Bern University
of Applied Sciences

Bachelor Thesis

An Android Client for Bitmessage

Author Christian Basler
Advisor Kai Brünnler
Expert Daniel Voisard
Date January 1, 2016

Contents

Abstract

If you don't want anyone snooping in your e-mails you might already encrypt your correspondence. But you still can't hide who you're writing to, and your e-mail client might even reveal much more about you, your computer, and the software you use. Bitmessage attempts to solve all this, but up until now there was no practical way to use it on mobile phones, which might be a downside when you're on the run.

Bitmessage provides some unique challenges for mobile clients. Its users tend to be bordering on paranoia, or as often might really be tracked. The protocol is as wasteful as a hen; it needs both immense amounts of traffic and a lot of CPU time. It works by distributing every message to every client, so they can pick up the ones they can decrypt with the available private keys. To protect the network from malicious flooding, a proof of work is required, which is done by calculating a partial hash collision. So typically you'd want as many CPU cores as possible, no shortage of electricity and a flat rate on internet access — not quite the perfect basis for a mobile app.

Apart from those protocol specific difficulties, Android™ has some challenges of its own. For example, to preserve memory and battery power, the operating system might kill practically any process at any time, especially those that just sit in the background and process incoming network objects. Then there are some major Java dependencies missing in the Android VM, most notably JDBC, widely used for accessing databases. And finally, there are a lot of very different devices out there, some with processors hardly fast enough to browse the web and others as strong as some desktop computers.

There were two main optimisations, both optional. Firstly, Android provides a highly optimized method to synchronize data with a server. Fortunately it was possible to leverage this while still using the official Bitmessage protocol. Secondly, for weaker phones an option to let a server do the proof of work was added. For both options the user must give up some of his anonymity towards the server — everything has its price.

You can get more information about the app at https://dissem.ch/abit

Trademarks

Android is a trademark of Google Inc.
Oracle and Java are registered trademarks of Oracle and/or its affiliates.
Other names may be trademarks of their respective owners.

Abbreviations

CPU	central processing unit
GPU	graphics processing unit
POW	proof of work
OS	operating system
PGP	Pretty Good Privacy
MIME	multipurpose internet mail extensions
S/MIME	Secure/MIME
JDBC	Java database connectivity
SQL	structured query language
NoSQL	Not only SQL
API	application programming interface
DOS	denial of service

1 Introduction

1.1 What is Metadata?

Metadata is information about data we create or access. This could be the websites we visit, what tv programmes we watch, or the sender, recipient, time, which mail client was used, maybe even where it was sent from and what anti-virus is used. The volume and quality of metadata greatly depends on the kind of data and the software used to create it.

While encryption technology like Pretty Good Privacy (PGP) or S/MIME provides a secure way to protect content from prying eyes, it can't hide the header information: sender, recipient, subject and possibly much more.

Ever since the revelations of whistleblower Edward Snowden we learned that metadata — most notably information about who communicates with whom — is equally interesting and much easier to analyse than the actual content.[5]

1.2 How Can We Hide Metadata?

With e-mail, all metadata is plain text, even for encrypted messages. We might be able to encrypt the connection to the e-mail provider, and they might or might not encrypt their connections to other providers. We can only hope that both our and the recipient's e-mail provider are both trustworthy and competent. Can we really expect that from something we get for free? And can we be sure they're not forced to release what they know to some agency?[2]

With Bitmessage we send a message to a sufficiently large number of participants, with the intended recipient among them. Content is encrypted such that only the person in possession of the private key can decrypt it. All participants try to do this in order to find their messages.

1.3 What is Bitmessage

Bitmessage is a peer to peer protocol building a mesh network among the participating clients. Every client tries to maintain multiple connections to other network nodes and has a full copy of every current object.

Objects are encrypted using a public key. Every client tries to decrypt each object using its private keys, processing the ones where it succeeds.

1.3.1 Advantages

A big advantage of Bitmessage is its inherent key management. The address contains a hash of the public key, and retrieving said key is an integral part of the protocol.

And the selling point of course is that everything is encrypted and signed, and there is next to no metadata an attacker could use.

