows, private keys and information related to the keys are stored in the `%USER%/Appdata/Roaming/PyBitmessage/keys.dat` configuration file. On Linux, the file can be found at `~/.config/PyBitmessage/keys.dat`.

messages[60] are stored in plain-text on the local computer.

It would be no problem for malware and the like to sweep both keys and messages from anyone using PyBitmessage. To remedy this issue, the data could be encrypted with a password that the user enters on Bitmessage startup. This would at least make it non-trivial for malware to steal keys and messages, since the software now also would have to incorporate a keylogger to steal the password from the user before the keys and messages could be decrypted.

Using acknowledgments to deanonymize users As mentioned, the `msg` object packs optional acknowledgment data. In the current[61] implementation of the reference client, PyBitMessage, any node that receives some acknowledgment data packed inside a `msg` object obliviously broadcasts this data to the network.

This knowledge could be used to deanonymize users in much the same way that relaying of acknowledgments is supposed to prevent; say that an attacker can eavesdrop on traffic to and from a list of IP-addresses and suspects that Alice is using Bitmessage on one of those IP-addresses. The attacker knows Alice's suspected Bitmessage-address, and sends an ordinary `msg` object, also containing some acknowledgment data, to that address. If Alice is indeed using the suspected address, her client will decrypt the outer message, see the acknowledgment data packed inside it, and broadcast this acknowledgment data to the network. The attacker would then be able to see that the broadcast of the raw acknowledgment data originated at some IP-address and link the suspected Bitmessage-address to the IP-address, thus deanonymizing the user using those addresses.

I posted this issue to the Bitmessage forums[62] and was informed that this is only a flaw in the implementation, and not the protocol itself; there exists in the protocol (in the `pubkey` object) a number of flags, and one of these flags signifies that they refuse to relay ackdata. However, this only exists at the protocol level, and there is currently no way to enable or disable that flag for your addresses through the user interface.

Everyone cannot use TOR If a user connects to the Bitmessage network through the anonymization network TOR, their communications with the outside world would essentially be encrypted and nearly impossible to track.

Although the PyBitmessage client supports using TOR[63], the default

[60]In Windows, decrypted sent and received messages, etc. are stored in a SQLite database in the `%USER%/Appdata/Roaming/PyBitmessage/messages.dat` file. On Linux, the database is stored at `~/.config/PyBitmessage/messages.dat`.

[61]Git commit `b02a5d31...7944f813` of November 13th 2014

[62]`https://bitmessage.org/forum/index.php?topic=4132.0`

[63]Support for TOR is realized indirectly through a SOCKS5 proxy, which the PyBitmessage client inherently supports.

setting is to not route the traffic through it. The problem is that when you connect to the Bitmessage network through TOR, you can only create outgoing connections, not accept incoming.

If everyone on the Bitmessage network used TOR, then there would be no one accepting incoming connections to connect to, and the network would break. However, if Bitmessage and TOR would have complete support for the IPv6 protocol, everyone inside the TOR network would essentially have an externally-visible IP address to accept incoming connections, and the described problem would no longer exist[64].

[64]`http://sourceforge.net/p/bitcoin/mailman/message/29066245/` has a relevant, but short discussion of the usage of TOR with Bitcoin, visited 2014-10-17

2.5 Invertible Bloom Lookup Tables

Invertible Bloom Lookup Tables (IBLT's) are a relatively new (2011) data structure [20] based on Bloom filters [11]. IBLT's support inserting and deleting key-value pairs to and from the table, querying a key to get its value, and listing the contents of the table, all while taking up significantly less space than an ordinary list. The trade-off is that querying and listing elements are not guaranteed to return the correct answer, but are instead *probabilistic* in nature, here signifying that querying for elements can return false negatives, and listing elements may return an incomplete list of the elements inserted into the set.

Instead of using an ordinary hashmap to store n elements, which would require $O(n)$ space, an IBLT only requires $O(t)$ space, where t is the threshold value; the number of elements in the table below which querying and listing are very likely to succeed. An important limitation to IBLT's is that both keys and values need to have fixed maximum sizes, which define the size of the data structure.

2.5.1 Bloom filters

The Bloom filter [11] is a well-known probabilistic data structure that supports inserting elements into a set and then testing whether or not an element is a member of this set. The key differences between, say, an ordinary list and a Bloom filter are the space-efficiency of the Bloom Filter and the fact that false positive matches are possible, while false negatives are not. These properties are related in the sense that the space saved by using a Bloom Filter results in not being 100% accurate when querying the filter.

A Bloom filter works by using a binary array[65] T and k random hash functions $h_1, \cdots, h_k$. To insert an element into the set, one sets $T[h_i(x)] = 1$ for all $h_1 \cdots, h_k$. To query the filter for membership of an element, one checks that $T[h_i(x)] = 1$ for each of the hash functions. If at least one of the $T[h_i(x)] = 0$, we are sure that the element was never inserted into the filter, but if all $T[h_i(x)] = 1$, then the element was *probably* inserted previously, but with some risk of a false positive.

2.5.2 Introduction to IBLT

Although the Bloom filter is very useful for simple set membership checks, it would be nice if we had a similar data structure, which would allow for key-value lookups, deletion of key-value pairs and listing the pairs in the set. Luckily, the Invertible Bloom Lookup Table [20] by Goodrich et al. (2011) has just these extra properties and is originally based on Bloom filters.

[65]The binary array is initially filled with all zeroes

Their paper mentions a few use cases; database reconciliation, tracking network acknowledgments, and oblivious selection from a table. We will primarily use this data structure for the first purpose, database reconciliation. The advantage of an IBLT over an ordinary hash map or the like, is that the IBLT takes space bounded by $O(t)$, where t is so predefined *threshold* value, instead of $O(n)$ where n is the number of elements in the map. This threshold value t signifies a "divide" where the table gives results very similar to a corresponding hash map when $n \leq t$, but breaks down[66] when $n > t$.

A key feature of the IBLT is that if n has been greater than t at some point, but is later reduced to below the threshold value by removing existing key-value pairs, the table once again works as expected even though the number of pairs contained in the table temporarily was arbitrarily high. In fact, if n never exceeds t, using an IBLT would actually require more space than a simple list of key-value pairs. In this regard the value of using an IBLT is that the the size is constant no matter how many key-value pairs we insert or delete.

Throughout the remainder of this section, we will assume that all keys are distinct, i.e., that no key x is present in the table more than once at any given time. Despite this assumption, the paper [20] has suggestions for tolerating multiple inserts or deletes of the same key.

I have implemented and open-sourced a functioning IBLT in Python, which can be found on GitHub: `https://github.com/jesperborgstrup/Py-IBLT`

Database reconciliation One use of IBLTs, as mentioned, is in the area of database reconciliation. Imagine that Alice and Bob each have a database and want to discover the differences between their databases. An IBLT makes this possible while the data exchanged is much less than actually sending their entire databases to each other. As mentioned, each IBLT has a threshold value t, which is approximately the maximum number of key-value pairs that the table can hold while still being able to correctly do key-value lookups and exhaustively list the contents of the table. This means that the threshold value t should not be set to the number of entries in their respective databases, but just the expected number of differences between their databases.

Now, for Bob to deduce the differences between the databases, Alice simply has to create an IBLT with a reasonable t, fill it with the entries from her database (A), send it to Bob[67], and Bob then takes this table and deletes from it the entries from his own database (B). Once he has done this, he can request a complete listing of (1) the entries which are in Alice's

[66]Breaking down here means returning inconclusive results to queries and incomplete listings

[67]Remember that the size of an IBLT is $O(t)$

database but not in his own ($= A \setminus B$) and (2) the entries which are not in Alice's database but are in Bob's ($= B \setminus A$).

So, here we have a way of discovering differences between remote datasets that is bounded in size not by the number of entries in each sets, but by the number of expected differences between the sets, which may be orders of magnitude lower if the datasets are similar.

2.5.3 The operations of an IBLT

An Invertible Bloom Lookup Table ($\mathcal{B}$) supports the following operations:

INSERT($\mathcal{B}, x, y$) Inserts the key-value pair (x, y) into the table $\mathcal{B}$. This operation always succeeds.

DELETE($\mathcal{B}, x, y$) Removes the key-value pair (x, y) from the table $\mathcal{B}$. This operation always succeeds. If $(x, y) \notin \mathcal{B}$, the pair is stored in the table as a "negative insert", which can be detected and retrieved later on.

GET($\mathcal{B}, x$) Returns the value y if there exists a key-value pair $(x, y) \in \mathcal{B}$. The operation can give the following results:

- $(NoMatch)$: It is certain that no key-value pair (x, y) exists in $\mathcal{B}$.
- $(Match, y)$: The key-value pair (x, y) was in $\mathcal{B}$.
- $(DeletedMatch, y)$: The key-value pair (x, y) was deleted from $\mathcal{B}$ without being inserted. Thus, this is a "negative" match.
- $(Inconclusive)$: It was not possible to determine with certainty if the key x existed in $\mathcal{B}$.

LISTENTRIES($\mathcal{B}$) Returns a list of all the key-value pairs in $\mathcal{B}$. This operation returns one of two results:

- $(Complete, [(x_0, y_0), \cdots, (x_n, y_n)])$ All n key-value pairs in $\mathcal{B}$ were retrieved.
- $(Incomplete, [(x_0, y_0), \cdots, (x_r, y_r)])$ Only r key-value pairs were retrievable from $\mathcal{B}$.

Depending on the implementation of this operation, the resulting key-value pairs could also be divided into two lists — one with inserted entries, and one with entries that were deleted without being inserted.

The *Inconclusive* result of the GET operation and *Incomplete* from LISTENTRIES usually stems from the number of key-value pairs n in $\mathcal{B}$ being higher than the threshold value t. It could also happen if the same key was inserted multiple times.

2.5.4 The inner workings of the IBLT

When an IBLT $\mathcal{B}$ is created, it initializes a lookup table T with m cells. Each cell stores a constant number of fields. This number is determined from the desired maximum length of keys and values in the table. Almost as with the ordinary Bloom filter, each key-value pair (x, y) is stored in the cells $T[h_1(x)], \cdots, T[h_k(x)]$, where k is the number of hash functions used and h_i is the i^{th} hash function.

In the simple version of the IBLT, each cell contains three fields:

- A `count` field, which counts the number of entries that have been mapped to this cell[68],
- a `keySum` field, which is the sum of all keys mapped to this cell, and
- a `valueSum` field, which is the sum of all values mapped to this cell,

With these fields, the two opposite Insert and Delete operations are defined as follows:

```
function INSERT($\mathcal{B}, x, y$)
    $T \leftarrow \mathcal{B}.T$
    for each distinct $h_i(x)$, for $i = 1, \cdots, k$ do
        add 1 to $T[h_i(x)]$.count
        add $x$ to $T[h_i(x)]$.keySum
        add $y$ to $T[h_i(x)]$.valueSum
    end for
end function

function DELETE($\mathcal{B}, x, y$)
    $T \leftarrow \mathcal{B}.T$
    for each distinct $h_i(x)$, for $i = 1, \cdots, k$ do
        subtract 1 from $T[h_i(x)]$.count
        subtract $x$ from $T[h_i(x)]$.keySum
        subtract $y$ from $T[h_i(x)]$.valueSum
    end for
end function
```

Now that we have seen how to insert and remove to and from the table, let's look at how we can retrieve data:

[68]Theoretically, if we don't provide an bound for this value, the size of the IBLT is actually $O(t \cdot \log\log n)$ instead of $O(t)$. However, in practice we can usually assign a constant 4 or possibly 8 bytes for this field, allowing counts of up to $2^{31} - 1$ or $2^{63} - 1$, respectively.

```
function GET($\mathcal{B}, x$)
    $T \leftarrow \mathcal{B}.T$
    for each distinct $h_i(x)$, for $i = 1, \cdots, k$ do
        if $T[h_i(x)]$.count $= 0$ then
            return $(NoMatch)$
        else if $T[h_i(x)]$.count $= 1$ then
            if $T[h_i(x)]$.keySum $= x$ then
                return $(Match,\ T[h_i(x)]$.valueSum$)$
            end if
        else if $T[h_i(x)]$.count $= -1$ then
            if $T[h_i(x)]$.keySum $= -x$ then
                return $(DeletedMatch,\ -T[h_i(x)]$.valueSum$)$
            end if
        end if
    end for
    return $(Inconclusive)$
end function
```

Essentially, the GET operation goes through all the cells in T that the hash functions h_i map to, and if one of the cells has a `count` of 0, we can be sure that the key isn't currently in the table[69].

On the other hand, if the cell has a `count` of 1, while having `keySum` $= x$, we know that the cell contains only the key-value pair (x, y) and we can return the value with certainty. The same goes for cells with a `count` of -1 and `keySum` $= -x$, which translates to a key-value pair (x, y) that has been deleted without being inserted, and we can return that as well as a flag that indicates that this is a deleted pair.

If none of the above cases were satisfied, we cannot know if the key has been inserted into the table or not, and we must then return $(Inconclusive)$ to signal this uncertainty.

With the GET operation covered, the final operation looks as follows:

```
function LISTENTRIES($\mathcal{B}$)
    $Result \leftarrow []$                                  ▷ Create output list
    $W \leftarrow \mathcal{B}.T$                            ▷ Create working copy $W$ of $\mathcal{B}.T$
    while there exists an $i \in [1, m]$ such that $W[i]$.count $= \pm 1$ do
        if $W[i]$.count $= 1$ then
            add the pair $(W[i]$.keySum$, W[i]$.valueSum$)$ to $Result$
            DELETE($\mathcal{B}, W[i]$.keySum$, W[i]$.valueSum)
        else if $W[i]$.count $= -1$ then
            add the pair $(-W[i]$.keySum$, -W[i]$.valueSum$)$ to $Result$
```

[69]If two distinct keys k_1 and k_2 map to the same cell c_0, and k_1 is inserted while k_2 is deleted, we also get a cell count in c_0 of 0. The following Section 2.5.5 shows how to deal with this case.

```
            INSERT($\mathcal{B}$, $-W[i]$.keySum, $-W[i]$.valueSum)
        end if
    end while

    if there exists an $i \in [1, m]$ such that $W[i]$.count $\neq 0$ then
        return ($Incomplete, Result$)
    else
        return ($Complete, Result$)
    end if
end function
```

The ListEntries operation searches through the lookup table T looking for cells with `count` $= \pm 1$; these cells contain only one inserted or deleted key-value pair, and can as such be stored in the result list and then removed from the lookup table, possibly changing other cells' `counts` to 1 or -1, which can then be used to extract another key-value pair.

This find/remove cycle continues until no more cells with `count` 1 or -1 exist, and the operation returns the result list along with either *Complete* if all cells now have a `count` of 0, or *Incomplete* if at least one cell contains a non-zero `count`.

Note that the Get and ListEntries operations provided above are modified from the original simple version in the paper to include support for recovering deleted key-value pairs. This addition is indeed described in the paper, along with some other additions to introduce additional fault tolerance to the IBLT:

2.5.5 Handling extraneous deletions

This section shortly describes one of the so-called fault tolerance mechanisms described in the IBLT paper.

The other fault tolerance mechanisms (multiple values for the same key, duplicate inserts/deletions of the same key-value pair, and fault tolerance to lost memory subblocks) are omitted as they won't be used in this thesis.

Let's consider the case where two key-value pairs with distinct keys (k_1, v_1) and (k_2, v_2), which both map to the same cell (a *shared cell*), have been inserted, and then a third pair (k_3, v_3), which coincidentally also maps to the same shared cell, is deleted (without a corresponding insert). This leaves the shared cell with a `count` of 1, which would cause the Get and ListEntries operations to regard $(k_1 + k_2 - k_3, v_1 + v_2 - v_3)$ as a valid key-value pair, which is obviously wrong.

This is mitigated by adding a fourth field, `hashkeySum`, to each cell, which contains the sum of hashes of all keys mapped to that cell. Note that the hash function used for hashing the key should be different than any of the k hash functions used to determine the cells in which to map a key.

When inserting or deleting entries, the Insert and Delete now also adds or subtracts the hash of the key in the `hashkeySum` field. And the Get and ListEntries operations checks that the hash of the key matches the `hashkeySum` before accepting a key-value pair as valid.

The size of the `hashkeySum` field must be of sufficiently many bits to make collisions sufficiently unlikely. If we use, e.g., 128 bits for this field, collisions will occur with a probability of 2^{-128}, which for our purposes can be viewed as a negligible probability.

This way, we can detect and discard invalid key-value pairs resulting from extraneous deletions. However, we cannot necessarily discover the inserts and deletes that lead to this state.

2.5.6 The threshold value t

The threshold value t determines approximately how many entries the IBLT can hold before ListEntries starts returning inconclusive results with a non-negligible probability [20]:

> As long as m is chosen so that $m > (c_k+\epsilon)t$ for some $\epsilon > 0$, ListEntries fails with probability $O(t^{-k+2})$ whenever $n \leq t$.

The threshold value is dependent on m, the number of cells in the lookup table and k, the number of random hash functions used to map a key to cells. Or, more specifically, $t = m/c_k$, where c_k is calculated from k:

$$c_k = 1 \ / \ \sup\left\{\alpha : 0 < \alpha < 1; \forall x \in (0,1), 1 - e^{-k\alpha x^{k-1}} < x\right\}$$

The following table shows sample c_k values for small values of k:

k	3	4	5	6	7
c_k	1.222	1.295	1.425	1.570	1.721

Table 3: c_k values for $k = 3, 4, 5, 6, 7$ [20]

For example, to achieve a threshold value of $t = 100$ while using $k = 5$ hash functions, one would choose m to be approximately 143:

$$100 = m/1.425 \Leftrightarrow m \approx 143$$

2.5.7 The size of an IBLT

The amount of space required to hold an IBLT depends on the parameters used to instantiate it; in particular, the following parameters determines the size of an IBLT:

Cell count (m) The cell count is one of two determining factors, and the resulting size of the IBLT is linear to m.

Cell size (c) The cell size is the other factor, and is the sum of the size of the fields in each cell:

`count` (c_c) can safely be set to 4 bytes, allowing for `count`s between -2^{31} and $2^{31} - 1$.

`keySum` (c_{key}) should be big enough to contain the longest possible key. If, e.g., keys are SHA-256 hashes, c_{key} should be set to 32 bytes.

`valueSum` (c_{val}) should, just as c_{key}, be big enough the contain the longest possible value, or, more correctly, should be as big as the longest value you want to be able to retrieve. Setting c_{val} to a smaller value will truncate larger values to fit into c_{val} bytes.

`hashkeySum` ($c_{h_{key}}$) should be set big enough to avoid collisions between different keys as described in Section 2.5.5.

Constants (c_0) We also need to encode the above mentioned values to ensure that a receiver will be able to correctly decode and use the IBLT. This can safely be regarded to take up 8 bytes, one byte for each of the above, and one byte for k, the number of hash functions.

With these values in place, we can calculate the size requirements of an IBLT:

$$m \cdot c + c_0 = m(c_c + c_{key} + c_{val} + c_{h_{key}}) + c_0$$

As an example, an IBLT with the parameters $t = 100 \Rightarrow m \approx 143$, $c_c = 4$, $c_{key} = 32$, $c_{value} = 64$, and $c_{h_{key}} = 16$ will take up around 16 kilobytes of space:

$$143(4 + 32 + 64 + 16) + 8 = 143 \cdot 116 + 8 = 16,596 \text{ B} \approx 16 \text{ kB}$$

This IBLT has a threshold value t close to 100, allows for cell counts up to $2^{31} - 1$, has keys up to 32 bytes and values up to 64 bytes, and 16 bytes is used for storing the hash sums of the keys.

2.5.8 Using an IBLT to store only keys

The IBLT data structure as described so far is a key-value store where each key has a corresponding value, which is stored and retrieved along with the key in all the operations. If we don't need that functionality, but simply want to use the IBLT as a space-efficient storage mechanism for a list of keys, we can fairly easily do that:

We can either redefine all four operations to not include values and remove the now-unneeded `valueSum` field from each cell. This isn't very hard to do, but there is an easier option: We can simply set the size of all `value-Sum` fields to be 0 bytes. This way, no space is used to store values, and we can simply use empty values when inserting/deleting keys, and ignore the empty values when retrieving keys.

Using the same values from the last section, but without the `valueSum` field, we can calculate the size needed for a key-only IBLT with a threshold value close to 100, with 32 byte keys and a 16 byte hash sum of the keys. Removing the 64-byte value fields gives us a size of around 7 kilobytes:

$$143(4 + 32 + 0 + 16) + 8 = 143 \cdot 52 + 8 = 7,444 \text{ B} \approx 7 \text{ kB}$$

As we shall see in Section 3.5.3, we will use a key-only IBLT as a central data structure for timestamping many keys at once.

2.6 Voting theory

Throughout time many different voting schemes have been invented and used for deciding on politics or electing leaders of a group. All of these schemes can have the following desirable properties in a greater or lesser degree — usually there is a tradeoff involved between some of the properties:

- Integrity
- Verifiability
- Privacy/anonymity
- Coercion resistance

The following paragraphs will define these properties further:

Integrity The integrity of a voting scheme is determined by how much the results of the election can be tampered with. Ideally, it should be impossible for anyone to alter the results in any way, but usually there will be some risk that the final results can be altered.

Verifiability When discussing verifiability, we generally distinguish between two aspects of it — *individual* verifiability and *universal* verifiability.

The degree of which any individual voter can be convinced that their own vote is included in the final tally is called individual verifiability.

On the other hand, the extent to which anyone can check that the election results are correctly computed given the individual votes is denoted universal verifiability.

Privacy/anonymity Privacy and anonymity measures how much of a voter's identity is kept secret when voting and during the subsequent processes of the voting scheme.

Ideally, it should be impossible for anyone the deduce the identity behind any given vote.

Coercion resistance Coercion resistance is a measure of how much the system is able to prevent coercion of voters to vote in a certain way.

In order to prevent coercion, among other things, a voter should not be able to prove to anyone how they have voted. If they cannot prove to a malicious third party that they voted in a specific way, they can still claim that they did with no way to prove them wrong or right.