.

.

TODO

1.3.2 Disadvantages

Of course the protocol uses a lot of resources. The traffic problem could be somewhat managed by splitting it into streams[6], but the POW is required in order to protect the network.

As you might guess, the protocol doesn't scale well. As the user base grows and traffic increases, it might be too much for weaker clients (read mobile phones) to process. Again, streams are said to be the solution, but if there's a big surge in network usage the implementations might not be ready. Unfortunately for this project, a successful mobile client might be responsible for that surge.

If somehow your private key gets into wrong hands, they might be able to read every message you ever received, even if you deleted them locally. Nobody can prevent an entity from collecting all encrypted objects just in case the might be of value some day. They can't read the messages you've *sent* though, unless they got your recipient's private key as well.

1.3.3 Risks

As when you decide to use Tor, a software for enabling anonymous communication formerly known as 'The Onion Router', you might be flagged by some security agency as a possible threat.[8] By itself this isn't necessarily a problem, but if you're a journalist it might be a risk for your informants, and combined with oder red flags you might be prevented from flying in or into the U.S.A.

.

.

TODO

1.4 Current state – what is missing?

Until recently there was no mobile client for Bitmessage, and the client that turned up since is very wasteful to the devices resources, draining the battery in little time. The alternative is to use an e-mail relay server, but this means to give up the private key to this server and end-to-end encryption is much more difficult to achieve. Therefore this might not be a viable option, especially

if you can't run your own server.

1.5 How should it be?

We need mobile Bitmessage clients that allows the user to choose their levels of convenience, privacy and resource hunger. There will always be trade-offs between needed traffic, battery use and privacy, and for each user the answer might look slightly different.

1.6 Why is it hard to do?

Bitmessage is very wasteful with resources by design. All messages are being sent to and stored on all nodes, and to protect the network POW is required for all objects that are distributed, meaning some very CPU-heavy calculations need to be done. The protocol wasn't developed with mobile users in mind, and while smartphones are getting increasingly powerful, there is at least the issue of battery use to watch out for, and many users have limited traffic on their data plan.

1.7 Why me, and how do I intend to do it?

I have seven years of experience developing Java applications, and was programming Android apps from the moment I had my "Android Dev Phone 1". As I developed Jabit, a Java implementation of the Bitmessage client, as my last project, I also have great knowledge about the Bitmessage protocol.

There are a few optimisations that I intend to do:

- Connect to only one reliable node instead of eight random nodes. This should reduce battery usage, but yields some risk if the node is compromised. Also, the node must forward all messages to all connected mobile clients instead of the default eight random nodes.
- Don't save objects we can't decrypt. We can solely save their hashes, but this means we're using the network without supporting it. This also might be an attack vector.
- Only connect to the network if we're on Wi-Fi and charging. This means of course that we'll only receive messages when we're connected with a Wi-Fi and charging.

Of course every option has its own drawbacks, so they will be configurable. As for the POW: Jabit highly optimises its calculation, which might be enough for modern smartphones.

Further optimisations might introduce a server component that *might* do

- proof of work
- Request public keys, requiring us to give up some anonymity towards the server.
- Inform the client about new messages sent to its addresses. This would mean to give up our

anonymity towards the server in the best case (which isn't supported by the protocol yet), towards the whole network (which is somewhat supported), or give up the private key to the server (which, for many users, is unacceptable).

To find out what is actually necessary, a naive impementation will be done first.