Also, filling out and casting the ballot should be done in private so no coercer can monitor and influence this process.

2.6.1 Information contained in a ballot

Another somewhat important aspect of an election is the amount of information that a voter must provide to fill out the ballot; if the election is just a simple yes/no question, the voter only needs to provide one bit of information. On the other hand, if the ballot is more complicated, e.g., "list these 10 candidates in order from most to least desired", the ballot essentially allows for $10! = 3{,}628{,}800$ different combinations, or almost 22 bits of information.

The amount of information to be encoded in the ballot is important because if that amount is large enough compared to the amount of voters, an entity could coerce a number of voters to vote a combination that is specific for exactly one voter. The entity could then later compare the combinations on the publicly available ballots in order to find out who (if any) has not voted as they were ordered, and thus corrupt the election. This attack is sometimes known as the "Italian Mafia attack" or the ballot-as-signature attack [16].

2.6.2 Consequences of an online voting scheme

It is worth noting that a great deal of coercion resistance is lost if a voting scheme allows for online voting; as soon as the actual vote casting is taken away from the controlled, private environments such as voting booths, it becomes significantly harder to enforce privacy in the vote-casting moment.

As an example, one could imagine the scenario where a family of eligible voters are coerced by the father who wishes to force everyone to vote in a certain way. It is virtually impossible to prevent such a malicious person from succeeding in coercing his peers.

This issue can be partly mitigated by the introduction of re-voting (discussed later in Section 4.1) where a coerced person later can discard his compromised vote and fill out the ballot differently when the coercer has left.

An obvious advantage of using an online voting scheme over an traditional, offline one, is the cost required to run an election. Where the traditional approach requires many man-hours of planning and executing, along with the cost of printing ballots and obtaining available locations for the polling stations, an online approach is orders of magnitude cheaper — the costs of running the election, printing ballots and finding locations are very small. This would allow for having elections way more often and for things that haven't traditionally been considered as worth running an election for.

2.7 Linkable ring signatures

Linkable ring signatures is a signature scheme published by Liu et al. in 2004 [23]. Put shortly, it allows anyone to compose a list, or *ring*, of identities (i.e., public keys) and sign a message on behalf of the entire ring without revealing which of the identities actually signed the message. In addition to this, the signatures are *linkable*, which means that two signatures from the same identity can be easily linked, so it is possible to detect if someone in the ring has signed more than one message, but without disclosing the identity of that signer.

Linkable ring signatures are also known under the more explicit name *linkable spontaneous anonymous group signature for ad hoc groups.*

Anonymity or rather *signer indistinguishability*, meaning that it is infeasible to determine which identity in the ring created the signature with better probability than random guessing.

Linkability is the ability to link two signatures from the same signer. The anonymity of the signer is still preserved.

Spontaneity meaning that the ring doesn't have any set-up phase and anyone can create the signature without asking the permission of the other identities or a group leader, as is the case with *group signatures* [13].

The following pages will give a brief overview of group signatures[70], ring signatures and linkability in ring signatures. Finally, a linkable ring signature adaption for elliptic curves will be presented.

Group signatures The notion of ring signatures is related to a similar, but in some ways different type of signatures, the *group signature*, which was first published in 1991 [13]. Put shortly, "Group signatures are useful when the members want to cooperate, while ring signatures are useful when the members do not want to cooperate" [30].

Ring signatures The idea of ring signatures were first published in 2001 [30]. Ring signatures differ from group signatures in the following ways:

- Group signatures require a trusted group manager who defines the group and distributes keys specifically made for that group, and as an inherent consequence of this...
- Group signatures require an initial *set-up phase* in which the group manager sets up the group, whereas ring signatures are created *ad hoc* with the signer defining the ring and creating the signature at the same time.

[70]Group signatures can be viewed as the predecessor to ring signatures

- Group signatures are usually constant sized as opposed to ring signatures which are usually linear in size to the number of members in the ring[71].

Linkability in ring signatures Linkability [23] — the property that allows a signature verifier to determine if two signatures come from the same private key — is designed as a tag which is intrinsically linked to a specific private key in a specific ring. Any verifier can compare the tags from two ring signatures to see if they come from the same signer identity. Note however that the two rings must be exactly equal for the tag to be usable; this means that the rings must consist of the same members *in the same order*[72].

2.7.1 Linkable ring signatures over elliptic curves

The original paper on linkable ring signatures [23] describes how to sign and verify with RSA instead of elliptic curves, so I had to adapt the provided algorithms to operate on elliptic curve points instead of RSA prime numbers. I wrote a blog post[73] about this, and will include the adapted algorithms below. Note that I do not take credit for anything other than adapting the original algorithms to operate on elliptic curves, and some of the language in the following specification is taken directly from the paper:

Let p denote a large prime number, E denote an elliptic curve, G denote a base point on the elliptic curve E with order p. Let $\mathbf{H_1} : \{0,1\}^* \to \mathbb{Z}_p$ and $\mathbf{H_2} : \{0,1\}^* \to \mathbb{Z}_p$ be some statistically independent cryptographic hash functions. For $i = 0, 1, \cdots, n-1$, each user i has a distinct public key Y_i and a private key x_i such that $Y_i = x_i G$. Let $L = Y_0, Y_1, \cdots, Y_{n-1}$ be the list of n public keys. Let $\mathbf{MapToPoint}(x, E)$ be a function that injectively maps an integer $x \in [1, p-1]$ to a point on the curve E, such as the try-and-increment algorithm described in Section 2.1.5.

Signing Given message $m \in \{0,1\}^*$, the signer, with index π, has private and public keys x_π and Y_π.

1. Compute $H = \mathbf{MapToPoint}(\mathbf{H_2}(L), E)$ and $\tilde{Y} = x_\pi H$
2. Pick a random $u \in \mathbb{Z}_p$, and compute

$$c_{\pi+1} = \mathbf{H_1}(L, \tilde{Y}, m, uP, uH)$$

[71]Ring signatures that are sublinear in size have been proposed, see introduction of Au et al. [3].

[72]A simple way to agree on the order in a decentralized way is to sort the list of members.

[73]`https://jesper.borgstrup.dk/2014/04/linkable-ring-signatures-over-elliptic-curves/`

3. For $i = \pi+1, \cdots, n-1$ and $0, \cdots, \pi-1$, pick $s_i \in_R \mathbb{Z}_p$, and compute[74]

$$c_{i+1} = \mathbf{H_1}(L, \tilde{Y}, m, s_iP + c_iY_i, s_iH + c_i\tilde{Y})$$

4. Compute $s_\pi = u - x_\pi c_\pi \mod p$

The signature is $\sigma_L(m) = (c_0, s_0, \cdots, s_{n-1}, \tilde{Y})$.

$\tilde{Y}$ is the signer's unique *tag* for this exact ring. If they are to sign another message with the exact same ring, the tag will be the same and thus anyone can link two signatures simply by comparing the tags.

However, if the rings aren't exactly the same, even if the keys in the ring are just in a different order, the tag will also be different and thus unusable for linking any two signatures.

The c_i values "wrap around" to form a ring, as shown in Figure 3. Each c_i value (except $c_{\pi+1}$) is computed from the previous value: $c_{\pi+2}$ is computed from $c_{\pi+1}$, c_{n-1} is computed from c_{n-2}, c_0 from c_{n-1}, c_1 from c_0, and finally c_π is computed from $c_{\pi-1}$, "tying the knot" to form the ring.

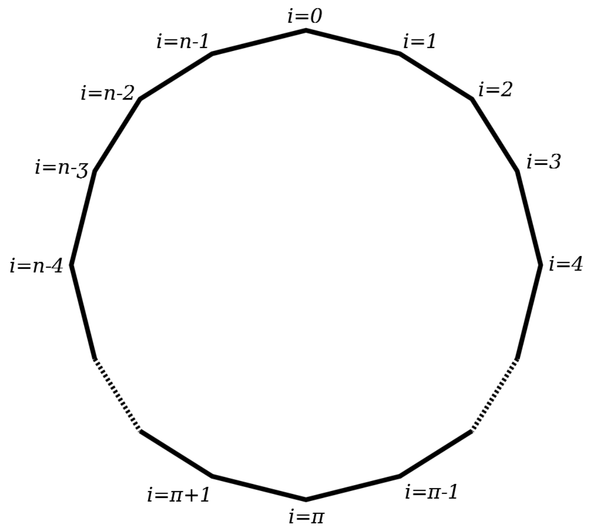


Figure 3: The n values in a linkable ring signature "wraps around", forming a ring where each link is equally likely to have been the value with the signer's index π.

The signer indistinguishability property of this signature scheme arises from the fact that the signature does not reveal the index π of the signer's key, and the signature verifier cannot determine this index because, although he can reconstruct all the c_i values, the c_π appears to fit right in the ring exactly as all the other c_i values.

[74]The values "wrap around", as described later, meaning that the c_0 value is computed as: $c_0 = \mathbf{H_1}(L, \tilde{Y}, m, s_{n-1}P + c_{n-1}Y_{n-1}, s_{n-1}H + c_{n-1}\tilde{Y})$

In much the same way, the s_π value cannot be identified: All the other s_i values are randomly chosen from $\mathbb{Z}_p$. The s_π value is computed mod p and cannot be distinguished from the other, random values.

Verifying The signature $\sigma_L(m) = (c_0, s_0, \cdots, s_{n-1}, \tilde{Y})$ on a message m and a list of public keys L is verified as follows:

1. Compute $H = \mathbf{MapToPoint}(\mathbf{H_2}(L), E)$ and for $i = 0, \cdots, n-1$, compute the following:

$$\begin{aligned} Z'_i &= s_i P + c_i Y_i \\ Z''_i &= s_i H + c_i \tilde{Y} \\ c_{i+1} &= \mathbf{H_1}(L, \tilde{Y}, m, Z'_i, Z''_i), \text{ if } i \neq n \end{aligned}$$

2. Check whether $c_0 = \mathbf{H_1}(L\tilde{Y}, m, Z'_{n-1}, Z''_{n-1})$. If yes, the signature is valid. Otherwise, it is not.

2.7.2 Size of a linkable ring signature

As noted previously, the size of a ring signature is linear to the number of public keys in the ring. Specifically, a linkable ring signature for a message m consists of the following tuple:

$$(c_0, s_0, \cdots, s_{n-1}, \tilde{Y})$$
$$c_0, s_0, \cdots, s_{n-1} \in \mathbb{Z}_p, \quad \tilde{Y} \in \mathbb{Z}_p \times \mathbb{Z}_p$$

This gives us $n+3$ integers in $\mathbb{Z}_p$. If we use the `secp256k1` elliptic curve as used in Bitcoin and Bitmessage, we have the field size $p < 2^{256}$ meaning that each of the $n+3$ integers can be stored with 256 bits or 32 bytes of space. This gives us the following size:

The size of a linkable ring signature on the `secp256k1` elliptic curve with n public keys in the ring is $32n + 96$ bytes.

2.7.3 Mapping directly to the curve

As the first step of the signing algorithm, we compute the following values:

$$\begin{aligned} H &= \mathbf{MapToPoint}(\mathbf{H_2}(L), E) \\ \tilde{Y} &= x_\pi H \end{aligned}$$

One could ask why should we even bother using the **MapToPoint** function when we could just compute $H = hG$, where $h = \mathbf{H_2}(L) \cdot G$ and G is the generator point of the curve.

It turns out that if we compute the H point this way, we make it possible for an attacker to determine the public key corresponding to the tag $\tilde{Y}$ given the list of public keys. We only need to realize the following:

$$\tilde{Y} = x_\pi H = x_\pi(hG) = h(x_\pi G) = hY_\pi$$

We now see that the tag $\tilde{Y}$ is equal to the public key Y_π multiplied by h, which is something that the attacker can easily check.

If we instead compute $H = \mathbf{MapToPoint}(h, E)$, it is no longer possible to deduce the public key behind the tag this way, because we map the hash h onto the curve without multiplication.

3 The decentralized deadline consensus protocol

In this section I present a protocol for achieving consensus of valid messages sent before a deadline in a decentralized manner. We will make no assumptions about the content of the messages or how to combine them to a result, if desired at all.

The protocol builds on top of a *shared bulletin board*, and is designed to delegate the interpretation of messages to a higher level *message interpretation layer* (MIL), as pictured in Figure 4. Section 3.1 outlines our requirements for the bulletin board. Sections 3.2 and 3.3 describe how we will use the Bitcoin blockchain for defining deadlines and timestamping messages. The details of the protocol are described in Sections 3.4 and 3.5. Finally, the message interpretation layer is described in detail in Section 3.6.

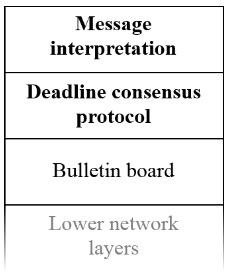


Figure 4: The deadline consensus protocol builds on top of a shared bulletin board and exposes an interface for a interpreting the messages to a higher-level message interpretation layer.

3.1 A shared bulletin board

For this protocol, every participant needs access to post to and read from a shared and anonymous bulletin board. It should be anonymous in the way that it isn't possible to determine which IP address a message came from. The messages we want to create a consensus around are posted to this bulletin board, and every participant is allowed to read any message on the board.

This bulletin board must support the following two methods:

- PostMessage(*message*): This method anonymously posts a new message to the board.
- MessageReceived(*message*): This is a callback method which will inform our protocol that a new message has been posted.

3.2 Deadlines on the Bitcoin blockchain

As mentioned in Section 1.2, we will define deadlines by means of a blockchain. In doing this, we want to be sure that the answer to the question "Has this deadline passed?" gives the same answer, no matter where in the world the question is asked. This ensures a canonical deadline which everybody can verify, but nobody has absolute control over.

The following section defines the *adjusted block timestamp* for the Bitcoin blockchain.

3.2.1 The adjusted block timestamp

A problem with the timestamps of the blocks on a decentralized blockchain is that they are the time provided by the miner who mined the block. This means that these timestamps cannot directly be used since the clocks of different miners can differ a lot. As an example, Table 4 shows the timestamps of blocks 326125-326144 on the Bitcoin blockchain. Note that blocks 326137 and 326141 appear to have been mined before the previous blocks. Obviously this isn't what happened, since each block contains proof[75] of being mined *after* the previous block. Instead, these anomalies are a result of the different miners' own clocks being different.

The blockchain, however, does have a mechanism in place to ensure that the block timestamps don't differ too much. Recall from Section 2.3.2 the following rule:

> A timestamp is accepted as valid if it is greater than the median timestamp of previous 11 blocks, and less than the network-adjusted time + 2 hours. "Network-adjusted time" is the median of the timestamps returned by all nodes connected to you.

Note that the timestamp of a new block must be greater than the median timestamp of the previous 11 blocks. This in effect ensures that the median timestamp of blocks $n - 10$ through n is greater than the median timestamp of blocks $n - 11$ through $n - 1$. Another way to put this is that we can be sure that *the median timestamps are constantly increasing*, while we cannot be sure about the same thing for the individual block timestamps. If we want to enforce deadlines with a blockchain, we have to be certain that a new block cannot set the time back and "undo" the deadline.

For brevity, when we mention the median timestamp of a single block n, we will actually refer to the median timestamp of blocks $n - 10$ through n. Figure 5 is a visualization of the blocks used to verify the timestamp of a new block on the blockchain.

In addition to the block timestamps, Table 4 also shows, for each block, the median timestamp of the blocks, and the time difference between the

[75]This proof is the hash of the previous block included in the block.

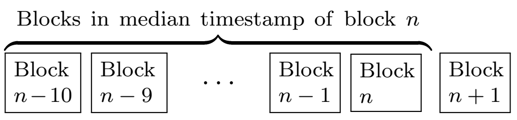


Figure 5: In order for a new block $n+1$ to be accepted on the blockchain, its timestamp must be greater than the median timestamp of the previous 11 blocks (the median timestamp of block n).

block's timestamp and the median. We can expect the block's median timestamp to be, on average, 50 minutes behind the block's reported timestamp: If the block in question is n, then the median timestamp will be selected from blocks $[n-10; n]$, and if each block takes an average of 10 minutes to mine, block $n-5$ will have the median timestamp and will on average be $5 \cdot 10 = 50$ minutes behind block n. We call this interval of 50 minutes the *adjustment constant.*

Thus we can add 50 minutes to the median timestamp in order to get a fair estimate of the time of block n, which we will call the *adjusted block timestamp.* Note that the average difference between the block timestamp and the adjusted timestamp for all 20 blocks in Table 4 is around 3 minutes — which is pretty close, although we can observe larger differences at times, such as the 122 minute difference in block 326132. Because the time for mining a single block is relatively unpredictable, these fluctuations will occur but will be flattened over time because the *average* time to mine blocks is fixed.

When we use the Bitcoin blockchain to get the current time, we do so by means of the adjusted block timestamp:

> The *adjusted block timestamp* for a block n on the Bitcoin blockchain is the median timestamp of the blocks $[n-10; n]$ plus 50 minutes.

3.2.2 Algorithms for adjusted block timestamps

This section describes three algorithms for working with adjusted block timestamps:

GetMedianBlockTimestamp(*block*): Compute the median block timestamp for a given block

GetAdjustedBlockTimestamp(*block*): Compute the adjusted block timestamp for a given block

GetFirstBlockWithAdjustedTimestamp(*time*): Find the first block whose adjusted timestamp is greater than or equal to a given timestamp, or NULL if no such block exists.

Block height	**Block timestamp**	**Adjusted timestamp**	**Difference (seconds)**
326125	05:08:30	05:14:22	0:05:52 (352)
326126	05:09:35	05:19:09	0:09:34 (574)
326127	05:18:51	05:34:37	0:15:46 (946)
326128	05:34:46	05:50:44	0:15:58 (958)
326129	05:47:02	05:54:20	0:07:18 (438)
326130	06:18:00	05:58:30	-0:19:30 (-1170)
326131	06:40:00	05:59:35	-0:40:25 (-2425)
326132	07:30:48	06:08:51	-1:21:57 (-4917)
326133	07:39:39	06:24:46	-1:14:53 (-4493)
326134	07:43:04	06:37:02	-1:06:02 (-3962)
326135	07:49:29	07:08:00	-0:41:29 (-2489)
326136	07:50:06	07:30:00	-0:20:06 (-1206)
326137	* 07:49:52	08:20:48	0:30:56 (1856)
326138	07:51:13	08:29:39	0:38:26 (2306)
326139	08:02:21	08:33:04	0:30:43 (1843)
326140	08:05:20	08:39:29	0:34:09 (2049)
326141	* 08:04:58	08:39:52	0:34:54 (2094)
326142	08:10:29	08:40:06	0:29:37 (1777)
326143	08:17:06	08:41:13	0:24:07 (1447)
326144	08:18:18	08:52:21	0:34:03 (2043)
Average			-0:03:00 (-180)

Table 4: Timestamps on the Bitcoin blockchain for blocks 326130-326144. Blocks with a * appear to have been mined before the previous blocks. The adjusted timestamp of the blocks (median of the timestamps of the block and the 10 previous blocks + 50 minutes) and the difference between these two timestamps are also shown. All times are on 2014-10-20 and in UTC.

Please note that these algorithms are not pure functions in the sense that they will not necessarily always return the exact same result given the exact same input. They all have an implicit dependency on the blockchain, which may change over time.

To avoid hardcoding Bitcoin constants, we have the two following blockchain constants:

NUMBER_BLOCKS: Number of blocks to use for calculating the median timestamp of a block. For Bitcoin, this number is 11.

BLOCK_TIME: Average time taken to mine a new block in seconds. For Bitcoin, this number is 600 (= 10 minutes).

We also have a few helper functions for retrieving blocks:

GetBlock(*number*): Returns the block with the given block number or NULL if no such block exists.

GetBlocksRange($from, to$): Returns a list of the blocks with block numbers in $[from; to]$. If some of the blocks don't exist, don't include them.

GetFirstBlockWithTimestamp(*time*): Returns the first block whose timestamp is greater than or equal to *time* or NULL if no such block exists.

For our purposes, a block is a simple data structure with the two following fields:

time: The timestamp of the block, as provided by the miner.

block_number: The block number of the block.

Figures 6 and 7 show the algorithms for getting the median and adjusted timestamps for a block.

To get the median timestamp for a block, we first ensure that the necessary blocks to compute exist. Then we sort the blocks by their timestamps, and then return the median timestamp. In order to get the adjusted timestamp, we simply compute the median timestamp and add the adjustment constant to it.

The third algorithm, which finds the first block whose adjusted timestamp is greater than or equal to a given timestamp, is shown in Figure 8. It basically starts with the lowest possible block whose adjusted timestamp can be greater than or equal to the provided timestamp, and then it computes the adjusted timestamp for that block. If the adjusted timestamp is large enough, we found the block, otherwise, try the next block.

```
function GetMedianBlockTimestamp($block$)
    $n \leftarrow block.\mathsf{block\_number}$
    $N \leftarrow \mathsf{NUMBER\_BLOCKS}$
    $Blocks \leftarrow$GetBlocksRange$(n - N + 1,\ n)$                ▷ 0-indexed
    Sort $Blocks$ by ascending timestamp
    if $N$ is even number then                      ▷ Median is average of the
                                                    ▷ two middle elements
        return ( $Blocks[\ \lfloor N/2 \rfloor\ ].\mathsf{time}\ +\ Blocks[\ \lceil N/2 \rceil\ ].\mathsf{time}$ ) /2
    else
        return $Blocks[\ N/2\ ].\mathsf{time}$
    end if
end function
```

Figure 6: GetMedianBlockTimestamp algorithm to get the median timestamp of a given block.

```
function GetAdjustedBlockTimestamp($block$)
    $Median \leftarrow$GetMedianBlockTimestamp$(block)$
    $AdjustmentConstant \leftarrow \mathsf{BLOCK\_TIME} \cdot ((\mathsf{NUMBER\_BLOCKS} - 1)/2)$
    return $Median + AdjustmentConstant$
end function
```

Figure 7: GetAdjustedBlockTimestamp algorithm to get the adjusted timestamp of a given block.

To find this start block (lowest possible block), we find the first block m whose timestamp is greater than or equal to the provided timestamp *minus* the adjustment constant. This block m is at least $BlocksToAdd = \lfloor \mathsf{NUMBER\_BLOCKS/2} \rfloor - 1$ blocks behind the first block whose adjusted timestamp can possibly be greater than or equal to the provided timestamp. So the start block is $m + BlocksToAdd$.

```
function GetFirstBlockWithAdjustedTimestamp($time$)
    $AdjustmentConstant \leftarrow \mathsf{BLOCK\_TIME} \cdot ((\mathsf{NUMBER\_BLOCKS} - 1)/2)$
    $MiddleTime \leftarrow time - AdjustmentConstant$
    $MedianBlock \leftarrow$GetFirstBlockWithTimestamp($MiddleTime$)
    if $MedianBlock$ is NULL then
        return NULL
    end if
    $BlocksToAdd \leftarrow \lfloor \mathsf{NUMBER\_BLOCKS/2} \rfloor - 1$
    $BlockNumber \leftarrow MedianBlock.\mathsf{block\_number} + BlocksToAdd$
    loop
        $Block \leftarrow$GetBlock($BlockNumber$)
        if $Block$ is NULL then
            return NULL
        end if
        $Adjusted \leftarrow$GetAdjustedBlockTimestamp($Block$)
        if $Adjusted \geq time$ then
            return $Block$
        end if
        $BlockNumber \leftarrow BlockNumber + 1$
    end loop
end function
```

Figure 8: GetFirstBlockWithAdjustedTimestamp algorithm to get the first block whose adjusted timestamp is greater than or equal to the given timestamp.

3.2.3 Adjusted timestamps and orphan blocks

Recall the concept of *orphan blocks* from Section 2.3.2, where a valid, mined block is abandoned by the network because another block belonging to a longer chain is discovered. Under specific circumstances, this switch can alter the current median block timestamp and thus the adjusted block timestamp as computed by the above algorithms.

To give an example, let's say that the network is currently at block n, and that the timestamps in blocks $n - 10$ through n are strictly increasing. This means that the median timestamp for block n is the timestamp of block $n - 5$. Now imagine that two blocks A and B are mined simultaneously —

block A has a timestamp larger than block n, but block B has a timestamp that is between those of blocks $n-5$ and $n-4$. Now, the clients using the blockchain with block A sees the current median timestamp to be the timestamp of block $n-4$, while those using block B computes the current median timestamp as the timestamp of block B. Figure 9 shows the blocks used for computing the median timestamps.

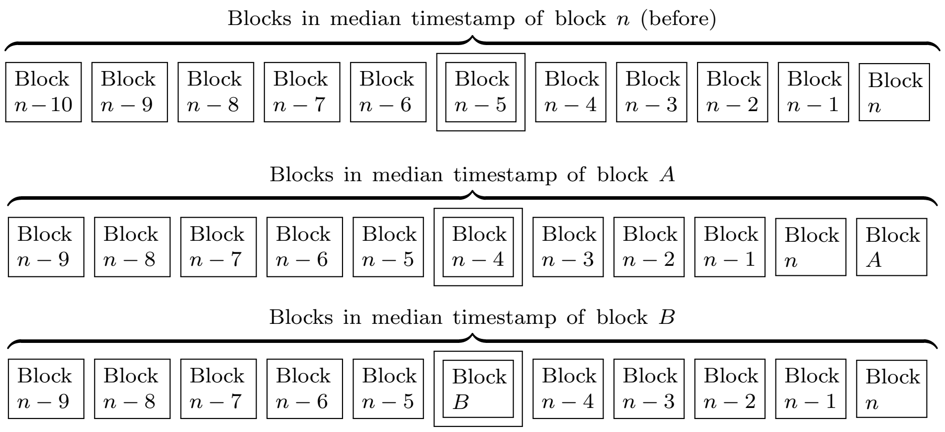


Figure 9: When two new blocks A and B are mined simultaneously, if block B has a timestamp between those of blocks $n-5$ and $n-4$, clients using blocks A or B report different median (and adjusted) timestamps.

While it is very uncommon for a newly mined block (B) to have a timestamp less than the previous 5 blocks ($n-4$ through n), it has happened 55 times before[76]. Out of these 55 times, only 5 of those has been in newer time where the Bitcoin network has had a significant hash rate. The Table 5 shows the date, block number and estimated hashrate for the 6 newest blocks where this has happened. We can see that it happens very rarely — 5 times in the latest 4 years, compared to 50 times before that (in a period of roughly one year[77]).

If a block like this is mined at roughly the same time as another block whose timestamp is larger than any of the 5 previous blocks, and the network is split between these two blocks, the two splits of the network will report different median timestamps.

Returning to the different blocks used for computing the median timestamps in Figure 9, if a deadline is defined as some time after that of block $n-5$ but before that of block B, observers who uses block A will report the deadline as reached, while those using block B will not. Furthermore,

[76]See appendix A.5 for a list of all such blocks.

[77]The first of these blocks, block number 24,157, was mined 2009-10-01 and the last one, 86,903, was mined 2010-10-22.

Date	**Block no**	**Network hashrate**[78] **(GHash/s)**
(2010-10-22)	*(86,903)*	*(25)*
2011-11-28	155,101	8,000
2011-12-06	156,368	8,000
2012-08-31	196,493	18,000
2014-01-06	278,850	10,000,000
2014-08-19	316,455	167,000,000

Table 5: The 6 latest blocks whose timestamps are lower than all of those of the previous 5 blocks

they will disagree on which block was the first to have reached the deadline adjusted timestamp.

The possible consequences of these *timestamp disagreements* are discussed later in Section 3.5

3.3 Timestamping on the Bitcoin blockchain

According to the formal description in Section 1.2, the timestamping service must provide a mechanism for *timestamping arbitrary messages, verifying these timestamps and deciding if a message was timestamped before or after a deadline.*

Bitcoin embeds a solution to the timestamping problem by establishing consensus on the current time as described in Section 2.3.2, so we can use it to timestamp arbitrary messages in both a canonical and decentralized fashion; by computing the hash of a message and using that hash as a Bitcoin address, we can send a small amount[79] of currency to that address, which will place the hash on the blockchain and serve as proof that the message was sent at that time.

The Bitcoin Wiki[80] has instructions on how to create a Bitcoin address from an arbitrary bitstring. Normally, this bitstring would be the public key corresponding to a secret private key, but if we pass the message that we want to timestamp as this bitstring, we get an address that corresponds to the message.

Such a scheme, called *Commitcoin*, was proposed in 2011 by Jeremy Clark [14] and has already seen multiple implementations such as Origin-

[79]As mentioned in the paragraph on dust in Section 2.3.1, the smallest possible transaction requires a transaction fee of 0.0001 BTC and transaction outputs of at least 0.0000543 BTC.

[80]`https://en.bitcoin.it/wiki/Technical_background_of_Bitcoin_addresses`, visited 2014-10-16

Stamp[81] and Crypto Stamp[82].

These services, however, only submit to the blockchain once per day[83], essentially making them *datestamping* services rather than timestamping.

We use a Commitcoin-like scheme for timestamping, but with no such once-a-day restriction, meaning that we get a precision of around 10 minutes[84] instead of 24 hours, making it suitable for finer timestamping tasks.

3.3.1 Verifying timestamps

As mentioned in Section 2.3.2, the de facto standard for accepting a transaction as confirmed, is to wait until it has 6 confirmations, which makes it extremely hard (if not practically impossible) to perform a double-spend.

If someone has success performing a double-spend when timestamping a message m, this would mean that the timestamp would be "forgotten" from the blockchain. But it's important to note that only the person performing the timestamping would be able to sign two transactions from the same address. So the only effect a double-spend would have in our situation is that a timestamp shortly would be on the blockchain and then disappear again afterwards.

If someone who is very frugal wants to timestamp a message, but doesn't want to spend any bitcoins if the blockchain already contains a timestamp for that message[85], this frugal timestamper could be fooled by a double-spend if the message already appeared timestamped on the blockchain, but would disappear again after they checked and thought that the timestamp would remain valid.

Two obvious mitigations to this potential problem are:

Wait for more confirmations If the timestamper can afford to wait some time, they could wait for more confirmations on the timestamp on the blockchain.

Always timestamp If a timestamper creates a timestamp on the blockchain for their message, regardless of if such a timestamp already exists, they can be sure that their own timestamp isn't double-spent. Given that the minimum transaction cost is 0.0001543 BTC (USD 0.06 at this writing[86]), this is a very small price to pay to avoid this problem.

[81]`www.originstamp.org`

[82]`www.cryptostamp.net`

[83]According to OriginStamp: "To keep the costs of running this free service low, we submit all hashes to the blockchain only once a day."

[84]10 minutes is the average rate of blocks being added to the Bitcoin blockchain. Other blockchains, such as Litecoin or Dogecoin, have average rates of 2.5 minutes and 1 minute, respectively.

[85]In theory, if the timestamper waits a bit before timestamping, another timestamper could already have timestamped the same message.

[86]1 BTC could be exchanged for USD 358.20 on 2014-11-10

Unless there is a compelling reason not to, I recommend always timestamping due to the very low cost to the timestamper.

When verifying timestamps, I suggest always requiring at least 6 confirmations to accept a timestamp as valid.

3.4 The participants

In the protocol, the participants can act as one or both of the following roles:

Posters only post messages to the board, and take no part in the actual consensus protocol.

Timestampers are responsible for timestamping the messages and jointly reaching the consensus.

The participants are divided into these two roles for two reasons:

- It should be possible for a participant to post their message(s) and then leave without having to take part in the actual consensus protocol. For example, a participant may not have constant access to the bulletin board, or they may be accessing the board from a mobile device for which it is not desirable to perform heavy computations.
- Given that we use the Bitcoin blockchain as the timestamping mechanism, not everybody are willing to spend a small amount[87] on the execution of this protocol.

It is important to stress that the successful consensus depends on the timestampers only; if no participants volunteer as timestampers, no messages will be timestamped and thus cannot be proven to have been sent before the deadline. The more timestampers participate, the more probable it is for a message close to the deadline to be timestamped.

Another incentive to participate as timestampers, is to ensure that your messages are indeed timestamped in case all the other timestampers collude to not timestamp your messages. This is discussed in more detail later in Section 6.2.5.

Note that the posters don't necessarily need to query the blockchain for the current adjusted timestamp if we "definitely" are in the posting phase according to their own clocks. As an example, imagine a posting phase stretching from midnight January 1st to midnight January 3rd; if their own clocks say that we are close to midnight January 2nd, they can safely assume that the protocol is in the posting phase, because the block timestamps on the blockchain are somewhat constrained by the entire network's view of the

[87]Supernodes must be prepared to spend BTC 0.0001543 for timestamping messages, which is around USD 0.06, EUR 0.04, or DKK 0.33 for exchange rates on 2014-11-10

current time (recall the definition of valid block timestamps from Section 3.2.1).

The following pages describe in detail the actions of the two roles during the different phases of the protocol:

3.5 The phases of the protocol

The protocol is divided into four phases — *before start*, *posting phase*, *timestamping phase* and *results phase* — where the end of one phases signifies the start of the next. Figure 10 shows an overview of the phases.

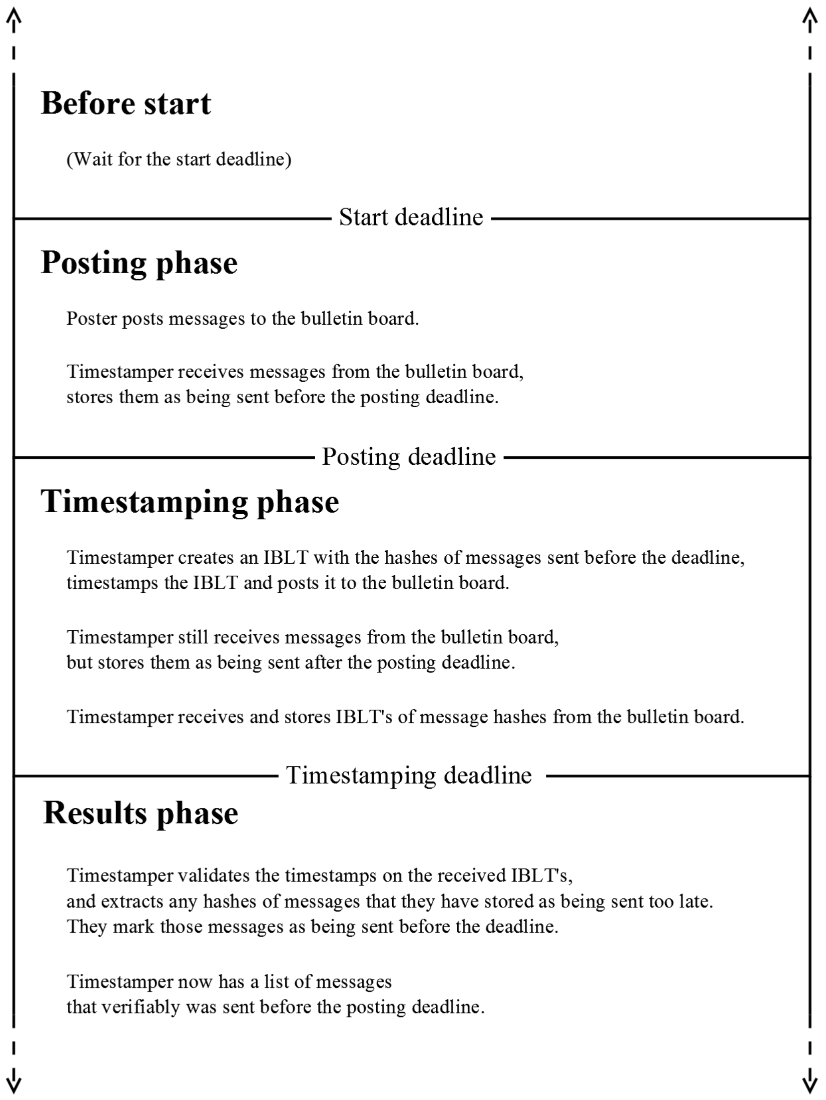

Before start
(Wait for the start deadline)
Start deadline
Posting phase
Poster posts messages to the bulletin board.
Timestamper receives messages from the bulletin board, stores them as being sent before the posting deadline.
Posting deadline
Timestamping phase
Timestamper creates an IBLT with the hashes of messages sent before the deadline, timestamps the IBLT and posts it to the bulletin board.
Timestamper still receives messages from the bulletin board, but stores them as being sent after the posting deadline.
Timestamper receives and stores IBLT's of message hashes from the bulletin board.
Timestamping deadline
Results phase
Timestamper validates the timestamps on the received IBLT's, and extracts any hashes of messages that they have stored as being sent too late. They mark those messages as being sent before the deadline.
Timestamper now has a list of messages that verifiably was sent before the posting deadline.

Figure 10: Overview of the four phases in the deadline consensus protocol

These four phases are defined by the three timestamps

$$(deadline_{start}, deadline_{post}, deadline_{timestamp})$$

All three timestamps are adjusted block timestamps as defined earlier in Section 3.2.1.

The following sections describe the four phases of the protocol in detail:

3.5.1 Before start

The first phase isn't strictly necessary, but is included as an easy way to define a starting time for the protocol, such that the following posting phase happens within a defined time interval. This phase could be skipped if no starting time is needed.

The end of the wait-for-posting phase indicates the start of the posting phase and is defined by the timestamp $deadline_{start}$.

Consequences of timestamp disagreements If nodes disagree on whether or not the start deadline has been reached and the posting phase has been started, some posters may be able to post their messages before other nodes would start posting. As long as it isn't necessary to enforce the start deadline as described later in Section 6.5.2, this makes no difference.

3.5.2 Posting phase

The posting phase is the only phase where the posters participate, namely by posting their message(s) to the shared bulletin board.

Each timestamper simply receives the messages and stores them locally after flagging them as being sent before the deadline as shown in the ReceiveMessage(*message*) algorithm in Figure 22 in the appendix.

The end of the posting phase is defined by the timestamp $deadline_{post}$, and marks the beginning of the timestamping phase. Clients implementing the protocol should disallow sending any more messages after the posting phase has ended.

Because the deadline of the posting phase is somewhat fuzzy — due to the nondeterministic nature of adjusted timestamps on the blockchain — it gives posters an incentive to post their messages some time before the deadline, in order to ensure that their messages will indeed be timestamped.

Consequences of timestamp disagreements If there are timestamp disagreements on whether or not the posting deadline has been reached, as explained in Section 3.2.3, the timestampers who think the deadline has been reached will regard messages as being sent too late, while the other timestampers will still see them as being sent on time. Thus, a message could be discarded if both (1) it is sent very close to the deadline and (2) all timestampers are in the part of the network that sees the deadline as being reached.

This could be mitigated by the poster posting their message earlier or by having more timestampers.

3.5.3 Timestamping phase

The timestamping phase is arguably the core phase of the consensus protocol.

In the beginning of the timestamping phase, each timestamper gathers all messages received so far (which have been flagged as being sent before the deadline) and then constructs a key-only IBLT containing the hashes of all these messages as the keys.

Each timestamper will timestamp their IBLT on the blockchain as described in Section 3.3, and post it on the bulletin board. The algorithm to do this is called PostTimestampCommitment and is shown in Figure 23 in the appendix.

The key-only IBLT will be constructed with the constants as described in Section 2.5.8 — 32 bytes used for storing keys (c_{key}), since we will be using SHA-256 hashes as keys and 16 bytes for the hash sum of the keys ($c_{h_{key}}$) and using 5 hash functions (k). The decision of the threshold value t and in turn the cell count m is delegated to the message interpretation layer as described later in Section 3.6, because it may provide a better estimate of the number of expected messages. The threshold value, the cell count and thus the size of the IBLT are all fixed for a specific instance of the consensus protocol.

During this phase and the next phase, the timestampers are still receiving and storing all messages, but they are flagged locally as being sent after the deadline as shown in the ReceiveMessage($message$) algorithm in the appendix. The timestampers store these messages in case they later learn that any of these messages were provably sent before the deadline.

The main reason to why we use IBLT's instead of just a plain list of message hashes, is that the size of an IBLT is $O(t)$, meaning that it is linear in the number of expected differences, instead of the list which has size $O(n)$, linear in the number of message hashes. The timestampers will agree on most messages being sent before the deadline, but may have a few differences. We save a lot of space by expecting that the timestampers mostly agree.

If a timestamper posts an IBLT that is too different from those of the other timestampers, the other timestampers will not be able to decode the entries in the IBLT. This means that a timestamper has an incentive to timestamp all timely messages, as the contents of the IBLT will then be close to the IBLT's from the other timestampers.

Since the IBLT has a constant size in each instance, posters cannot cause a denial of service in the timestamping phase by posting a lot of messages and having the timestampers post large commitment messages as a result, as it would be possible if we used any data structure whose size was linear in the amount of messages.

Note that if a timestamper isn't receiving messages for some time during the posting phase and doesn't start receiving them until after the timestamping phase has begun — for example, if they were disconnected from the network during the switch from posting to timestamping phase — they will mark all new messages as being too late, and will probably not be able to decode the IBLT's from the other timestampers.

Consequences of timestamp disagreements If timestampers disagree on whether or not the timestamping deadline has been reached and the results phase has started, there is a possibility that the blockchain transactions as created in the beginning of this phase have not yet reached enough confirmations to be validated. This can easily be mitigated by making the timestamping phase longer, i.e., by "pushing back" the timestamping deadline, delaying the results phase.

3.5.4 Results phase

In the results phase, each timestamper processes the valid commitments sent from the other timestampers in the timestamping phase in order to maximize the number of messages proven to be sent before the deadline.

Before we can process a commitment, we have to make sure that it is actually valid as described in Section 3.3.1. If this is not the case, the commitment must be discarded. The algorithm to validate the commitments can be found in the ValidateCommitmentMessages algorithm in Figure 24 in the appendix.

Then we prepare for processing the commitments:

- All n commitments contain an IBLT. We will denote those IBLT's as $\mathcal{T} = [T_1, \cdots, T_n]$ and flag all of them as not processed.
- We create a list L_h of the hashes of all the messages we have flagged as being sent before the deadline as well as an empty list $L_{new} \leftarrow \emptyset$.

Now we can start processing the commitments:

1. While there exists a not processed IBLT $\mathcal{B}$ for which ListEntries$(\mathcal{B} \setminus L_h)$ returns one or more hashes not present in either L_h or L_{new}, do the following:
   (a) Let $(Result, Hashes) \leftarrow$ListEntries$(\mathcal{B} \setminus L_h)$
   (b) Add the hashes to L_{new}: $L_{new} \leftarrow L_{new} \cup Hashes$

(c) If $Result = Complete$, mark $\mathcal{B}$ as processed.

2. Move the new hashes from L_{new} to L_h: $L_h \leftarrow L_h \cup L_{new},\ L_{new} \leftarrow \emptyset$
3. If any unprocessed IBLT's still exist and at least one hash was moved from L_{new} in the previous step, go back to step 1.
4. For each message flagged as being sent after the deadline whose hash is in L_h, flag it as being sent before the deadline.

Now we have maximized the number of messages that we can prove have been sent before the deadline.

The purpose of the extra list L_{new} to contain newly extracted hashes may not be immediately clear; the reason is to maximize our chances of decoding as many hashes from the IBLT's as possible: If all new hashes were immediately added to L_h, we run the risk of subtracting too many hashes from some IBLT's so we cannot extract any more and thus rendering that IBLT useless. On the other hand, if we never moved the new hashes from L_{new} to L_h, we run the risk of having subtracted too few hashes from some other IBLT's, again causing us to be unable to extract more hashes.

Instead, we use a hybrid solution, where we only move hashes from L_{new} to L_h if we have "exhausted" all IBLT's, meaning that we cannot extract any new hashes from any IBLT $T \setminus L_h$. We then try to exhaust the IBLT's again after adding more hashes to L_h.

This algorithm is called ProcessCommitmentMessages and is written as pseudocode in Figure 25 in the appendix.