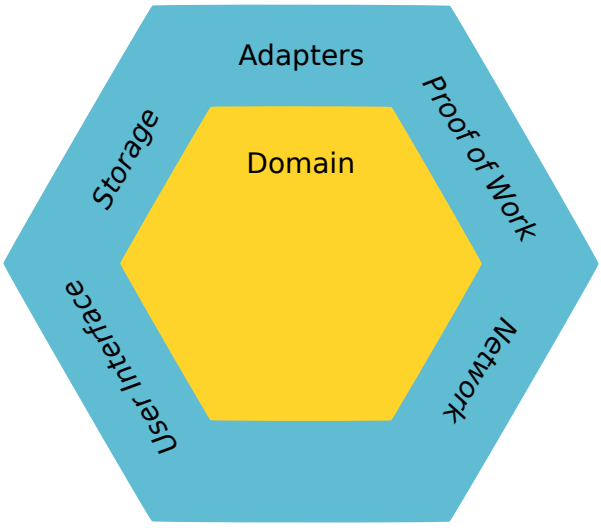
1.8 The Jabit Bitmessage Library

Jabit has been developed as a preparing project for this thesis. It implements (most of) the Bitmessage protocol in Java and can therefore easily be used both in server applications as well as Android apps.

Almost all alternative clients are based on PyBitmessage, which is somewhat inconvenient as it always requires PyBitmessage to run along with the client. This isn't really a problem on desktop computers, but might render creating an easily deployable server application slightly more difficult, and is quite impractical on mobile devices.

1.8.1 Architecture

Jabit follows the Ports and Adapters architecture. There is a domain module which contains all the data types and most parts of the protocol implementation, and provides several ports, to which adapters can be attached. Adapters include the data repositories, POW, network code and, more recently, cryptography (see 3.1.1 Bouncy Castle vs. Spongy Castle).



While there was hope to reuse the JDBC implementation of the *repositories*, it was clear that at some point someone might want to use a NoSQL database or some other means of storage. Unfortunately, an Android specific implementation came first, see 3.1.2 JDBC.

The *proof of work* adapter exists in a single- and a multithreading version. The first mainly exists to show the concept, while the second one currently is always used to do the calculation, albeit remotely on a server or wrapped in some Android compatibility code. A future immplementation that uses the GPU, which sometimes have thousands of cores, might be magnitudes faster and could easily be implemented to replace the current worker.

The *network* code was put in an adapter as some Android specific tweaks were expected. This turned out to be wrong, but it didn't slow down development and might facilitate a future rewrite of the network code.

The *cryptography* adapter was another fine example why you should never rely on static helper classes. Rewriting the code to use an adapter was a pain and it wouldn't have been complicated to do it right the first time.

2 The Bitmessage Protocol

2.1 Bitmessage Commands

Commands are messages between nodes, used to initialize a connection and exchange information about the network and available objects.

The **magic** is a number to mark the beginning of a command. It is set to `0xE9BEB4D9`, but *could* be changed for test clients so they don't inadvertently connect to the real network.

A **command** defines how the payload looks like and what the node is supposed to do with it. The commands used by the Bitmessage protocol are *version*, *verack*, *addr*, *inv*, *getdata* and *object*. Their uses are specified in the Bitmessage Protocol Specification.[7]

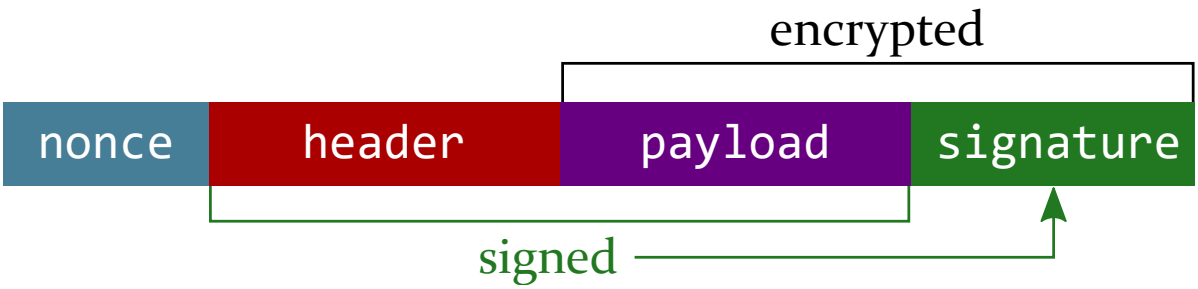
Size defines how many bytes of payload need to be read.

The **checksum** is used to prevent transmission errors.

Payload is the data that belongs to a command. It could be anything from empty to an object containing a message.