3.6 The message interpretation layer (MIL)

The message interpretation layer (MIL) lies "on top of" the deadline consensus protocol (see Figure 4 on page 55), letting the underlying protocol handle the consensus process without any knowledge of the meaning of the messages passed around. This layer defines how to interpret the messages and how to combine the messages in order to produce a result, if any, and decides the IBLT threshold value t.

The MIL can have extra satellite data, further defining the interpretation of messages. Also, it must implement the five following functions used by the deadline consensus protocol:

3.6.1 Functions implemented by a MIL

- GetIbltThreshold(): Decide the IBLT threshold value t as described in Section 3.5.3.
- Serialize(): Serialize any extra data to binary format.
- Deserialize(): Deserialize any extra data from binary format.

- HASHDATA(): Compute hash from any extra data to create unique identifier.
- MESSAGEVALID(*message*): Validate an incoming message — is it valid or not?

The first function decides on a threshold value t for the IBLT's created by timestampers in the timestamping phase. This value in turn determines the size of the IBLT's. The next three functions handle any additional data needed by the MIL and are used to load and store the protocol details and to generate a unique identifier for this particular instance of the protocol. The last function decides if messages are valid or not.

The extra data mentioned could be anything needed to correctly validate and interpret the messages, or anything that defines characteristics of the specific instance of the consensus protocol[88].

The function MESSAGEVALID(*message*) is called by the consensus protocol when a new message arrives and must return if that message is valid and should be stored, or if it is invalid and should be discarded. The consensus protocol passes the raw message without any attempted interpretation, meaning that it is entirely up to the MIL to define how messages are to be interpreted.

The following pages describe a specific use case of a message interpretation layer, namely handling an anonymous election.

[88]As an example, the extra data of our voting scheme is the question and possible answers on the ballot, as well as the public keys of registered voters

4 A voting protocol built on the deadline consensus protocol

We will now use the decentralized deadline consensus protocol described previously to build a voting protocol that allows for anonymous voting among a group of registered voters while being able to detect if one of the registered voters tries to cast more than one vote. We will do this by creating a specific message interpretation layer (MIL) for the consensus protocol that will handle messages as votes.

In order for us to create and execute an election with our proposed scheme, we need (1) a *ballot form* that the voters must fill out, which essentially consists of a question and a list of possible answers to that question, and (2) the *public keys* of all those who should be able to vote[89], so we can identify and store votes from registered voters, while discarding votes from everyone else. These data are extra data to the MIL, which must be serializable and hashable, as described in Section 3.6.

Our voting scheme signs each message, or ballot, with a linkable ring signature so we can ensure the following:

- The vote comes from a registered voter.
- The identity of the voter cannot be determined with higher probability than random guessing.
- Double votes are easily detectable.

When a new message arrives, we must validate it in the MessageValid(*message*) function by checking that the included linkable ring signature can be verified in the ring of the public keys of the voters. This ensures that only ballots with valid signatures are stored, while discarding those without.

When the consensus protocol finishes, we can also discard ballots whose ring signatures have the same tag, as they are signed with the same key and can be considered attempts at double-voting. The remaining ballots can then be summed to compute the result of the election.

The protocol must also define a threshold value for the IBLT's sent in the timestamping phase. I propose using a function that increases with the number of voters, while the increase gets smaller and smaller as the number of voters grows, so we expect a higher difference as we get more voters without letting the IBLT's grow linearly. A good candidate for this function is the square root of the number of voters. Also, if the number of voters is sufficiently low, we want a minimum number of expected differences. I suggest having 10 as the minimum number, so we'll compute the threshold value for our voting protocol as follows:

[89]Note the implicit assumption here that every voter already has a public key known to whoever initiates the election.

$$t = \max(\ 10,\ \sqrt{|\mathcal{V}|}\ )$$

where $|\mathcal{V}|$ is the number of voters in the election.

4.1 Re-voting

It is possible to extend the above protocol to allow *re-voting*, meaning that a voter can cast a new vote from an address which have already cast a vote previously. The previous vote will the be discarded in favor of the new vote.

This is possible by *chaining* votes from the same tag: When a voter casts the first vote from a given voter address, they remember the hash of the vote in case they change their mind and want to cast a new vote later. In this new vote they include the hash of the previous vote to indicate that this new vote is a re-vote. It is possible for the voter to cast a virtually unlimited amount of re-votes, as long as each re-vote references a previously valid vote.

Of course, it must not be possible to have more than one vote per voter address counted towards the final tally. Thus, we must ensure that the chain of votes from the original vote to the final re-vote doesn't split into multiple chains. If this happens, we will assume that the voter is trying to cast more than one vote, and all votes from that address must be discarded.

The following algorithm GetValidVotes in Figure 11 shows how the voting protocol follows these chains of votes to get to a final list of votes that should be counted in the final tally. In this algorithm a vote is a simple data structure with two fields:

children: A set of other votes referring to this vote as the previous vote. This set is populated during the execution of GetValidVotes.

previous_hash: The hash of the previous vote, as provided by the vote signer, or NULL if the voter cast this as a first vote.

```
function GetValidVotes( $\mathcal{V}$ )
    $\mathcal{V}_{valid} \leftarrow \emptyset$                                  ▷ Set of all valid votes
    $\mathcal{V}_{tags} \leftarrow$ votes from $\mathcal{V}$ grouped by tag
    for all $(tag, \mathcal{V}_{tag})$ in $\mathcal{V}_{tags}$ do

        $\mathcal{V}_{hash} \leftarrow$ Map of votes in $\mathcal{V}_{tag}$ by hash
        $\mathcal{V}_{orig} \leftarrow$ all votes $v \in \mathcal{V}_{tag}$ where $v$.previous_hash = NULL
                                          ▷ $\mathcal{V}_{orig}$ is a list of all original votes
        if $|\mathcal{V}_{orig}| \neq 1$ then
            continue    ▷ Only process tags with exactly one original vote
        end if
        $v_{orig} \leftarrow \mathcal{V}_{orig}[0]$                              ▷ Only original vote

        for all $v$ in $\mathcal{V}_{tag}$ where $v$.previous_hash $\neq$ NULL do  ▷ All revotes
            if $v$.previous_hash $\in \mathcal{V}_{hash}$ then
                $\mathcal{V}_{hash}[v$.previous_hash$]$.children.append$(v)$
                        ▷ Add this revote to the children of its previous vote
            end if
        end for

        $v_{valid} \leftarrow$FollowVoteChain$(v_{orig})$
        if $v_{valid} \neq$ NULL then
            $\mathcal{V}_{valid}$.append$(v_{valid})$
        end if

    end for
    return $\mathcal{V}_{valid}$
end function

function FollowVoteChain( $v$ )
    if $v$.children $= \emptyset$ then
        return $v$                                             ▷ End of chain
    else if $|v$.children$| = 1$ then
        return FollowVoteChain($v$.children[0])              ▷ Follow chain
    else
        return NULL                                    ▷ More than one child
    end if
end function
```

Figure 11: Algorithm GetValidVotes($\mathcal{V}$) that produces a list of valid votes from the list of all votes $\mathcal{V}$ (including re-votes) and helper function FollowVoteChain(v).

5 Implementation of the protocols

This section describes the implementation of the consensus protocol described in Section 3 and the overlaid voting protocol from Section 4.

Section 5.1 details using Bitmessage as the shared bulletin board in the consensus protocol. Section 5.2 describes how Invertible Bloom Lookup Tables are implemented. In Section 5.3 the implementation of the Bitcoin blockchain is examined. Section 5.4 describes how Linkable Ring Signatures are implemented. Finally, Section 5.5 contains a list of the relevant source files in the implementation.

The implementation can be downloaded from GitHub at the following address: `https://github.com/jesperborgstrup/PyBitmessageVote`.

A guide to using the client implementation can be found in appendix A.1.

5.1 Using Bitmessage as the shared bulletin board

The two protocols require a shared bulletin board that, as mentioned in Section 3.1, must support the following two methods:

- PostMessage(*message*): This method posts a new message to the board.
- MessageReceived(*message*): This is a callback method which will inform our protocol that a new message has been posted.

For our purposes, we will use the decentralized mailing list functionality in Bitmessage as the bulletin board, described in Section 2.4.3, and we will use the "DML to DML" message sending mechanism, which, in addition to providing a shared bulletin board, also provides anonymity for the message senders, so a message on the bulletin board cannot be tied to any individual participant[90].

Using the DML functionality In order to use a DML (chan) in Bitmessage, we have to define a chan address with an associated keypair that will be used for encrypting and decrypting messages to and from the chan. To do this, we use Bitmessage's built-in function to deterministically create an address from a predetermined passphrase, namely the hash returned from the MIL's HashData() function, which creates a unique hash from the data associated with the consensus protocol and the MIL. Using this hash as a passphrase yields a unique address that we can use for this specific instance of the consensus protocol. When the address and its corresponding keys have been computed, we join the chan to be able to post and receive messages.

[90]If anonymity is undesired for whatever reason, the "Person to DML" approach can be used instead which will tie messages to the sender's personal Bitmessage address.

Everyone who wants to post or receive messages to and from this chan, must compute the address and join the chan.

Given that the Bitmessage reference client, PyBitmessage[91], is written in Python, my implementation is likewise Python code. My implementation was branched from commit `ced463bf`[92] of the PyBitmessage client. I should note, however, that I originally started implementation and testing from an earlier commit, but moved to this newer commit when I had cleaned up the implementation.

We assume that the message propagation in Bitmessage works fully, although the messages may possibly be delayed.

In order to fully integrate our protocols with the PyBitmessage client, I had to redesign two parts of the PyBitmessage code:

- Requesting of public keys had to be made more general
- The possibility of interpreting some messages as having special meaning instead of all being treated as plain-text messages

Requesting public keys In the original implementation of PyBitmessage, the requesting of public keys for unknown addresses is tightly coupled to the sending of messages. In fact, requesting the public key for an unknown address is one of the steps in sending a message, between the user pressing 'Send' and the message actually being sent.

The result of this was that the only way to request public keys was to send a message to the address for which you wanted the public key. This didn't reflect the underlying message protocol, where a public key request is a separate message type (`getpubkey`) from a normal message (`msg`).

This design was not suited for our purpose; imagine that you wanted to create a linkable ring signature with a ring of a few hundred members. You would have to compose and send messages to everyone, potentially informing them that they are about to be part of a ring, which may or may not be appropriate.

So, in order to separate these two functionalities, I refactored all the code that deals with requesting public keys to use a new data path as well as a new database table. This functionality can be found in the `helper_keys.py` source file.

Identifying protocol messages So far, the PyBitmessage client had treated all incoming messages as being the same; a plain text message with a subject and a body. If we want to fully integrate our protocols, the client will have to be able to treat some messages as having special meaning.

[91]`https://github.com/Bitmessage/PyBitmessage`

[92]Git commit `ced463bf232abe5f8b5109bddc31d3a421ee2f93` of 2014-10-22

Luckily, the encrypted data in the `msg` object contains a message encoding[93] field which, despite its name, is used to determine how the data should be interpreted and is largely unused[94], meaning that we can define our own encoding type[95] which flags messages in our consensus protocol. This way, the client can tell the difference between "our" messages and ordinary messages.

When a new message arrives, we add an extra check for our protocol encoding type, and redirect it to our protocol code if the encoding type matches.

5.2 Invertible Bloom Lookup Tables

For the IBLT's used in the commitments, no Python implementation existed at the time, so I created Py-IBLT, which is a IBLT data structure for Python with the following functionality:

- Customizable values for m (number of cells), k (number of hash functions), c_{key} (maximum key size), c_{val} (maximum value size), $c_{h_{key}}$ (key checksum size), and the hash functions used.
- Insertion and deletion of key/value-pairs
- Retrieval of key/value-pairs if possible
- Listing of contents if possible
- Serializing and deserializing to and from binary format

Refer to Section 2.5 for the meaning of the customizable values in the first point.

Note that we in our protocol only need key storage and retrieval (we do not need values associated with these keys), and as such Py-IBLT only supports checksums for keys and *not* for values[96].

See appendix A.4 for more information about Py-IBLT.

5.3 Blockchain usage

In order to be able to query the Bitcoin blockchain for transactions, blocks and addresses, and push transactions on the Bitcoin blockchain, we use the

[93]`https://bitmessage.org/wiki/Protocol_specification#Message_Encodings`, visited 2014-10-28

[94]Only 3 out of a maximum possible 2^{64} encodings are defined in the protocol specification

[95]The constant with a value of 12345 is defined as `ConsensusProtocol.ENCODING_TYPE` in `consensus/consensus_protocol.py`

[96]Storing and retrieving values is supported but without a hash sum for validating the values.

API at *blockr.io*[97]. The decision to use the blockr.io API was that it has identical API's for the standard Bitcoin network and the Bitcoin testnet[98].

To create, sign and post transactions, we used the third party `pybitcointools`[99] library. All Bitcoin and blockchain related code can be found in the source file `consensus/bitcoin_helper.py`.

Recall from Sections 2.4.2 and 2.4.7 that Bitmessage and Bitcoin keys are compatible and that Bitmessage uses two keys (one for encrypting and one for signing). This means that every Bitmessage address has two corresponding Bitcoin addresses, which can be computed from the public keys. We use this fact to compute the Bitcoin addresses from the signing keys of the user's Bitmessage addresses, so timestampers don't have to create new Bitcoin addresses and import the keys manually. Figure 12 shows a screenshot from the timestamper settings dialog, which shows a user's Bitmessage addresses along with the corresponding Bitcoin addresses and that address' balance.

The functions for detecting that an adjusted block timestamp has been reached and for determining the first block in which an adjusted block timestamp has been reached are in the `consensus/bitcoin_helper.py` file under the `BitcoinThread.getAdjustedBlockTimestamp(testnet, block_no)` and `BitcoinThread.getFirstBlockWithAdjustedTimestamp(testnet, timestamp)` methods, whose functionalities correspond to the algorithms listed in Section 3.2.2.

Please note that querying blocks and transactions as well as pushing new transactions on the blockchain happens through the use of centralized services, especially `blockr.io` but also `blockchain.info`. This was done in order to keep the implementation simple by not implementing or integrating a full Bitcoin node inside of the Bitmessage client. In order to be fully decentralized, the implementation should incorporate a "real" Bitcoin node as described later in Section 6.5.1, but this wasn't necessary for a proof-of-concept.

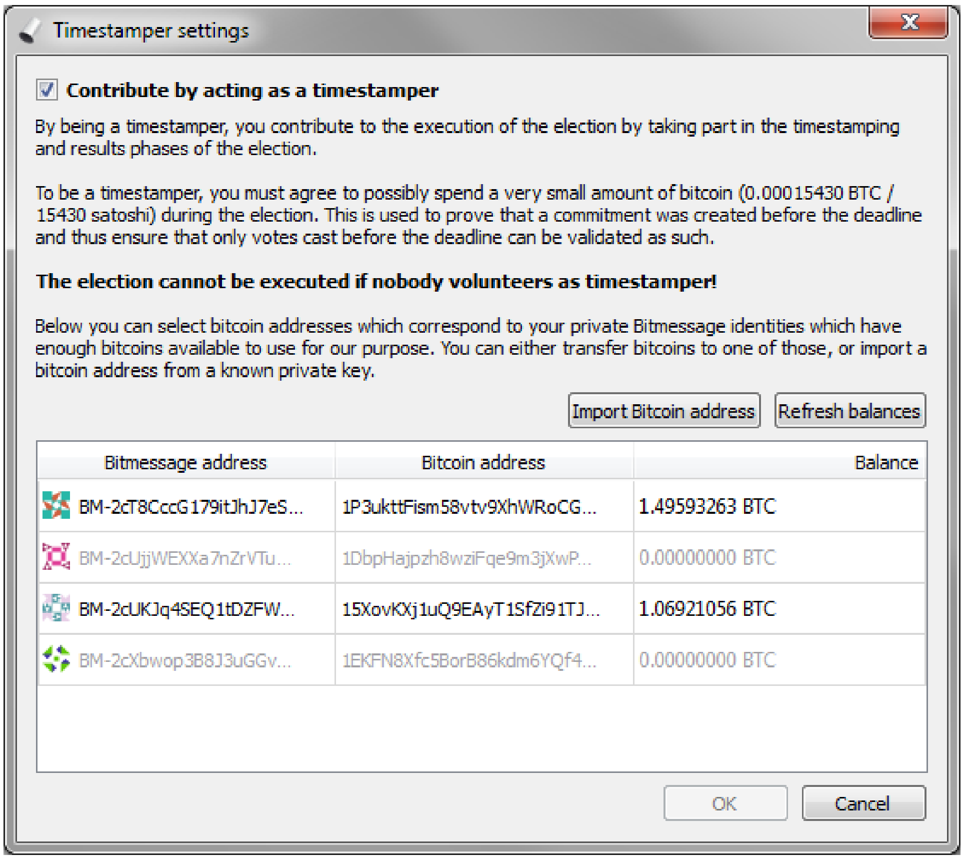
5.4 Linkable ring signatures

Just as with the IBLT's, no Python implementation of linkable ring signatures over elliptic curves existed before, so I created an elliptic curve helper library, Py-EC, and used that as a base for implementing linkable ring signatures.

[97] I use `http://btc.blockr.io` for everything except for pushing non-testnet transactions. We had problems using the blockr.io API for that and use the `http://blockchain.info` API for that instead.

[98] Having identical API's made testing free, since testnet bitcoins can be obtained for free at designated testnet bitcoin *faucets*, such as the one at `http://testnet.bitcoin.peercoinfaucet.com/`

[99] `https://github.com/vbuterin/pybitcointools`, licensed under the MIT license



Timestamper settings

☑ **Contribute by acting as a timestamper**

By being a timestamper, you contribute to the execution of the election by taking part in the timestamping and results phases of the election.

To be a timestamper, you must agree to possibly spend a very small amount of bitcoin (0.00015430 BTC / 15430 satoshi) during the election. This is used to prove that a commitment was created before the deadline and thus ensure that only votes cast before the deadline can be validated as such.

The election cannot be executed if nobody volunteers as timestamper!

Below you can select bitcoin addresses which correspond to your private Bitmessage identities which have enough bitcoins available to use for our purpose. You can either transfer bitcoins to one of those, or import a bitcoin address from a known private key.

Bitmessage address	Bitcoin address	Balance
BM-2cT8CccG179itJhJ7eS...	1P3ukttFism58vtv9XhWRoCG...	1.49593263 BTC
BM-2cUjjWEXXa7nZrVTu...	1DbpHajpzh8wziFqe9m3jXwP...	0.00000000 BTC
BM-2cUKJq4SEQ1tDZFW...	15XovKXj1uQ9EAyT1SfZi91TJ...	1.06921056 BTC
BM-2cXbwop3B8J3uGGv...	1EKFN8Xfc5BorB86kdm6YQf4...	0.00000000 BTC

Figure 12: The timestamper settings dialog showing corresponding Bitmessage and Bitcoin address along with the balance of the Bitcoin address. The “Import bitcoin address” button allows the user to import their own bitcoin private key in order to spend bitcoins from other addresses.

Recall again from Section 2.4.2 that each Bitmessage address is comprised of two private keys — an encryption and a signing key. The linkable ring signatures are signed with the Bitmessage signing key.

Note that we don’t have to use the Bitmessage private and public signing keys. It is merely a convenience since Bitmessage already has a public key infrastructure in place. If another set of keys are available for all voters, those keys could just as well be used for the linkable ring signatures.

For more information about Py-EC, see appendix A.3. This library is situated in the `ec/` folder of the implementation. The actual linkable ring signature implementation resides in `ringsignature/ringsignature.py`.

5.5 Relevant source files

This section contains a list of the source files that are relevant to my implementation of the consensus and voting protocols in PyBitmessage. All new source files are found in the `consensus/` folder.

- `bitmessageqt/__init__.py`: Extended to include voting scheme functionality in the UI
- `bitmessageqt/bitmessageui.*`: UI definitions modified to include the "Voting" tab in the client
- `bitmessageqt/createelectiondialog.*`, `bitmessageqt/electiondetailsdialog.*` and `bitmessageqt/timestampersettingsdialog.*`: Three dialogs needed for the voting UI: *Create new election*, *Election details* and *Timestamper settings*
- `class_singleWorker.py`: Worker thread modified to enable the following tasks in a background thread: Request public keys, load and initialize elections and compute and cast votes.
- `consensus/bitcoin/*.py`: Contains the `pybitcointools`[100] library
- `consensus/ec/*.py`: My Py-EC library, see also appendix A.3
- `consensus/pyiblt/iblt.py`: My Py-IBLT implementation, see also appendix A.4
- `consensus/bitcoin_helper.py`: Bitcoin blockchain methods, and timers for adjusted timestamps.
- `consensus/consensus_protocol.py`: The core functionality of the consensus protocol
- `consensus/consensus_helper.py`: A few helper functions for the consensus protocol
- `consensus/consensus_data.py`: An example (trivial) implementation of a message implementation layer (MIL). Also contains the data structure that defines the timestamps of a protocol
- `consensus/helper_keys.py`: Functions for retrieving private keys and requesting public keys
- `consensus/voting_data.py`: The voting protocol MIL
- `consensus/ringsignature.py`: My implementation of linkable ring signatures over elliptic curves

[100]`https://github.com/vbuterin/pybitcointools`, licensed under the MIT license

6 Evaluation

In this section I will analyze and evaluate the design and implementation proposed in the previous sections, assess their strengths and weaknesses, describe the threat model, and finally suggest improvements to both the design and implementation.

6.1 Scalability

This section discusses the scalability of the voting protocol based on using Bitmessage as the shared, anonymous bulletin board. We shall see that the optimal number of voters in a single election is a trade-off between low storage, bandwidth and computing requirements on one hand, and high anonymity on the other.

6.1.1 Size of an election

As mentioned in Section 2.4.5, each message is restricted to a maximum size of 256 kB (= 262,144 bytes). If we combine that with the size of a linkable ring signature from Section 2.7.2, which is $32n + 96$ bytes, where n is the number of members in the ring, we can compute the maximum number of voters as such:

$$262{,}144 \geq 32n + 96 + v \quad \Leftrightarrow \quad n \leq 8{,}189 - \frac{v}{32}$$

... where v is the maximum size of the filled-out ballot. So if we have that the maximum ballot size is, e.g., 128 bytes[101], this leaves us with a maximum number of 8,185 voters.

While keeping the number of voters as high as possible would be desired for voter anonymity, having the maximum number of registered voters would require a lot of storage space on each client[102]. If we assume that all n voters cast one vote of size $32n + 96 + 128$, each client would have to reserve the following amount of bytes of space just for this one election:

$$n(32n + 96 + 128) = 32n^2 + 224n$$

This evaluates to 2,145,648,640 bytes or approximately 1.998 GB for $n = 8{,}185$, which is quite a lot for Bitmessage, considering that Bitmessage users have been experiencing problems when their local message store grew to around 600 MB[103].

Figure 13 shows how the total space required to store the votes of an election grows when the number of voters n grow, assuming that each voter

[101]In our voting protocol implementation, the theoretical maximum ballot size is just 9 bytes, the maximum size of a variable length integer [9]

[102]Not just on each voter client, but every client on the Bitmessage network.

[103]`https://bitmessage.org/forum/index.php?topic=4055.0`, visited 2014-11-04

casts one vote. We can see that if we keep the number of voters around 1,000, an election will only require around 32 MB of space from each client, which is much more acceptable than the almost 2 GB for a little over 8,000 voters.

The size of the commitment messages that are sent by the timestampers (see Section 3.5.3) is around 7 kilobytes, and thus isn't a significant factor compared to the size of the votes. Even if every voter acts as a timestamper, the amount of space required to store all commitment messages is $7,444n$ bytes. This gives us the following equation to decide when the votes take up more space than the commitment messages:

$$7,444n \leq 32n^2 + 224n \quad \Leftrightarrow \quad n \geq 225.625$$

Meaning that if the election has 226 voters, the space required to store all votes and all commitment messages is around $2 \cdot 1.60$ MB. If we have more voters, then *the vote size is the determining factor*. Also, this calculation is for the extreme case where every voter volunteers as timestamper, which most likely isn't the case in practice.

6.1.2 Proof-of-work needed to broadcast a vote

Another thing to consider when the number of voters and thus the size of the vote messages increase, is that the proof-of-work required to broadcast that vote also increases. Figure 14 shows how the target value decreases and the average time to compute the proof-of-work increases as the number of voters increase.

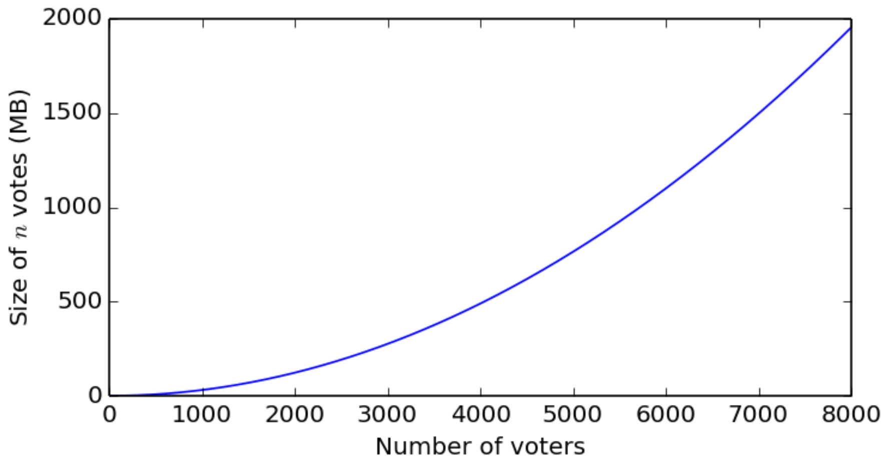
We can see that not only does the amount of voters have an influence on the size and bandwidth requirements of an election, but it also decides how much computing power a voter is required to use in order to broadcast their vote. While an average time of around two minutes for approximately 1,000 voters may seem acceptable for most purposes, an election may infer special circumstances where a voter can't afford to spend this much computing power to broadcast a vote.

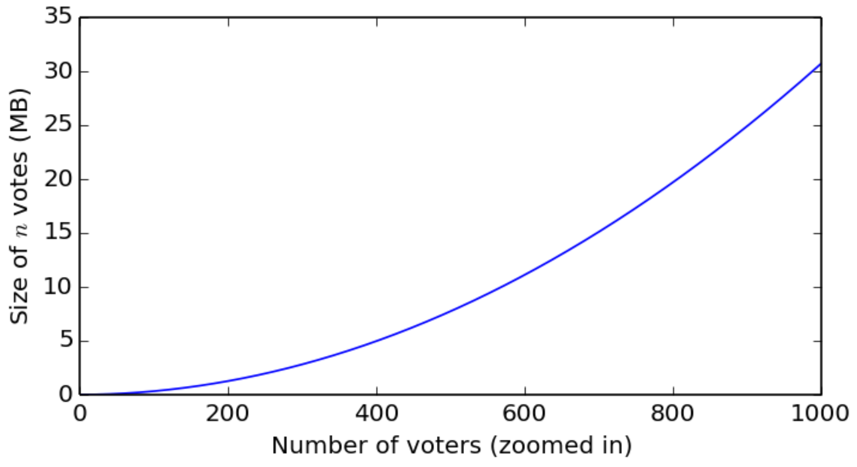
6.1.3 Subdivision of elections

If an election has more than a thousand voters (or whatever number we decide is suitable for n), instead of creating one huge election with all the voters together, we can divide the voters into *subelections* and join the results from those subelections into a final result.

This approach is not unlike that of using many smaller polling stations from regular, offline parliamentary elections. But where the division of voters is somewhat restricted geographically in those elections[104], the subdivision of voters in an online election scheme such as the one in this thesis can

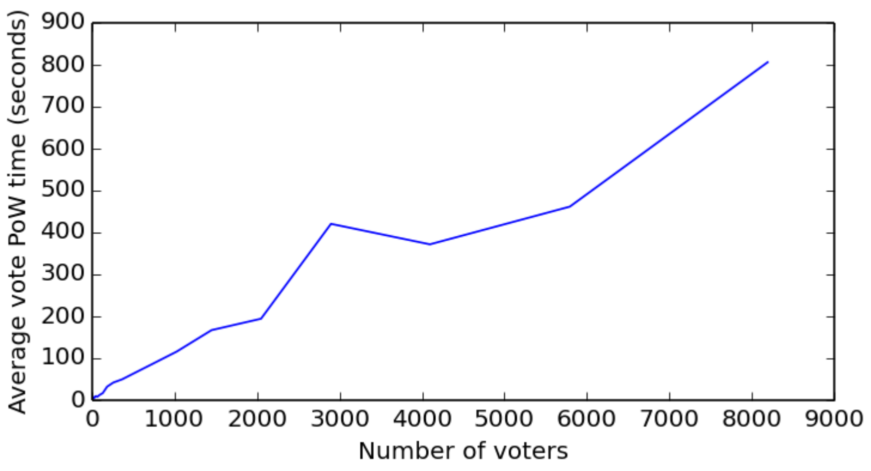
[104]Voters that have homes close to eachother vote at the same polling station





Voters (n)	Space	Voters (n)	Space
2	576 B	128	540.0 kB
4	1408 B	256	2.05 MB
8	3840 B	512	8.11 MB
16	11.5 kB	1024	32.22 MB
32	39.0 kB	2048	128.44 MB
64	142.0 kB	4096	512.88 MB

Figure 13: Space required to store all votes from one election relative to the number of voters (n) in the election, assuming that every voter casts one vote. First graph shows $n \in [1; 8000]$, second graph shows $n \in [1; 1000]$, and the table shows the space required for selected values of n.



Voters (n)	**Target** ($\log_2$)	**Time (average)**	**Voters** (n)	**Target** ($\log_2$)	**Time (average)**
2	43.838	0:04.733	128	41.685	0:16.852
4	43.762	0:05.350	256	40.847	0:42.094
8	43.619	0:04.011	512	39.935	1:04.725
16	43.370	0:04.912	1024	38.982	1:55.876
32	42.974	0:06.597	2048	38.005	3:14.379
64	42.407	0:08.323	4096	37.017	6:11.777

Figure 14: The proof-of-work increases as the amount of voters increase. The *Target* columns show the logarithm $\log_2 t$ of the desired target value. We can expect the proof-of-work to require on average $2^{64-\log_2 t}$ hashes before a suitable value is found. The graph and the *Time* columns show the average time to compute the required proof-of-work over 20 attempts on my Lenovo Thinkpad E420s. The raw data can be found in appendix A.6.

be completely arbitrary; voters can be divided by geography, age, occupation, or completely by random.

We have to be careful, however, to pick the right number of voters in each subdivision — more voters means more anonymity for the voters, but higher storage requirements for the network. On the other hand, fewer voters will require less storage, but provide poorer anonymity. The “best” number is a trade-off, and should be chosen with great care for the situation at hand.

6.2 Threat model

This section lists possible attack vectors that could compromise the anonymity of a voter or the integrity of the election.

When speaking of attacks that could reduce the anonymity, we use the attacker properties from Diaz et. al [17]:

Internal/External Internal attackers have control of one or more nodes, while external attackers can only eavesdrop on or control the communication channels.

Passive/Active Passive attackers can only listen to communications or read information. Active attackers can also alter information or communications.

Local/Global A local attacker only has control of part of a communication system, while a global attacker has access to the entire system.

If nothing else is mentioned, we assume that the election data, and thus the chan address and private keys for the chan, is public knowledge — any attacker knows of it.

6.2.1 Deanonymizing by private key

If an internal, passive attacker manages to retrieve a user’s private keys, the attacker can deanonymize any votes sent by that user. The tag, $\tilde{Y}$, of the linkable ring signature in a vote can easily be computed by anyone who has the private key and knows the list of members in the ring (see Section 2.7.1). Conversely, this scheme *does not* provide perfect forward secrecy.

Because Bitmessage stores private keys in plain-text on the user’s computer (see Section 2.4.8), anyone who can read the user’s local files is able to determine which votes are sent by that user. As mentioned, this is a vulnerability that is inherent to Bitmessage in its current implementation.

This could be mitigated by using a hardware signing device, such as the Trezor Hardware Bitcoin Wallet[105], that stores the user’s private keys and is used for signing messages without leaking the keys, even on compromised

[105]`http://www.bitcointrezor.com`, visited 2014-11-04

computers. In order for this to be realistic, Bitmessage would have to incorporate support for a hardware signing device, and the Trezor (or a similar device) would need a way of creating linkable ring signatures.

6.2.2 Deanonymization by message origin

If a global, passive attacker can listen to the communication between any clients, they can discover the IP address that any vote originated from by examining which client the vote was first sent from.

A user can mitigate this by hiding his vote inside an acknowledgment message to a third party, as the original paper describes as "passive operating mode" [36] and as briefly mentioned in Section 2.4.8, but this would require the third party to allow relaying of acknowledgment data, which they may have disabled[106]. The user could then try having multiple third parties relaying the vote until he comes across one that actually relays the vote.

Another way to circumvent this attack is to use the TOR network to encrypt and relay the Bitmessage traffic so the attacker cannot read the contents of the traffic. However, beware of the current limitations on using TOR with Bitmessage as outlined in Section 2.4.8.

This vulnerability is inherent to using Bitmessage as the shared, anonymous bulletin board, and not to the consensus or voting protocols.

If the attacker doesn't know of the election data — in particular the private key for the chan — he has no way of determining the contents of a message sent to that chan, or even knowing that it was sent to the chan.

6.2.3 Linking timestamper activity across consensus instances

Recall from Section 5.3 that the default[107] Bitcoin addresses are computed by using the private signing keys from a timestamper's Bitmessage addresses as the Bitcoin private keys.

This information could be used to link an individual's timestamper activity to that Bitcoin address if they re-use an address across several instances of the consensus protocol. While this isn't inherently a problem if a timestamper doesn't mind that information being public, they may not understand this possibility.

One solution to this problem, aside from explicitly importing Bitcoin private keys for each instance of the consensus protocol, is to derive a new private key that is specific to the current instance. This could be done by computing the private key as the hash of the Bitmessage private signing key concatenated with the election data. This way, the default addresses are different with each new instance.

[106]A user may choose to disable relaying of acknowledgment data due to the vulnerability "Using acknowledgments to deanonymize users" in Section 2.4.8.

[107]It is possible to import Bitcoin private keys and not use the computed Bitcoin addresses.

A drawback to this feature is that the private keys of the derived addresses are easily lost as an instance of the protocol completes. If a timestamper has deposited more bitcoins than needed into that address and the keys are lost, the bitcoins are out of the timestamper's reach forever.

6.2.4 Subverting the election with a Sybil attack

A Sybil attack [18] in our scheme can be performed by the election organizer who defines the list of voter addresses. If the organizer puts a sufficient amount of "fake identities" that he controls in this address list, he can control the outcome of the election simply by using all the fake identities to cast votes as he desires.

The voting protocol described in this thesis requires some trust in the organizer that decides which addresses are allowed to vote. This can be mitigated if the voters jointly decide on the voter list prior to an organizer creating the election data.

6.2.5 Misbehaving timestampers

Malicious timestampers could choose to only timestamp certain messages — e.g., in an election a timestamper could "forget" to timestamp some votes for a party which the timestamper doesn't want to win. That party, and all other parties as well, should run their own timestamping node to ensure that the votes for their party would indeed be timestamped —put simply, the more independent timestampers we have, the more "neutral" we can expect the list of timestamped messages to be.

Another thing is that a "rogue" timestamper can decide to commit to messages received after the beginning of the timestamping phase, i.e., timestamping too-late messages. The probability of this succeeding gets smaller and smaller as we approach the timestamping deadline, due to the fact that the timestamp needs at least 6 confirmations on the blockchain before the timestamping phase ends in order for the timestamp to be considered valid (see Section 3.3.1). Also, the other timestampers may not be able to fully decode the IBLT if the timestamper does this for many messages that haven't been timestamped by anyone else.

The other side of this is that if you really want to post a message after the timestamping phase has started, you can still do it if you volunteer as a timestamper and pay a small amount to commit to your own message. This means that in actuality, the "real" posting deadline is defined by the timestamping deadline and the number of confirmations required in order for a timestamp to be valid.

6.2.6 Coercion of voters

When voting no longer happens in a controlled environment, such as a physical, private polling station, it provides a possibility of voters being coerced. As an example, imagine a situation where a small community holds an election and everybody over the age of 12 gets one vote. A dominant patriarch could force his child to vote as he desires by being standing behind the child as he/she votes (or even by casting the vote himself from the child's computer).

Outside of a controlled environment, it is impossible to introduce complete coercion resistance, but by introducing the possibility of re-voting as described in Section 4.1, the coerced can invalidate the forced vote and instead cast his/her originally desired vote when the coercer has left.

Re-voting introduces another challenge, namely that the voter needs to remember the hash of the previous vote. The implementation already does this for the voter, but if the they need to cast their re-vote from another device, they would have to transfer the vote hash themselves.

This is by no means a complete solution for coercion resistance for our voting scheme, but it is an improvement over a similar voting scheme without re-voting.

6.3 Voting properties satisfied

Recall from section 2.6 the four described properties that are desirable for a voting scheme:

- Integrity
- Verifiability
- Privacy/anonymity
- Coercion resistance

I will here try to qualify how much each of these properties have been satisfied:

Integrity It is, by means of the linkable ring signature, infeasible to alter existing votes or create new ones in order to influence the final result of an election. However, it is possible for colluding timestampers to exclude certain votes from the set of valid votes as described in Section 6.2.5. However, this risk is minimized as more and more neutral timestampers take part in the timestamping phase.

Verifiability Both aspects of verifiability — universal and individual — are satisfied. Given that all votes are public, anyone who has the full list of votes can compute the result themselves. Given a the list of voters and a private key, we can compute the tag for that private key and check that a vote with that tag exists in the list of votes.

Privacy/anonymity Voters are anonymous in the set of all voters in the election, which for practical purposes will be smaller than a few thousand voters. Whether or not this number is large enough, depends on the anonymity requirements of a given election.

As mentioned previously, if a voter fails to secure their computer enough to allow for malware or viruses to obtain their private key, their anonymity is compromised. Since we cannot reasonably expect every voter to be able to protect their private computer against such attacks, the only viable solution for this may be to use hardware signing devices as suggested in Section 6.2.1.

Coercion resistance As explained in Section 2.6.2, an election loses a lot of its coercion resistance if it allows for casting votes outside of the controlled, private environments of voting booths. We have regained a little of the lost by introducing the possibility to re-vote, but as is symptomatic to online voting schemes, coercion of voters is a very real problem and must be treated as such.

6.4 Extensibility

Having divided the responsibilities between the consensus and the voting protocols, we can easily extend the decentralized functionality into other domains than voting. This section lists a few examples where new functionality can be built on top of the consensus protocol, simply by implementing a new Message Interpretation Layer (MIL).

When using our consensus protocol, everybody involved can see all messages posted and verify that no foul play was involved in selecting the valid messages.

Note that although the consensus protocol uses an anonymous bulletin board, the messages posted to the bulletin board can be signed with the identity of the poster if required.

Auctions Bidding on items is another example where we need to collect all bids before a deadline in order to determine the winner of the auction. The winner would then be the highest bid among all the posted bids.

This type of auction where the deadline cannot be precisely predicted ahead of time prevents the phenomenon of *online auction sniping* [32], where a bidder places a bid at the final moments of an auction, preventing other bidders to place a higher bid, and thus winning the auction.

Contests Imagine a contest where the contestants have to submit their contributions before a specific deadline. The consensus protocol would ensure that only contributions posted before the deadline would be considered.

Applications to a college, for a job position or the like can be posted using our consensus protocol, if it is desirable that everybody can see the applications posted.

6.5 Possible improvements

The following section is a list of possible improvements to the design and/or implementation.

6.5.1 Remove reliance on centralized blockchain services

As mentioned in Section 5.3, the implementation uses the API at `http://btc.blockr.io`. Relying on a single centralized service like this is contradictory to the main goal of this thesis, namely to propose a high-level design and working implementation a trustless, private and *decentralized* peer-to-peer voting scheme.

Although this is acceptable for a proof-of-concept implementation, because the core of the protocol indeed is decentralized, I suggest improving on the implementation by removing the reliance of a single centralized service.

An easy way to fix this would be to use more centralized services like *blockchain.info*, *blockexplorer.com*, *biteasy.com*, etc. However, this only makes the design rely on more centralized services, although we depend less on each service.

A better solution would be for the client to implement functionality to run or connect with a Bitcoin node on the local machine. This way we could read the necessary information from and push transactions to the Bitcoin blockchain in a decentralized way.

We don't even need to run a "full" Bitcoin node that has to store the entire blockchain locally. We have multiple options on how to run a decentralized Bitcoin client that uses as few resources as possible:

Simplified Payment Verification (SPV) The original Bitcoin paper actually proposes a mechanism for a "light-weight" network node [25]. In this proposal, the node doesn't validate blocks or store unneeded transactions, but relies on the network being controlled by honest nodes.

Connection Bloom Filtering Bitcoin Improvement Proposal (BIP) 37[108] is an accepted[109] proposal for the Bitcoin protocol that allows nodes to define which transactions to receive from any node, in order to not receive irrelevant transactions.

Any or both of these functionalities can be used to reduce the bandwidth, storage and computing costs of running a Bitcoin node for the purposes described in Sections 3.2 and 3.3, and thus make the implementation completely decentralized.

6.5.2 Enforce the starting deadline

If we wanted to enforce the starting deadline of the consensus protocol — i.e., to prevent posters from getting messages accepted which were sent before the posting phase had started — we could require posters to include in their messages the hash of the first block whose adjusted timestamp had exceeded the start deadline. Since this hash cannot be known in advance, its inclusion in a message would be proof that the message was created after the start deadline.

However, the poster would still have to make sure that they didn't include the hash of a block that ends up being an orphan block. To mitigate this, we could require that the messages include the hash of one of several possible blocks before the actual block that passed the deadline.

For example, if block n is the first block whose adjusted timestamp is greater than the start deadline, but ends up being an orphan block, we could simply require that the messages include the hash of any of the blocks $n-6, n-5, \cdots, n-1, n$. This would still make it possible to prepare and post messages slightly before the posting phase has canonically started, but we have limited the time in which a message can be posted significantly, and a poster that posts a message before the posting phase started cannot be certain that their included hash will eventually be one of the valid ones.

6.5.3 Prove the "freshness" of the election

The election data could, in addition to the ballot form and the list of voters, also contain the hash of a newly mined blockchain block. This would ensure the "freshness" of the election, and prevent an attacker from preparing any attacks that require some of the election data, such as the election hash or the derived chan address.

[108] `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`, visited 2014-11-05

[109] An accepted BIP means that the community has accepted the changes proposed, but the reference implementation has not yet been completed.

6.5.4 Eliminate running totals in an election

The way our voting protocol is realized has the inherent side-effect of broadcasting the current tally in real time. Having this information public has the power to actually influence the behaviour of voters who haven't voted yet, just as exit polls are blamed of having in traditional elections:

A 1986 study showed that "[e]xit polls appear to cause small declines in total voting in areas where the polls close late for those elections where the exit polls predict a clear winner when previously the race had been considered close." [33], and a 2013 study had similar conclusions while also showing that "exit poll information [significantly] increases bandwagon voting; that is, voters who choose to turn out are more likely to vote for the expected winner" [24].

It seems clear that having running totals broadcast in real time could have serious implications on the election result, and eliminating this possibility could be a great improvement to our voting scheme.

An approach for hiding the contents of the votes until the election is over, could be that of *time-lock puzzles* as introduced by Rivest et al. [29]. Time-lock puzzles are a way of "encrypt[ing] a message so it cannot be decrypted by anyone, until a pre-determined amount of time has passed".

However time-lock puzzles bear a resemblance to proof-of-work puzzles in the sense that they are both CPU-intensive for the solver. Furthermore, they are intentionally not parallelizable, so decrypting all votes may prove to be an insurmountable amount of work for a single node. If this direction were to be pursued, the nodes participating in the election would probably have to cooperate in order to decrypt all the votes and compute the final tally for the election.