2.2 Bitmessage Objects

Objects are distributed throughout the network using the "object" command. With exception of some legacy objects, they are signed and encrypted.



To prevent malicious flooding of the network and, to a lesser extent, spam, a **nonce** needs to be found such that a specific hash over the whole object represents a number lower than a calculated threshold. This is called POW.

The object **header** consists of the expiration time, object type and version, stream number and the actual payload. (Streams ar an optimisation feature of the protocol, so it stays somewhat scalable if many people start using Bitmessage.)

The **payload** contains the actual data, depending on the message type.

The **signature** covers everything except the nonce and is encrypted along with the payload, thus covering the unencrypted payload.

There are four different types of objects in the Bitmessage protocol: *getpubkey* is used to request a public key to some address, *pubkey* contains said public key, *msg* is a typical person-to-person message, and *broadcast* is a message that is broadcast to anyone who subscribed to the sending address — which needs to be known in order to subscribe.

2.3 Encryption

Bitmessage uses elliptic curve cryptography for both signing and encryption. The math behind it is rather complicated, yet bases on the established principle to use a mathematical operation that has an inverse that is magnitudes more complicated than the operation itself. Instead of the typically used huge prime numbers, a point on an elliptic curve is multiplied by a high number.

Advantages of elliptic curves is the fact that we don't need to search for big primes, but also that the keys can be much smaller while having the same encryption strength.

The user, let's call her Alice, needs a key pair, consisting of a private key

$$k$$

which represents a huge random number, and a public key

$$K = Gk$$

which represents a point on the agreed on curve – by default that's *secp256k1* for Bitmessage. Please note that this is not a simple multiplication, but the multiplication of a point along an elliptic curve. G is the starting point for all operations on a specific curve.

Another user, Bob, knows the public key. To encrypt a message, Bob creates a temporary key pair

$$r$$

and

$$R = Gr$$

He then calculates

$$Kr$$

uses the resulting Point to encrypt the message by calculating a double SHA-512 hash over the x-coordinate and sends K along with the message.

When Alice receives the message, she uses the fact that

$$Kr = Gkr = Grk = Rk$$

so she just uses Rk to decrypt the message.

The exact method used in Bitmessage is called Elliptic Curve Integrated Encryption Scheme or ECIES, which is described in detail on Wikipedia. [3] [4]

2.4 proof of work

TODO

3 Naive Implementation

The naive implementation attempts to use the Jabit Bitmessage library as it is, with as little mobile optimisations as possible. The plan was to either discard or improve it afterwards. Fortunately improvement was possible.

3.1 Unexpected Problems

Most problems can be summarised as this: Android builds on the Java language, but not on the Java platform. While the programming language is the same and most libraries can be used without any restrictions, there are some subtle differences that make a programmer's life hard — some due to the limited resources of a mobile handset, some due to design decicions made by the Android development team.

3.1.1 Bouncy Castle vs. Spongy Castle

Jabit heavily relies on Bouncy Castle, a very popular open source encryption library.[1] Unfortunately, Android ships with a broken version of Bouncy Castle. Even worse, when building an Android app, the toolchain just discards any Bouncy Castle dependencies in favour of the built-in, broken version.

Some people recognised this problem, and built a fork of Bouncy Castle, called Spongy Castle. It basically just replaces "Bouncy" with "Spongy" wherever necessary, so it doesn't get discarded during the build process. This works fine and is quite easily done. Unfortunately, this doesn't work on the Desktop.

The Oracle JVM requires Security Providers to be signed, which is done for Bouncy Castle builds, but not so for the Spongy derivation. As forking Jabit wasn't an option, the whole cryptography part had to be refactored into an exchangeable module, and implemented twice, in both a bouncy and a spongy manifestation.

3.1.2 JDBC

Android has its own API to access the included SQLite database. While it's a nice, easy to use application programming interface (API), the Android team didn't deem it necessary to support Java database connectivity (JDBC), which is the Java standard API to connect to databases.