7 References

[1] National Security Agency. *The Case for Elliptic Curve Cryptography.* Jan. 2009. URL: http://www.nsa.gov/business/programs/elliptic_curve.shtml (visited on 07/23/2014).

[2] Adrian Antipa et al. “Validation of Elliptic Curve Public Keys”. In: *Proceedings of the 6th International Workshop on Theory and Practice in Public Key Cryptography: Public Key Cryptography.* PKC ’03. London, UK, UK: Springer-Verlag, 2003, pp. 211–223. ISBN: 3-540-00324-X.

[3] Man Ho Au et al. “Short Linkable Ring Signatures Revisited”. In: *Proceedings of the Third European Conference on Public Key Infrastructure: Theory and Practice.* EuroPKI 2006. Turin, Italy: Springer-Verlag, 2006, pp. 101–115. ISBN: 3-540-35151-5, 978-3-540-35151-1. DOI: 10.1007/11774716_9.

[4] Adam Back. *Hashcash - A Denial of Service Counter-Measure.* Tech. rep. 2002. URL: http://www.hashcash.org/papers/hashcash.pdf (visited on 12/18/2013).

[5] BitmessageWiki. *Address.* Jan. 2014. URL: https://bitmessage.org/wiki/Address (visited on 08/21/2014).

[6] BitmessageWiki. *Decentralized Mailing List.* Jan. 2014. URL: https://bitmessage.org/wiki/Decentralized_Mailing_List (visited on 02/20/2014).

[7] BitmessageWiki. *Encryption.* Jan. 2014. URL: https://bitmessage.org/wiki/Encryption (visited on 08/22/2014).

[8] BitmessageWiki. *Mailing List.* Jan. 2014. URL: https://bitmessage.org/wiki/Mailing_List (visited on 02/20/2014).

[9] BitmessageWiki. *Protocol specification.* Jan. 2014. URL: https://bitmessage.org/wiki/Protocol_specification (visited on 08/22/2014).

[10] BitmessageWiki. *Protocol specification.* Jan. 2014. URL: https://bitmessage.org/wiki/Protocol_specification_v3 (visited on 10/30/2014).

[11] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782.

[12] Joppe W. Bos et al. “Elliptic Curve Cryptography in Practice.” In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 734.

[13] David Chaum and Eugène Van Heyst. "Group Signatures". In: *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques.* EUROCRYPT'91. Brighton, UK: Springer-Verlag, 1991, pp. 257–265. ISBN: 3-540-54620-0. (Visited on 03/11/2014).

[14] Jeremy Clark and Aleksander Essex. "CommitCoin: Carbon Dating Commitments with Bitcoin - (Short Paper)". In: *Financial Cryptography and Data Security - 16th International Conference, FC 2012, Kralendijk, Bonaire, Februray 27-March 2, 2012, Revised Selected Papers.* 2012, pp. 390–398. DOI: `10.1007/978-3-642-32946-3_28`.

[15] Explicit-Formulas Database. *Genus-1 curves over large-characteristic fields.* 2008. URL: `http://hyperelliptic.org/EFD/g1p/index.html` (visited on 07/23/2014).

[16] Roberto Di Cosmo. "On privacy and anonymity in electronic and non electronic voting: the ballot-as-signature attack." Apr. 2007. URL: `https://hal.archives-ouvertes.fr/hal-00142440`.

[17] Claudia Díaz et al. "Towards Measuring Anonymity". In: *Privacy Enhancing Technologies, Second International Workshop, PET 2002, San Francisco, CA, USA, April 14-15, 2002, Revised Papers.* 2002, pp. 54–68. DOI: `10.1007/3-540-36467-6_5`.

[18] John R. Douceur. "The Sybil Attack". English. In: *Peer-to-Peer Systems.* Ed. by Peter Druschel, Frans Kaashoek, and Antony Rowstron. Vol. 2429. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 251–260. ISBN: 978-3-540-44179-3. DOI: `10.1007/3-540-45748-8_24`.

[19] Niels Ferguson, Bruce Schneier, and Kohno Tadayoshi. *Cryptography Engineering.* John Wiley and Sons, Mar. 2010. ISBN: 978-0-470-47424-2.

[20] Michael T. Goodrich and Michael Mitzenmacher. "Invertible Bloom Lookup Tables". In: *CoRR* abs/1101.2245 (2011).

[21] Nermin Hajdarbegovic. *One Does Not Simply Find Satoshi Nakamoto.* Mar. 2014. URL: `http://www.coindesk.com/one-simply-find-satoshi-nakamoto/` (visited on 10/16/2014).

[22] Thomas Icart. "How to Hash into Elliptic Curves". In: *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings.* 2009, pp. 303–316. DOI: `10.1007/978-3-642-03356-8_18`.

[23] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. "Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups". In: *In ACISP'04, volume 3108 of LNCS.* Springer-Verlag, 2004, pp. 325–335.

[24] Rebecca B. Morton et al. *Exit Polls, Turnout, and Bandwagon Voting: Evidence from a Natural Experiment.* CREMA Working Paper Series 2013-01. Center for Research in Economics, Management and the Arts (CREMA), Feb. 2013. URL: `http://ideas.repec.org/p/cra/wpaper/2013-01.html`.

[25] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008. URL: `http://bitcoin.org/bitcoin.pdf` (visited on 12/17/2013).

[26] Max Raskin. *Bitmessage's NSA-Proof E-mail.* June 2013. URL: `http://www.businessweek.com/articles/2013-06-27/bitmessages-nsa-proof-e-mail` (visited on 08/26/2014).

[27] Certicom Research. "SEC 1: Elliptic Curve Cryptography". In: *Standards for Efficient Cryptography.* 2009.

[28] Certicom Research. "SEC 2: Recommended Elliptic Curve Domain Parameters". In: *Standards for Efficient Cryptography.* 2009. URL: `http://www.secg.org/download/aid-386/sec2-final.pdf`.

[29] R. L. Rivest, A. Shamir, and D. A. Wagner. *Time-lock Puzzles and Timed-release Crypto.* Tech. rep. Cambridge, MA, USA, 1996.

[30] Ronald L. Rivest, Adi Shamir, and Yael Tauman. "How to Leak a Secret". In: *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology.* ASIACRYPT '01. London, UK, UK: Springer-Verlag, 2001, pp. 552–565. ISBN: 3-540-42987-5.

[31] SafeCurves. *Choosing safe curves for elliptic-curve cryptography.* URL: `http://safecurves.cr.yp.to/` (visited on 07/23/2014).

[32] Ina Steiner. *Online Auction Sniping: The Thrill of the Hunt.* Feb. 2002. URL: `http://www.ecommercebytes.com/cab/abu/y202/m08/abu0077/s02` (visited on 11/13/2014).

[33] Seymour Sudman. "Do Exit Polls Influence Voting Behavior?" In: *Public Opinion Quarterly* 50.3 (1986), pp. 331–339. DOI: `10.1086/268987`. eprint: `http://poq.oxfordjournals.org/content/50/3/331.full.pdf+html`.

[34] Igor Tolkov. "Counting points on elliptic curves: Hasse's theorem and recent developments". In: (June 2009). URL: `http://www.math.washington.edu/~morrow/336_09/papers/Igor.pdf` (visited on 11/10/2014).

[35] Gonzalo Tornaría. "Square Roots Modulo p". English. In: *LATIN 2002: Theoretical Informatics.* Ed. by Sergio Rajsbaum. Vol. 2286. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 430–434. ISBN: 978-3-540-43400-9. DOI: `10.1007/3-540-45995-2_38`.

[36] Jonathan Warren. *Bitmessage: A Peer-to-peer Message Authentication and Delivery System*. Nov. 2012. URL: `https://bitmessage.org/bitmessage.pdf` (visited on 12/18/2013).

[37] Jonathan Warren. *Proposed Bitmessage Protocol Technical Paper*. Jan. 2013. URL: `https://bitmessage.org/Bitmessage%20Technical%20Paper.pdf` (visited on 08/22/2014).

A Appendix

A.1 Using the voting client

The client is based on the Bitmessage desktop client and extended with the decentralized consensus deadline protocol and the voting protocol as the Message Interpretation Layer on top of the consensus protocol.

In order to run the client, you need PyQt[110] 4 installed. The Bitmessage Wiki has instructions on how to install PyQt for Linux, OS X and Windows here: `https://bitmessage.org/wiki/Compiling_instructions`

A.1.1 Opening the client for the first time

The appearance of the client is similar to that of the original Bitmessage client, but with an extra tab called "Voting" in the top of the window. Opening this tab takes you to the empty voting dashboard as seen in Figure 15.

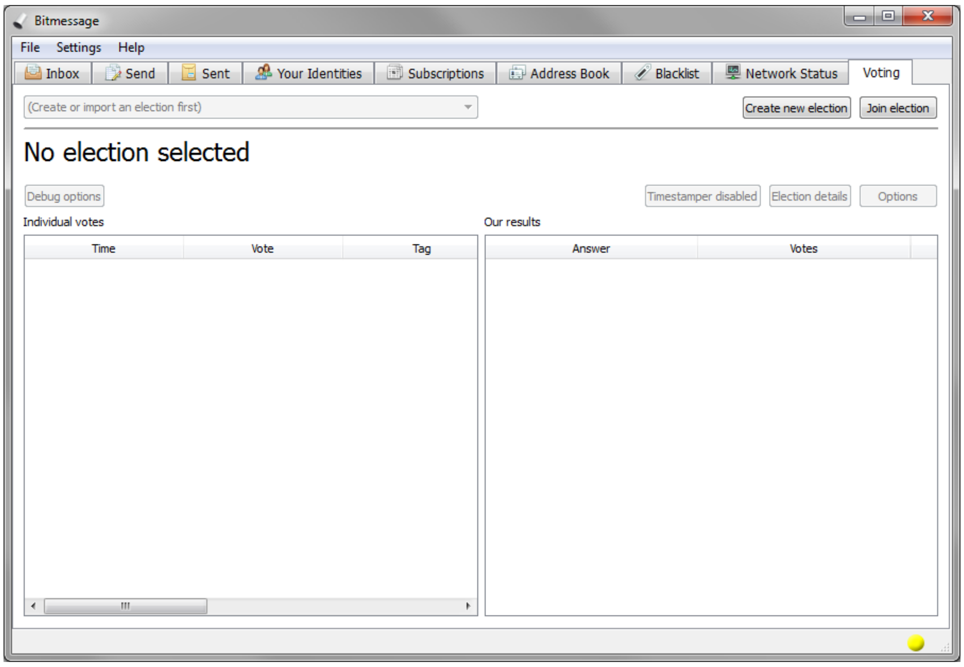


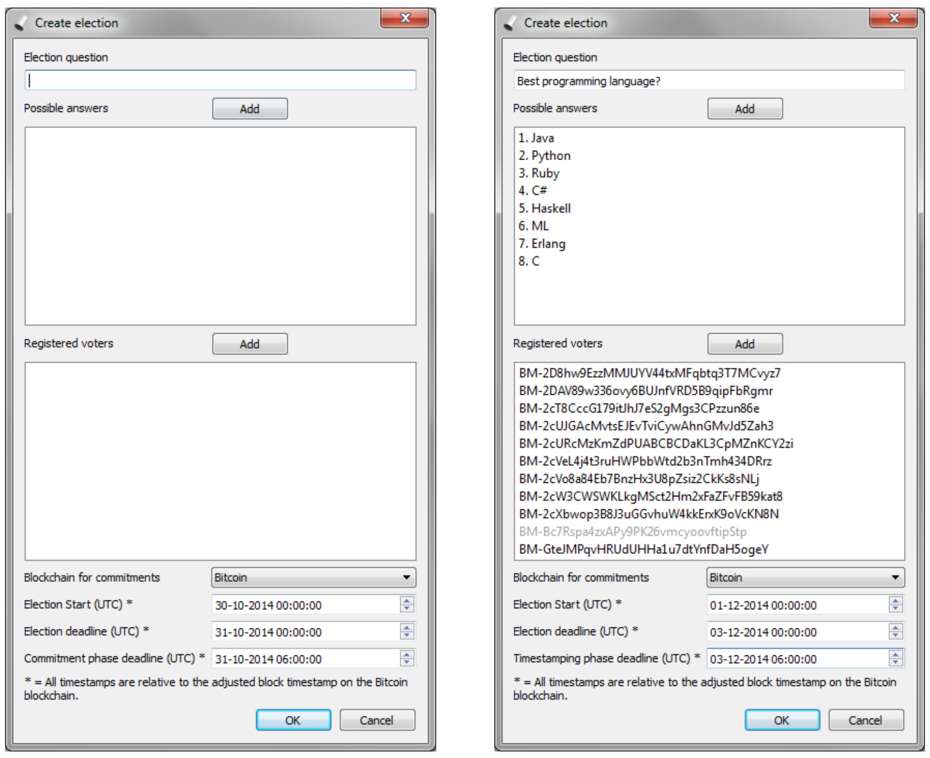
Figure 15: The empty voting dashboard (no elections loaded)

In order to participate in an election, you need to either create a new election or join an existing election. The following sections describe how to do just that.

[110]PyQt is a Python binding of the cross-platform GUI toolkit Qt.

A.1.2 Creating a new election

In order for an election to ever take place, someone has to create it in the first place. You do so by clicking the "Create new election" button in the top right corner of the voting dashboard. This opens a new dialog that looks like in Figure 16.



Election question

Possible answers

Registered voters

Blockchain for commitments: Bitcoin
Election Start (UTC) *: 30-10-2014 00:00:00
Election deadline (UTC) *: 31-10-2014 00:00:00
Commitment phase deadline (UTC) *: 31-10-2014 06:00:00
* = All timestamps are relative to the adjusted block timestamp on the Bitcoin blockchain.

Election question: Best programming language?

Possible answers
1. Java
2. Python
3. Ruby
4. C#
5. Haskell
6. ML
7. Erlang
8. C

Registered voters
BM-2D8hw9EzzMMJUYV44txMFqbtq3T7MCvyz7
BM-2DAV89w336ovy6BUJnfVRD5B9qipFbRgmr
BM-2cT8CccG179itJhJ7eS2gMgs3CPzzun86e
BM-2cUJGAcMvtsEJEvTviCywAhnGMvJd5Zah3
BM-2cURcMzKmZdPUABCBCDaKL3CpMZnKCY2zi
BM-2cVeL4j4t3ruHWPbbWtd2b3nTmh434DRrz
BM-2cVo8a84Eb7BnzHx3U8pZsiz2CkKs8sNLj
BM-2cW3CWSWKLkgMSct2Hm2xFaZFvFB59kat8
BM-2cXbwop3B8J3uGGvhuW4kkErxK9oVcKN8N
BM-Bc7Rspa4zxAPy9PK26vmcyoovftipStp
BM-GteJMPqvHRUdUHHa1u7dtYnfDaH5ogeY

Blockchain for commitments: Bitcoin
Election Start (UTC) *: 01-12-2014 00:00:00
Election deadline (UTC) *: 03-12-2014 00:00:00
Timestamping phase deadline (UTC) *: 03-12-2014 06:00:00
* = All timestamps are relative to the adjusted block timestamp on the Bitcoin blockchain.

(a) Empty dialog (b) Filled out dialog

Figure 16: The "Create election" dialog used to create a new election. The gray addresses in the "Registered voters" list in Figure (b) are addresses for which we don't yet have the private keys.

You have to fill out the following fields:

Election question The question that this election is about. Cannot be empty.

Possible answers Answers to the question that the voters can choose among. There must be at least two answers.

Registered voters The Bitmessage addresses of all voters who are to vote in the election. One address equals one vote. You must enter at least three addresses. If the client doesn't know some of the public keys to the addresses, those addresses are displayed in gray text.

Blockchain for commitments Choose between using the normal Bitcoin blockchain or the Bitcoin testnet blockchain for timestamping. Coins on the testnet are free, but the timestamps are very unstable. Unless you have a specific reason for doing so, use the normal Bitcoin blockchain.

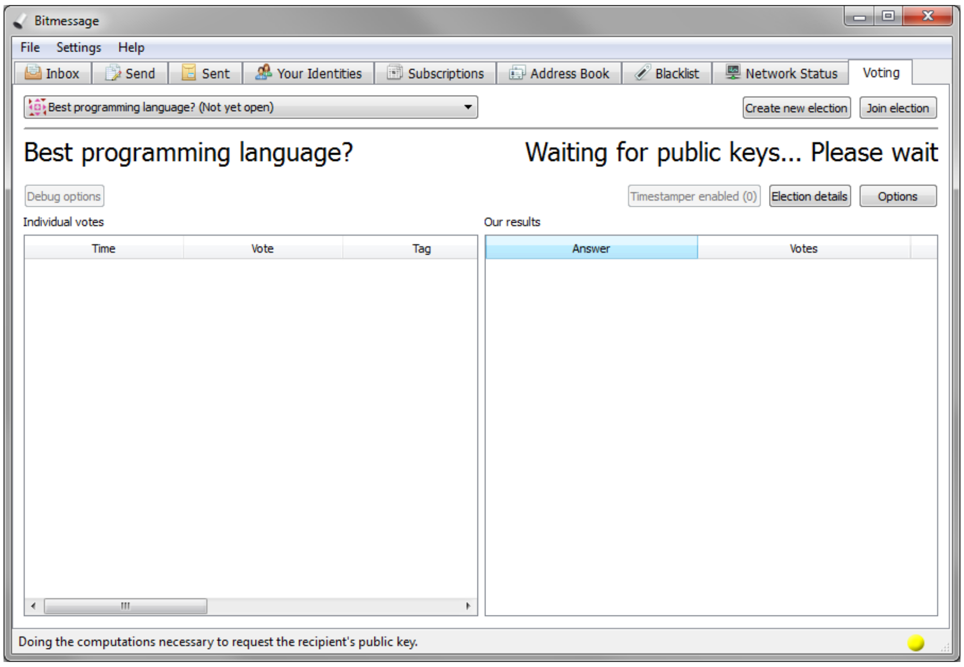
Election start The time for starting the election

Election deadline The time to stop the casting of votes and begin the timestamping phase. Must be after the election start.

Timestamping phase deadline The time to stop the timestamping phase and begin the results phase. Must be after the election deadline.

After you have entered the election details and pressed the “OK” button, you will return to the voting dashboard.

Missing public keys If some of the addresses in the dialog were gray, the client were missing the public keys for those addresses. Requests for those public keys are automatically sent out, and you will have to wait for those to arrive before anything can happen with the election. While the client is waiting for the required public keys, the voting dashboard looks like in Figure 17.



Doing the computations necessary to request the recipient's public key.

Figure 17: Voting dashboard when the client is waiting for public keys

While waiting for public keys, you can open the “Election details” dialog to see how many public keys the client has and how many it needs. This dialog is explained in detail later.

A.1.3 Joining an existing election

When you have created an election and want people to participate in it, they have two options for loading the election data into their clients. By clicking on the “Join election” button in the top right corner of the dashboard, you are presented with two options:

Join by election hash By entering the the unique 64-byte hexadecimal hash of the election, people are able to join the communications channel (*chan*) and receive messages through that channel. You can then broadcast the election data to the channel by clicking the button “Options” in the left of the dashboard and selecting “Broadcast election data”. After a short while, the election will be automatically loaded on their clients.

Import election from file You can export the election data to a file by clicking the “Options” below the “Join election” button and selecting “Export election to file”. This allows you to create a file with the data which you can send to the participants in whatever way you choose. When the participants import the file into their clients, the election is automatically loaded.

Please note that the creator’s client *must* have all the required public keys before they can broadcast or export the election data.

A.1.4 The voting dashboard

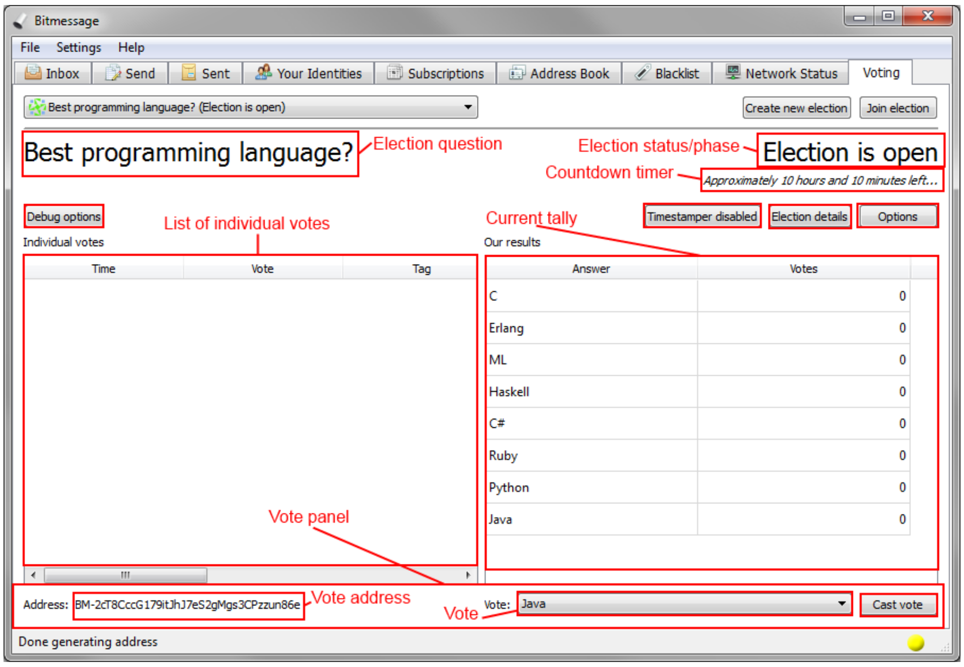
Once you have created or joined an election, the voting dashboard changes to something like what you can see in Figure 18. The voting dashboard has many controls and buttons, so we’ll go through them now from top to bottom, left to right:

Election question Shows the question of the currently selected election.

Election status/phase Shows the current phase of the selected election.

Countdown timer Shows an estimate of time left until the current phase ends.

Debug options button Allows you to manually trigger the different phases, and to clear all messages received in the election. This button should never be used in a real election, as they may alter the state and/or outcome of the election for this client.



Best programming language? (Election is open)

Best programming language?

Election is open

Approximately 10 hours and 10 minutes left...

Individual votes

Time	Vote	Tag

Our results

Answer	Votes
C	0
Erlang	0
ML	0
Haskell	0
C#	0
Ruby	0
Python	0
Java	0

Address: BM-2cT8CccG179itJhJ7eS2gMgs3CPzzun86e

Vote: Java

Done generating address

Figure 18: The voting dashboard with an election loaded that is ready for voting

Timestamper options button This button opens the timestamper settings dialog which allows you to activate or deactivate timestamper behavior for this client. The contents of this dialog is explained later in Section A.1.6. The text on the button reflects whether or not timestamping behavior is enabled or disabled. This button is only enabled before the timestamping phase. After that, you cannot make changes to the timestamper settings.

Election details button This opens the election details dialog which shows many variables about the current election. The dialog is explained later in Section A.1.5.

Options button This button allows you to broadcast or export the election data as mentioned earlier, and also allows you to remove the election from your client. Note that removing the election from your local client doesn't affect anyone else.

List of individual votes This is a list of all votes received (as well as your own votes). It shows the time the vote was received on the local client, the actual vote, the tag of the ring signature (the voter's anonymous identity), the hash of the vote, and the hash of the previous vote if the vote is a re-vote. The list is updated in real-time.

Current tally This table shows the current result of the election computed from the list of individual votes. Just as with the list of individual votes, this table is also updated in real-time.

Vote panel This panel is shown whenever the election is open for casting votes *and* you control one or more of the voter addresses.

Vote address This text shows the address you are using for voting. If you control more than one of the registered voter addresses, the text changes to a dropdown box where you can select which address to use. If the address has already cast a vote earlier, the text "(already voted)" will be added to the address. You can still perform a re-vote then.

Vote This dropdown box is used to select what you want your vote to be.

Cast vote button Clicking this button will cast the selected vote using the selected vote address.

A.1.5 Election details

Clicking on the "Election details" button in the dashboard brings up the election details dialog which looks like Figure 19.

This dialog lists a lot of variables and details about the election. The question, start time, deadline, timestamping phase deadline, # addresses, blockchain and registered addresses are the values provided when the election was created. The rest of the information is as follows:

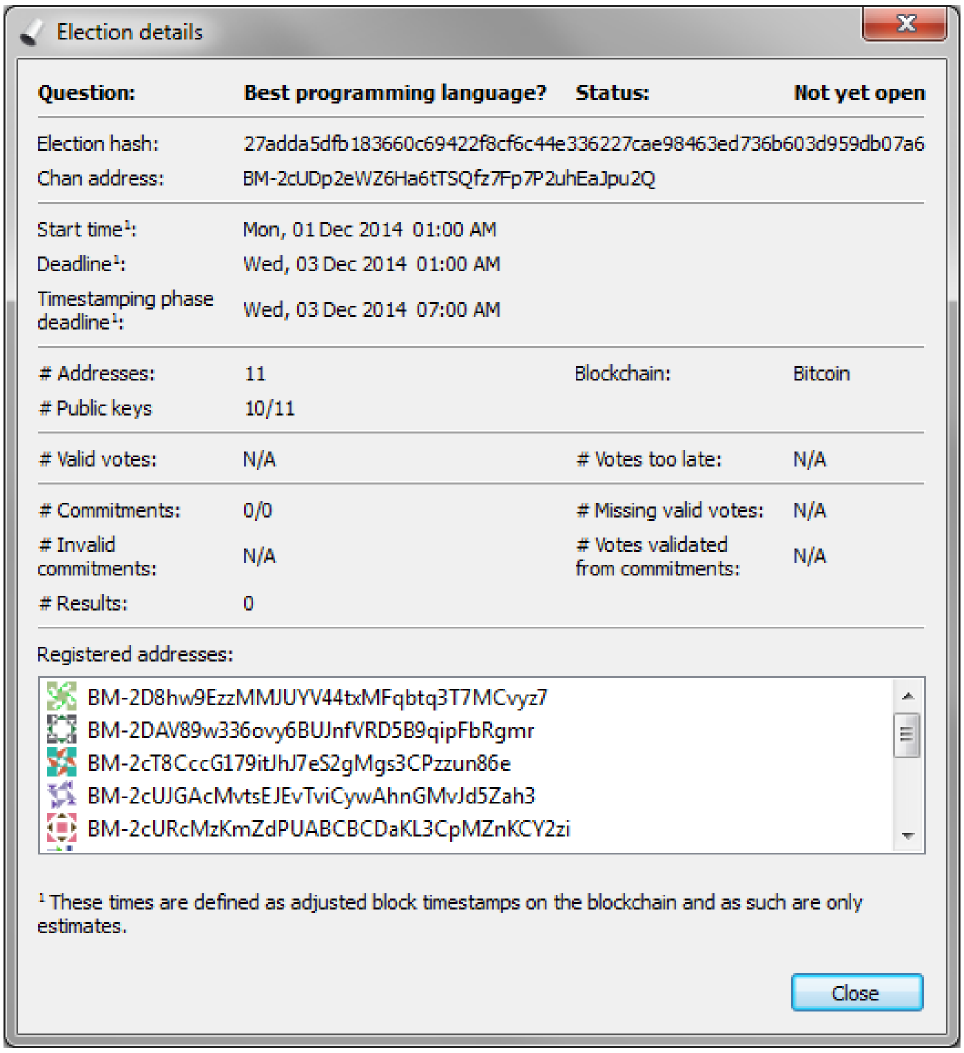
Status The current phase of the election. This text is the same as on the dashboard.

Election hash The unique hash that identifies this election. The text is selectable so you easily can copy the hash and give to others who want to join the election.

Chan address This is the address of the Bitmessage chan used as the shared bulletin board. The chan adress is computed from the above hash.

# Public keys shows the number of known and required public keys. If the first number is smaller than the second, we don't yet have all the required public keys and must wait for them to arrive.

# Valid votes shows the number of votes that the client has declared as being sent before the deadline, either by receiving them before the deadline or by timestampers having committed to them.


Election details

Question:	**Best programming language?**	**Status:**	**Not yet open**
Election hash:	27adda5dfb183660c69422f8cf6c44e336227cae98463ed736b603d959db07a6		
Chan address:	BM-2cUDp2eWZ6Ha6tTSQfz7Fp7P2uhEaJpu2Q		
Start time[1]:	Mon, 01 Dec 2014 01:00 AM		
Deadline[1]:	Wed, 03 Dec 2014 01:00 AM		
Timestamping phase deadline[1]:	Wed, 03 Dec 2014 07:00 AM		
# Addresses:	11	Blockchain:	Bitcoin
# Public keys	10/11		
# Valid votes:	N/A	# Votes too late:	N/A
# Commitments:	0/0	# Missing valid votes:	N/A
# Invalid commitments:	N/A	# Votes validated from commitments:	N/A
# Results:	0		

Registered addresses:

- BM-2D8hw9EzzMMJUYV44txMFqbtq3T7MCvyz7
- BM-2DAV89w336ovy6BUJnfVRD5B9qipFbRgmr
- BM-2cT8CccG179itJhJ7eS2gMgs3CPzzun86e
- BM-2cUJGAcMvtsEJEvTviCywAhnGMvJd5Zah3
- BM-2cURcMzKmZdPUABCBCDaKL3CpMZnKCY2zi

[1] These times are defined as adjusted block timestamps on the blockchain and as such are only estimates.

Figure 19: The election details dialog

# **Votes too late** shows the number of votes that the client has declared as being sent *after* the deadline. If a commitment for some of these votes is processed, those votes are moved to the valid votes.

# **Commitments** shows the number of commitments that are validated and processed and the total number received. Commitments aren't validated or processed before the results phase, so the first number will always be 0 until the results phase starts. Ideally all commitments will be both validated and processed, so the first number is the same as the last.

# **Invalid commitments** shows the number of commitments that are declared invalid because they couldn't be verified on the blockchain.

# **Results** shows the number of results messages received.

# **Missing valid votes** shows the number of votes which we know are valid based on commitments from timestampers, but we haven't (yet) received.

# **Votes validated from commitments** is the number of votes which the client originally declared as being sent too late, but where commitments have proved that they were indeed sent in time.

A.1.6 Timestamper settings

The timestamper settings dialog, which was briefly shown in Section 5.3, allows you to choose whether or not to volunteer as a timestamper, and, if so, which Bitcoin address to use when committing to the blockchain. The dialog is shown again in Figure 20.

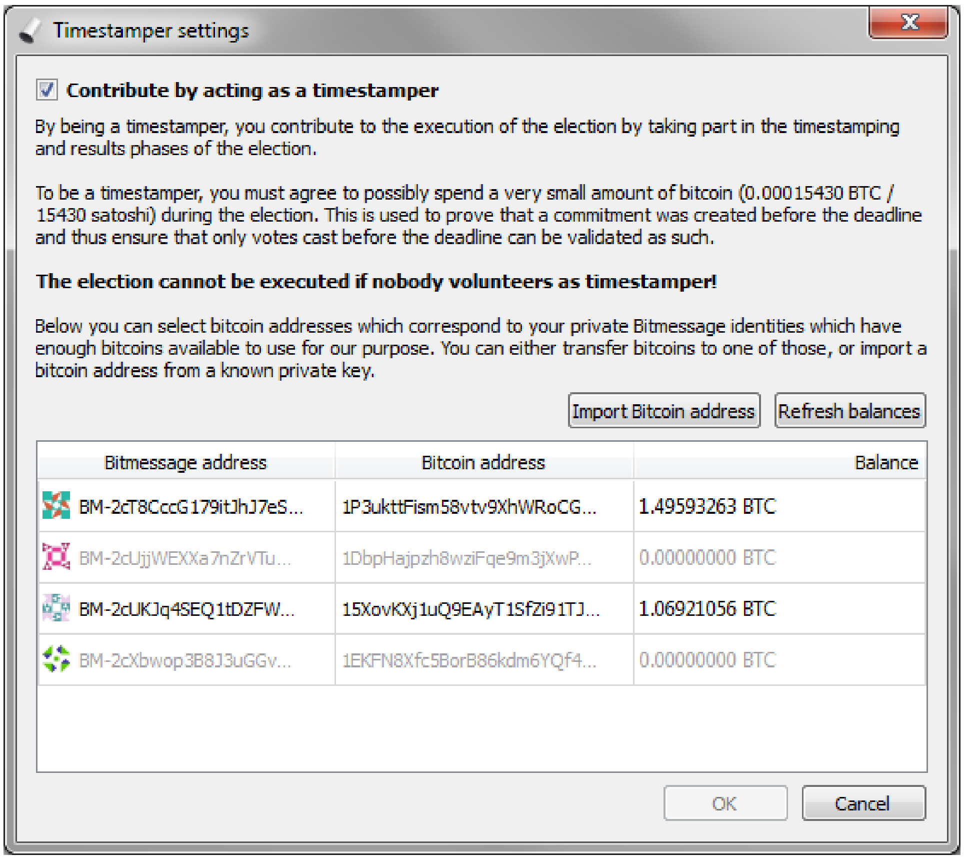
In addition to showing the addresses, the dialog also shows the current balance of each address and allows you to import a Bitcoin private key by pressing the "Import Bitcoin address". You can choose to provide a private key either as a 64-character hexadecimal number or as Bitcoin Wallet import format[111] (WIF).

As the dialog points out with the bold text, an election cannot be executed if nobody volunteers as timestampers.

A.1.7 Election results

When an election is over (the commitment deadline has been reached), the voting dashboard changes slightly and shows two more lists, namely "All results" and "Result details". The first list shows an overview of all election

[111] This format is the one used, e.g., when dumping a Bitcoin private key with the `dumpprivkey` tool. The technical description for WIF can be found here: `https://en.bitcoin.it/wiki/Wallet_import_format`, visited 2014-10-30



[✓] **Contribute by acting as a timestamper**

By being a timestamper, you contribute to the execution of the election by taking part in the timestamping and results phases of the election.

To be a timestamper, you must agree to possibly spend a very small amount of bitcoin (0.00015430 BTC / 15430 satoshi) during the election. This is used to prove that a commitment was created before the deadline and thus ensure that only votes cast before the deadline can be validated as such.

The election cannot be executed if nobody volunteers as timestamper!

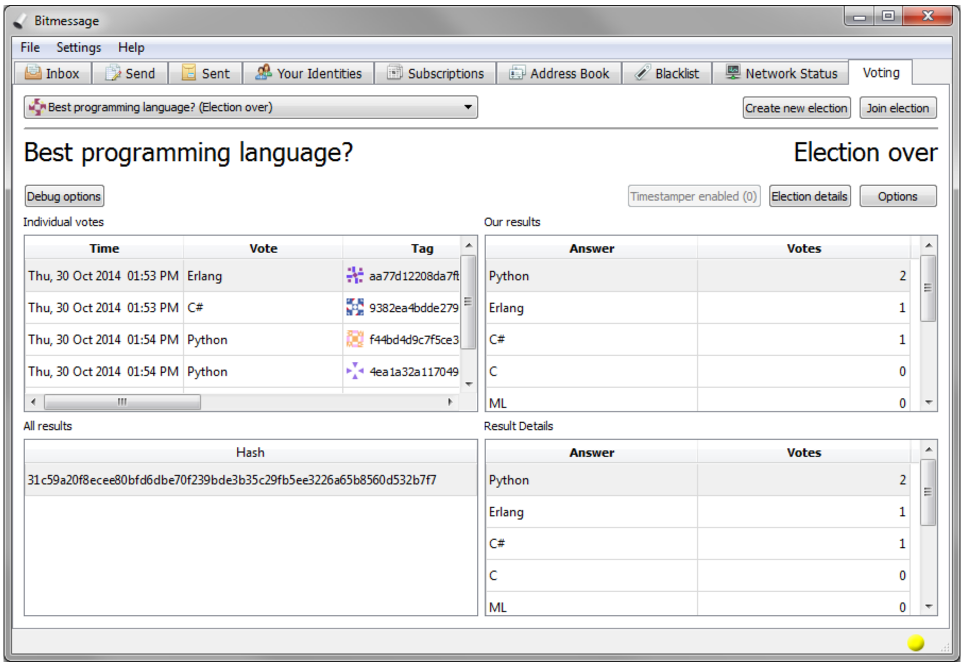
Below you can select bitcoin addresses which correspond to your private Bitmessage identities which have enough bitcoins available to use for our purpose. You can either transfer bitcoins to one of those, or import a bitcoin address from a known private key.

Bitmessage address	Bitcoin address	Balance
BM-2cT8CccG179itJhJ7eS...	1P3ukttFism58vtv9XhWRoCG...	1.49593263 BTC
BM-2cUjjWEXXa7nZrVTu...	1DbpHajpzh8wziFqe9m3jXwP...	0.00000000 BTC
BM-2cUKJq4SEQ1tDZFW...	15XovKXj1uQ9EAyT1SfZi91TJ...	1.06921056 BTC
BM-2cXbwop3B8J3uGGv...	1EKFN8Xfc5BorB86kdm6YQf4...	0.00000000 BTC

Figure 20: The timestamper settings dialog allowing you to configure your client as a timestamper.

results posted to the bulletin board (as well as our own), and clicking one of them shows the result in the second list.

NB: *The results can be faked by anyone and should not be trusted.* However, they can be used to get an overview of other clients having arrived at different results, if any.


Bitmessage
File Settings Help
Inbox Send Sent Your Identities Subscriptions Address Book Blacklist Network Status Voting
Best programming language? (Election over)
Create new election
Join election
Debug options
Timestamper enabled (0)
Election details
Options

Best programming language?

Election over

Individual votes

Time	Vote	Tag
Thu, 30 Oct 2014 01:53 PM	Erlang	aa77d12208da7ft
Thu, 30 Oct 2014 01:53 PM	C#	9382ea4bdde279
Thu, 30 Oct 2014 01:54 PM	Python	f44bd4d9c7f5ce3
Thu, 30 Oct 2014 01:54 PM	Python	4ea1a32a117049

Our results

Answer	Votes
Python	2
Erlang	1
C#	1
C	0
ML	0

All results

Hash
31c59a20f8ecee80bfd6dbe70f239bde3b35c29fb5ee3226a65b8560d532b7f7

Result Details

Answer	Votes
Python	2
Erlang	1
C#	1
C	0
ML	0

Figure 21: The voting dashboard as it looks when an election is over. Notice the new lists “All results” and “Result details”

A.2 Pseudocode for consensus protocol

The next pages show pseudocode for the following functions:

- ReceiveMessage(*message*): Runs when an ordinary message is received. Decides whether or not to store the message and how to store it.
- PostTimestampCommitment(): Runs when the posting phase is over, in order to construct a timestamped commitment to post to the other peers.
- ValidateCommitmentMessages(): Runs when the timestamping phase is over, and validates all received IBLT's.
- ProcessCommitmentMessages(): Runs immediately after ValidateCommitmentMessages, and processes all valid IBLT's by extracting as many hashes as possible from them and then using these hashes to accept messages as being sent before the deadline.

In these functions, we have the following lists available (which are all empty at the beginning):

- M: Messages which are accepted as being before the deadline, either by us locally, or by a valid commitment by another peer.
- M_{Late}: Messages which are marked as received after the deadline.
- $IBLTS_{Unvalidated}$: IBLT's from other peers which have only been received, and not yet validated.
- $IBLTS_{Incomplete}$: IBLT's from other peers which have been validated, and are not processed completely.
- $MH_{Accepted}$: List of message hashes which have valid commitments (through IBLT's)

And this function which is not directly part of the consensus protocol, but instead a part of the Message Interpretation Layer (MIL) on top of the consensus protocol:

- MessageValid(*message*): Decide if the message is valid or not based on custom rules, e.g., validate linkable ring signature.

We also have the following three functions which are used to commit to and check commitments on the Bitcoin blockchain:

- ComputeAddress(*Content*): Used to deterministically compute a Bitcoin address from the provided content.

- COMMITTOBTC(*Address*): Used to make a commitment to the blockchain by sending a small amount to the provided Bitcoin address.
- GETADDRESSFIRSTSEEN(*Address*): Used to check when (if ever) a Bitcoin address has first received an amount of currency.

Algorithms

```
function ReceiveMessage($message$)
    if MessageValid($message$) then
        if $Time > PostDeadline \wedge message \notin MH_{Accepted}$ then
            $M_{Late} \leftarrow M_{Late} \cup [message]$        ▷ Mark message as late
        else
            $M \leftarrow M \cup [message]$        ▷ Accept message
        end if
    end if
end function
```

Figure 22: Algorithm for processing incoming messages from posters. This method is run whenever a new message is received. This method is implemented in part of the `received_message(message)` method in `consensus_protocol.py`.

```
function PostTimestampCommitment()
    $IBLT \leftarrow$ CreateIBLT($M$)
    $Addr \leftarrow$ ComputeAddress($IBLT$)
    CommitToBTC($Addr$)
    PostMessage($TimestampCommitment, IBLT$)
end function
```

Figure 23: Algorithm for timestamping valid messages and posting an IBLT with the hashes of those messages. This method is run when the posting phase ends and the timestamping phase begins. It is implemented as the method `do_commitment()` in `consensus_protocol.py`

```
function ValidateCommitmentMessages
    for all $IBLT$ in $IBLTS_{Unvalidated}$ do
        $IBLTS_{Unvalidated} \leftarrow IBLTS_{Unvalidated} \setminus IBLT$    ▷ Remove it first
        $Addr \leftarrow$ComputeAddress($IBLT$)
        $FirstSeen \leftarrow$GetAddressFirstSeen($Addr$)
        if $FirstSeen$ is NULL then
            continue                                        ▷ Never timestamped
        end if
        $TxBlockNo, TxConfirmations \leftarrow FirstSeen$
        if $TxBlockNo \geq TimestampingPhaseEndBlockNo$ then
            continue                                        ▷ Timestamped too late
        else if $TxConfirmations \leq MIN\_CONFIRMATIONS$ then
            continue                                        ▷ Too few confirmations
        end if
        $IBLTS_{Incomplete} \leftarrow IBLTS_{Incomplete} \cup IBLT$          ▷ Validated
    end for
end function
```

Figure 24: Algorithm for validating all received IBLTs. This algorithm is run when the timestamping phase ends and the results phase begins. It is implemented as the method `validate_commitment_messages()` in `consensus_protocol.py`.

```
function ProcessCommitmentMessages()
    repeat
        $Hashes_{new} \leftarrow \emptyset$
        for all $IBLT$ in $IBLTS_{Incomplete}$ do
            $IBLT \leftarrow$Copy($IBLT$)                              ▷ Create a copy
            for all $Hash$ in $MH_{Accepted}$ do
                Delete($IBLT, Hash$)              ▷ Subtract valid message hashes
            end for
            $Result_{IBLT},\ Hashes_{IBLT} \leftarrow$ListEntries($IBLT$)
            $Hashes_{IBLT_{new}} \leftarrow Hashes_{IBLT} - MH_{Accepted}$
            if $Hashes_{IBLT_{new}} = \emptyset$ then
                continue                                         ▷ Try next IBLT
            end if
            $Hashes_{new} \leftarrow Hashes_{new} \cup Hashes_{IBLT_{new}}$
            if $Result_{IBLT} = Complete$ then
                $IBLTS_{Incomplete} \leftarrow IBLTS_{Incomplete} \setminus IBLT$
                                            ▷ IBLT is now completely processed
            end if
        end for
        if $Hashes_{new} \neq \emptyset$ then
            $MH_{Accepted} \leftarrow MH_{Accepted} \cup Hashes_{new}$
        else
        end if
    until $IBLTS_{Incomplete} = \emptyset$ or $Hashes_{new} = \emptyset$

    for all $m$ in $M_{Late}$ do
        if $m$.Hash()$\in Entries_{IBLT}$ then
            $M_{Late} \leftarrow M_{Late} \setminus m$              ▷ Move $m$ from $M_{Late}$ to $M$
            $M \leftarrow M \cup m$
        end if
    end for
end function
```

Figure 25: Algorithm for processing all validated IBLTs. This algorithm is run immediately after ValidateCommitmentMessages. It is implemented as the method `process_commitment_messages()` in `consensus_protocol.py`.

A.3 Py-EC

Py-EC is a Python wrapper of the OpenSSL elliptic curve functions. It can be found on GitHub[112]. The rest of this section lists the contents of the README file:

In Python, dealing directly with the OpenSSL library (through PyElliptic [113]) easily becomes a hassle with the use of C pointers and string buffers.

To make things easier, I decided to make a wrapper for PyElliptic to make the manipulation of elliptic curves and points more Pythonic.

The wrapper has been tested with all recommended SEC curves (`secp192k1`, `secp192r1`, `secp224k1`, `secp224r1`, `secp256k1`, `secp256r1`, `secp384r1`, `secp521r1`, `sect163k1`, `sect163r1`, `sect163r2`, `sect233k1`, `sect233r1`, `sect239k1`, `sect283k1`, `sect283r1`, `sect409k1`, `sect409r1`, `sect571k1` and `sect571r1`).

Especially point addition and multiplication is way easier, as the following console example usage shows:

A.3.1 Example use

```
>>> from curve import Curve
>>> c = Curve( 'secp256k1' )

>>> c
Curve<Equation: y^2 = x^3+7 (mod p), Field: Prime field, p:
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F>

>>> c.p
115792089237316195423570985008687907853269984665640564039457584007908834671663L

>>> c.a
0

>>> c.b
7

>>> c.order
115792089237316195423570985008687907852837564279074904382605163141518161494337L

>>> c.G
Point<0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798,
0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8>

>>> c.G.x
55066263022277343669578718895168534326250603453777594175500187360389116729240L
```

[112]https://github.com/jesperborgstrup/Py-EC
[113]https://github.com/yann2192/pyelliptic

```
>>> c.G.y
32670510020758816978083085130507043184471273380659243275938904335757337482424L

>>> 4 * c.G + (255 * c.G)
Point<0xC2C80F844B70599812D625460F60340E3E6F36054A14546E6DC25D47376BEA9B,
0x86CA160D68F4D4E718B495B891D3B1B573B871A702B4CF6123ABD4483AA79C64>

>>> from keypair import KeyPair
>>> kp = KeyPair( c )

>>> kp
KeyPair<Private:0x5091AD80EEE3FB065A6E3FF126A112C4905F8E79566E22396807A55ADE1B5C6F,
Public:Point<0x13FCF42341462150B8366F11659E396DF88D19F65D533CEEAC78C9EC6F94B45D,
0x18DDDF6DCA0C097FC0359E680BAED36403D77657ABE7F76E64E1B787D90C485A>>

>>> kp.private_key
36442418189203456142546292588071998273845228785350611568921618467649899682927L

>>> kp.public_key
Point<0x13FCF42341462150B8366F11659E396DF88D19F65D533CEEAC78C9EC6F94B45D,
0x18DDDF6DCA0C097FC0359E680BAED36403D77657ABE7F76E64E1B787D90C485A>
```

A.3.2 Curves

Getting a curve instance A curve can be initialized in three ways; by its name, id or by a pointer to an OpenSSL `EC_GROUP` instance:

```
>>> from curve import Curve

>>> Curve( curvename='secp256k1' )
Curve<Equation: y^2 = x^3+7 (mod p), Field: Prime field, p:
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F>

>>> Curve( curveid=714 )
Curve<Equation: y^2 = x^3+7 (mod p), Field: Prime field, p:
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F>

>>> from pyelliptic.openssl import OpenSSL
>>> Curve( openssl_group=OpenSSL.EC_GROUP_new_by_curve_name( 714 ) )
Curve<Equation: y^2 = x^3+7 (mod p), Field: Prime field, p:
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F>
```

Properties of a curve Depending on whether the curve is over a prime field, F_p, or a power-of-2 field, F_{2_m} , the curve has slightly different properties:

- `prime_type`: Either `'prime'` or `'power-of-two'`
- `G`: The base `Point` (or generator) of the curve.
- `order`: The order of the curve (number of elements)

- `h`: The cofactor of the curve
- `a`: The curve coefficient a
- `b`: The curve coefficient b
- `p` **(Only** F_p**)**: The prime p specifying the field
- `m` **(Only** F_{2_m}**)**: The integer m specifying the field
- `poly_coeffs` **(Only** F_{2_m}): The degrees of the polynomials specifying the field
- `os_group`: A pointer to the underlying `EC_GROUP` instance.

```
>>> from curve import Curve
>>> c1 = Curve( 'secp256k1' )
>>> c2 = Curve( 'sect239k1' )

>>> c1
Curve<Equation: y^2 = x^3+7 (mod p), Field: Prime field, p:
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F>
>>> c2
Curve<Equation: y^2+xy = x^3+1, Field: Power-of-two field, f(x): x^239+x^158+1>

>>> c1.field_type
'prime'
>>> c2.field_type
'power-of-two'

>>> c1.G
Point<0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798,
0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8>
>>> c2.G
Point<0x29A0B6A887A983E9730988A68727A8B2D126C44CC2CC7B2A6555193035DC,
0x76310804F12E549BDB011C103089E73510ACB275FC312A5DC6B76553F0CA>

>>> c1.order
115792089237316195423570985008687907852837564279074904382605163141518161494337L
>>> c1.h
1
>>> c1.a
0
>>> c1.b
7

>>> c1.p
115792089237316195423570985008687907853269984665640564039457584007908834671663L

>>> c2.m
239
>>> c2.poly_coeffs
[158]
```

A.3.3 Points

Getting a point instance You can get the base point from the `G` property of a curve as described above:

```
>>> from curve import Curve
>>> Curve( 'secp256k1' ).G
Point<0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798,
0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8>
```

You can also create a point on a curve from either (1) the x and y coordinates of the point or (2) by a pointer to an OpenSSL `EC_POINT` instance:

```
>>> from curve import Curve
>>> from point import Point
>>> c = Curve( 'secp256k1' )

>>> Point( c, x=255, y=255 ) # Invalid coordinates, only a demonstration
Point<0xFF, 0xFF>

>>> from pyelliptic.openssl import OpenSSL
>>> Point( c, openssl_point=OpenSSL.EC_POINT_new( c.os_group ) )
Point<0x0, 0x0>
```

Finally, you can hash a string directly onto a curve (using the 'try-and-increment' method for finding points close to a certain x coordinate):

```
>>> from curve import Curve
>>> c = Curve( 'secp256k1' )

>>> c.hash_to_point( 'somestring' )
Point<0xE4998BB769D5AF19526738527E13ECF753F5CC7AA60DD0ADF94BB0A248CF577A,
0x79FCD45DD59999C5D916FB31C0F023B4A1A1BCD63F11FD3D3E31D5C5E7D79C1D>

>>> c.hash_to_point( 'someotherstring' )
Point<0xB661EE62474532EF1C8EA78B1CE3634E2EEC06B8E256E46A5CE25DF0FFABF332,
0x1DAB745A01B745CA9BF276D8E990E8EF11CFA954C5956DF9BF4C0684FABB00A6>
```

Performing arithmetics Point addition and multiplication is intuitive:

```
>>> from curve import Curve
>>> c = Curve( 'secp256k1' )

>>> c.G
Point<0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798,
0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8>

>>> 2 * c.G
Point<0xC6047F9441ED7D6D3045406E95C07CD85C778E4B8CEF3CA7ABAC09B95C709EE5,
0x1AE168FEA63DC339A3C58419466CEAEEF7F632653266D0E1236431A950CFE52A>
```

```
>>> c.G + c.G
Point<0xC6047F9441ED7D6D3045406E95C07CD85C778E4B8CEF3CA7ABAC09B95C709EE5,
0x1AE168FEA63DC339A3C58419466CEAEEF7F632653266D0E1236431A950CFE52A>

>>> ( 5 * c.G ) + ( 256 * c.G )
Point<0x9CF606744CF4B5F3FDF989D3F19FB2652D00CFE1D5FCD692A323CE11A28E7553,
0x8147CBF7B973FCC15B57B6A3CFAD6863EDD0F30E3C45B85DC300C513C247759D>
```

Properties of a points

- `x`: The x coordinate
- `y`: The y coordinate
- `os_point`: A pointer to the underlying `EC_POINT` instance.

A.3.4 Key pair

Getting a key pair instance A key pair for a curve can be generated randomly or by providing a private key:

```
>>> from curve import Curve
>>> from keypair import KeyPair
>>> c = Curve( 'secp256k1' )

>>> KeyPair( c ) # Random key pair
KeyPair<Private:0x94087552C3C72CC867E555854B9DD6392A611A40C168B0C6B7AEFC63DD9F5818,
Public:Point<0x7C3FF4B9AE4D4EFCD22185F5ED7B6C8EF79CFF83AC0A3DFA4A258CDDBFC2AC3E,
0xEBFD9904CB8398524022BCDC268D6B03207737F35E7591EE5ACEE338D5272733>>

>>> KeyPair( c, private_key=12345 )
KeyPair<Private:0x3039,
Public:Point<0xF01D6B9018AB421DD410404CB869072065522BF85734008F105CF385A023A80F,
0xEBA29D0F0C5408ED681984DC525982ABEFCCD9F7FF01DD26DA4999CF3F6A295>>
```

Alternatively, you can provide a pointer to an OpenSSL `EC_KEY` instance:

```
>>> from curve import Curve
>>> from keypair import KeyPair
>>> from pyelliptic.openssl import OpenSSL
>>> c = Curve( 'secp256k1' )
>>> k = OpenSSL.EC_KEY_new_by_curve_name( 714 )
>>> OpenSSL.EC_KEY_generate_key( k )
1

>>> KeyPair( c, os_key=k )
KeyPair<Private:0xECBCB11DB69B0A8876986571E336A4F486E7B2C355712D2FA32C9836A153AAA,
Public:Point<0x732F6911AC325F41CEAB478D4D5AE3EB033A06EA8ECC03AF58CF2FF022A1FE5,
0x156B3C906BB70070B946F8565C425FEA00EA3350A71073F5B4818C96D41610C6>>
```

Properties of a key pair

- `private_key`: The private key (an integer)
- `public_key`: The public key (a `Point`)
- `os_key`: A pointer to the underlying `EC_KEY` instance.

A.4 Py-IBLT

Py-IBLT is a Python implementation of Invertible Bloom Lookup Tables, that I created in order to use IBLT's in the voting protocol. It is licensed under the permissive MIT license and can be found on GitHub[114]. The rest of this section lists the contents of the README file:

A.4.1 Notes

- Records that are deleted without a corresponding insert operation can be recovered with the `get` and `list_entries` functions.
- Error detection using `hashValueSum` as described in the paper is currently not implemented.
- Duplicate keys are not supported.

A.4.2 Usage

Constructor Use the constructor to create a new IBLT:

```
>>> from iblt import IBLT
>>> t = IBLT( m, k, key_size, value_size, hash_key_sum_size=10, hash=None )
```

- `m` is the number of cells in underlying lookup table, and is closely related to the threshold value that determines how many key/value pairs the IBLT can hold before giving inconclusive answers to queries.
- `k` is the number of hash functions to be used.
- `key_size` is maximum size for keys.
- `value_size` is maximum size for values.
- `hash_key_sum_size` is number of bytes used for the hashkeySum field.
- `hash` is function(i, value), where i is index of hash function and value is value to be hashed (or `None` for default hash functions).

Insert and delete The `insert` function inserts a key/value pair into the IBLT.

```
>>> t.insert( key, value )
```

The `delete` function likewise deletes a key/value pair from the IBLT.

```
>>> t.delete( key, value )
```

The `insert` and `delete` functions do not have any return values.

[114]`https://github.com/jesperborgstrup/Py-IBLT`

Retrieving a value based on a key The `get` function tries to retrieve a value by key and gives you additional information on how the operation went:

```
>>> t.get( key )
```

For the `get` function, the return value is `( <Result>, <Value> )`. The `<Result>` value can be one of the following four possibilities:

- `IBLT.RESULT_GET_NO_MATCH` It is certain that no such key exists in the table. `<Value>` is `None`.
- `IBLT.RESULT_GET_MATCH` The key was previously inserted into the table and the value is returned in `<Value>`.
- `IBLT.RESULT_GET_DELETED_MATCH` The key was found, but it had been deleted instead of inserted and the value is returned in `<Value>`.
- `IBLT.RESULT_GET_INCONCLUSIVE` It wasn't possible to determine if the key was in the table or not. `<Value>` is `None`.

Listing entries The `list_entries\verb` function tries to extract a complete list of all entries in the table. It either returns all entries or an incomplete list:

```
>>> t.list_entries()
```

The return value of `list_entries` is `( <Result>, [<Entries>], [<Deleted entries>] )`.

The `<Result>` value is one of two possibilities:

- `IBLT.RESULT_LIST_ENTRIES_COMPLETE` The entries lists are complete.
- `IBLT.RESULT_LIST_ENTRIES_INCOMPLETE`: It wasn't possible to extract a complete listing. Only the returned entries were recoverable.

The `<Entries>` list contains all recoverable entries that have been inserted into the table. Likewise, the `<Deleted entries>` list contains all recoverable entries that have been deleted from the table without being inserted.

Serializing and unserializing Table instances have a `serialize` function, that takes no argument and serializes the data and metadata of the table for into a bitstring storage or communication.

Likewise, the IBLT class has a static `unserialize` function that takes the serialized bitstring and constructs a copy of the table from the bitstring.

```
# t is an IBLT instance
>>> s = t.serialize()

# Prints True
>>> print IBLT.unserialize( s ) == t
```

Serialized data format:

```
[ Magic bytes ][  Header  ][ Data ]
     4 bytes      24 bytes

Magic bytes:
0x49 0x42 0x4C 0x54 (ASCII for IBLT)

Header:
    [ Cell count (m) ]
         32-bit uint

    [ Key sum length ][ Value sum length ]
        32-bit uint       32-bit uint

    [ HashKeySum length ][ ValueKeySum length ]
        32-bit uint           32-bit uint

    [ # hash funcs (k) ]
        32-bit uint

Data:
    For each of the m cells:
        [  Count   ][ keySum ][ valueSum ][ hashKeySum ][ valueKeySum ]
          32-bit int
```

A.5 Bitcoin blocks with timestamps smaller than previous blocks

This list was created by downloading the complete blockchain from `https://bitcoin.org/bin/blockchain/` and running the following Java code. The Java code requires bitcoinj downloadable from `https://bitcoinj.github.io/`

Java code

```
import java.io.File;
import java.util.ArrayList;
import java.util.Deque;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;

import javax.xml.crypto.dsig.keyinfo.KeyValue;

import org.bitcoinj.core.Block;
import org.bitcoinj.core.NetworkParameters;
import org.bitcoinj.params.MainNetParams;
import org.bitcoinj.utils.BlockFileLoader;

public class MedianTimestampAnalyzer {
    public static void main( String[] args ) {
        NetworkParameters np = new MainNetParams();
        List<File> blockChainFiles = new ArrayList<>();
        blockChainFiles.add(new File("Q:/bootstrap.dat"));
        BlockFileLoader bfl = new BlockFileLoader(np, blockChainFiles);

        Deque<Long> previousTimestamps = new LinkedList<Long>();
        Map<Integer, List<Integer>> blockMap = new HashMap<Integer, List<Integer>>();

        int i = -1, blocksBack = 0;
        for ( Block block: bfl ) {
            i++;

            long blockTimestamp = block.getTimeSeconds();

            if ( previousTimestamps.size() < 20 ) {
                // Fill up queue with the first blocks
                previousTimestamps.addFirst( blockTimestamp );
                continue;
            }

            blocksBack = 0;
            for ( Long previousTimestamp: previousTimestamps ) {
                if ( blockTimestamp > previousTimestamp ) {
                    break;
                } else {
                    blocksBack++;
                }
            }

            if ( blocksBack > 0 ) {
                if ( !blockMap.containsKey( blocksBack ) ) {
                    blockMap.put( blocksBack, new LinkedList<Integer>() );
                }
                blockMap.get( blocksBack ).add( i );
            }

            // Update queue
            previousTimestamps.removeLast();
            previousTimestamps.addFirst( blockTimestamp );
        }

        for ( Entry<Integer, List<Integer>> entry: blockMap.entrySet() ) {
            System.out.println( "Blocks with timestamp smaller than " + entry.getKey() + " blocks back (" + entry.getValue().size() + "):" );
            for ( Integer block: entry.getValue() ) {
                System.out.print( block + "," );
            }
            System.out.println();
            System.out.println();
        }
    }
}
```

Timestamp smaller than 3 blocks back (105)

23993, 24078, 24622, 32650, 33271, 39218, 53471, 54108, 54232, 55346, 62834, 64489, 65618, 65806, 65885, 66018, 66169, 66176, 66215, 66270, 66403, 66449, 66454, 66461, 66617, 66636, 67591, 67612, 67699, 67805, 68984, 69650, 70071, 70927, 71527, 72670, 77911, 85649, 161226, 177233, 177568, 197765, 201167, 203326, 203678, 205536, 208411, 244773, 267843, 270033, 270728, 270977, 272304, 272816, 276342, 278075, 279210, 280048, 282094, 282229, 284190, 284504, 284523, 284541, 289121, 290124, 291305, 291730, 293110, 293831, 294040, 294264, 294631, 295969, 296283, 296484, 296604, 297363, 297728, 297778, 297889, 299214, 299278, 299312, 299404, 299547, 300042, 300419, 303358, 304919, 305147, 305753, 306112, 306383, 307360, 309110, 309581, 310932, 311194, 311266, 311475, 312750, 315073, 316331, 316596

Timestamp smaller than 4 blocks back (41)

24068, 24176, 58910, 59335, 59373, 59959, 60823, 61550, 62651, 62808, 63431, 63773, 63814, 64041, 65665, 65817, 65845, 65946, 66108, 66288, 66297, 66339, 66351, 66956, 67381, 67660, 68463, 68686, 68997, 69330, 70871, 271374, 280512, 281485, 284176, 292848, 295902, 297805, 298315, 303390, 314626

Timestamp smaller than 5 blocks back (55)

24157, 32634, 38084, 38138, 38146, 44054, 46091, 46100, 46362, 56945, 57060, 60025, 60131, 60169, 61492, 61563, 61769, 61928, 61970, 62557, 63637, 63843, 63936, 63958, 63971, 64499, 64545, 65704, 65796, 65895, 65964, 66012, 66071, 66476, 66504, 66619, 66996, 67276, 67359, 67409, 67445, 67694, 67795, 67814, 68021, 68103, 68115, 68754, 83398, 86903, 155101, 156368, 196493, 278850, 316455

A.6 Proof-of-work average times

This section shows the raw data from computing proof-of-work for votes with a varying amount of voters. The amount of voters decide the size of the vote as shown in table 13 on page 83.

In the below data, `AV` is approximate the amount of voters[115], `PL` shows the payload length, the vote size, `Target` shows the target value which the proof-of-work must be smaller than, `Log2(target)` is the binary logarithm of the target value. Finally, `seconds` is the average amount of seconds it took to compute the proof-of-work over 20 tries on my Lenovo Thinkpad E420s.

```
AV:     2,  PL:     169, Target:  15728096840420, Log2(target): 43.838409, seconds: 4.733
AV:     2,  PL:     195, Target:  15379336277000, Log2(target): 43.806058, seconds: 3.241
AV:     4,  PL:     233, Target:  14911715495905, Log2(target): 43.761511, seconds: 5.350
AV:     5,  PL:     286, Target:  14296943048983, Log2(target): 43.700772, seconds: 2.772
AV:     8,  PL:     361, Target:  13509291114218, Log2(target): 43.619017, seconds: 4.011
AV:    11,  PL:     467, Target:  12532829268770, Log2(target): 43.510777, seconds: 6.348
AV:    16,  PL:     617, Target:  11370528884633, Log2(target): 43.370365, seconds: 4.912
AV:    22,  PL:     829, Target:  10052142008875, Log2(target): 43.192568, seconds: 7.369
AV:    32,  PL:    1129, Target:   8636047537084, Log2(target): 42.973508, seconds: 6.597
AV:    45,  PL:    1553, Target:   7201344005439, Log2(target): 42.711403, seconds: 9.855
AV:    64,  PL:    2153, Target:   5831317858056, Log2(target): 42.406959, seconds: 8.323
AV:    90,  PL:    3001, Target:   4595032145708, Log2(target): 42.063212, seconds: 12.641
AV:   128,  PL:    4201, Target:   3535117324832, Log2(target): 41.684895, seconds: 16.852
AV:   181,  PL:    5897, Target:   2665578639368, Log2(target): 41.277586, seconds: 32.285
AV:   256,  PL:    8297, Target:   1977642810202, Log2(target): 40.846919, seconds: 42.094
AV:   362,  PL:   11690, Target:   1448841694425, Log2(target): 40.398037, seconds: 49.702
AV:   512,  PL:   16489, Target:   1051297684627, Log2(target): 39.935308, seconds: 64.725
AV:   724,  PL:   23275, Target:    757395898616, Log2(target): 39.462257, seconds: 85.839
AV:  1024,  PL:   32873, Target:    542796481163, Log2(target): 38.981620, seconds: 115.876
AV:  1448,  PL:   46445, Target:    387517695437, Log2(target): 38.495471, seconds: 167.168
AV:  2048,  PL:   65641, Target:    275898398980, Log2(target): 38.005346, seconds: 194.379
AV:  2896,  PL:   92786, Target:    196041720131, Log2(target): 37.512370, seconds: 420.670
AV:  4096,  PL:  131177, Target:    139102455090, Log2(target): 37.017357, seconds: 371.777
AV:  5792,  PL:  185468, Target:     98601724265, Log2(target): 36.520894, seconds: 461.694
AV:  8192,  PL:  262249, Target:     69843172078, Log2(target): 36.023400, seconds: 806.083
AV: 11585,  PL:  370832, Target:     49447372827, Log2(target): 35.525175, seconds: 1174.644
```

[115] The set of amount of voters is calculated as $\{2^{x/2} \mid x \in [2, 28]\}$