There is an open source project attempting to implement a JDBC driver for Android's SQLite database called SQLDroid,[1] which looked very promising. Unfortunately, it lacks essential features, such as returning the automatically generated key of an inserted row. Even worse, it doesn't have the courtesy of throwing a NotImplementedException for missing features, instead it just does nothing and returns null where a result is required, making debugging unnecessarily tedious.

Unfortunately, discovering SQLDroid was unfit for the job took more time than reimplementing all repositories using the Android database API. As they were already implemented as adapters, no change was necessary on the Jabit library. Many changes were made in the futile attempt to use JDBC though, most of them only to be reverted later on.

3.2 Expected Problems

Those problems were to be expected and need to be fixed as part of this thesis.

3.2.1 proof of work

Although modern smartphones tend to have faster processors than cheap personal computers, POW for sending a public key takes around 15 minutes on a device with four cores at 2.5 GHz. Even

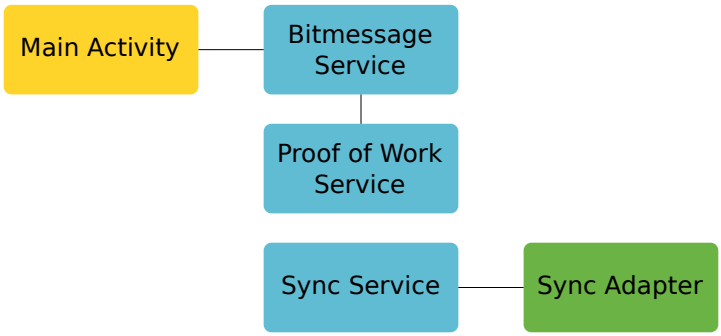
[1] https://github.com/SQLDroid/SQLDroid

worse, during this time it uses so much power that the device discharges even when connected to a power supply.

4 Android Specific Challenges

4.1 Application Lifecycle

As with any good mobile system, Android doesn't hesitate to end all threads of an application running in background, in order to save resources. Consequently anything that should be kept alive – network connections and POW come to mind – needs special treatment. This is done through foreground services. The networking part and POW hence needed to be moved into separate services. While not difficult, it needed some research to find out how it should best be done in our case.



5 Optimisations

5.1 Sync Adapters

Android provides an API that can vastly reduce battery usage for apps that regularly update data over the internet. The system triggers synchronisation, so it can optimise its sleep modes. Imagine otherwise, 15 apps synchronising every 15 minutes, one after the other. In the worst case scenario the device would need to wake up every minute, synchronize and could barely go back to sleep before the next app wants to synchronize.

Although it's not intended, synchronisation can be done within the Bitmessage protocol without any modifications. What is needed is a trusted node that's always available and therefore best run on a proper server. On successful connection all new messages are being exchanged, so we just need to connect, wait until new messages were exchanged, and then disconnect.

As Jabit is a Java library, it was trivial to create a server application using Spring Boot and Jabit. It turned out though that it's necessary to limit the number of connections in a Bitmessage server,

therefore connections are now being severed after 12 hours or when a limit of 100 connections is reached.

5.2 Server Side proof of work

To reduce power consumption and possibly reduce the time to send a message, POW can be calculated by the synchronisation server. This feature needs some changes on both server and client, and some custom extensions to the protocol.

5.2.1 Initial Idea

The server accepts messages without POW, which it will compute and fill in, and then relay that message. Of course this would leave the server extremely vulnerable to denial of service (DOS) attacks – an attacker could just send a bunch of messages and the server would be busy the rest of its life.

Fortunately, Bitmessage has very secure authentication built in: every message is signed by a private key, and can be securely verified by anyone who knows the sender's address. We'll just send a message to the server that contains the actual message as content. All nonce fields are set to zero. Ther server checks if the sender is on a white list and then calculates the nonce and relays the complete message.

A broadcast message would be best qualified for this task, as the client wouldn't have to be configured with a recipient address. The server is configured with a list of addresses that are allowed to request POW and the client just needs to know the server's IP address or host name.

5.2.2 Acknowledging Issues

Unfortunately this approach has some issues. For one, sending a message would be very different using this method, requiring some major changes on the client side. Moreover, it would be impossible to generate acknowledgements.

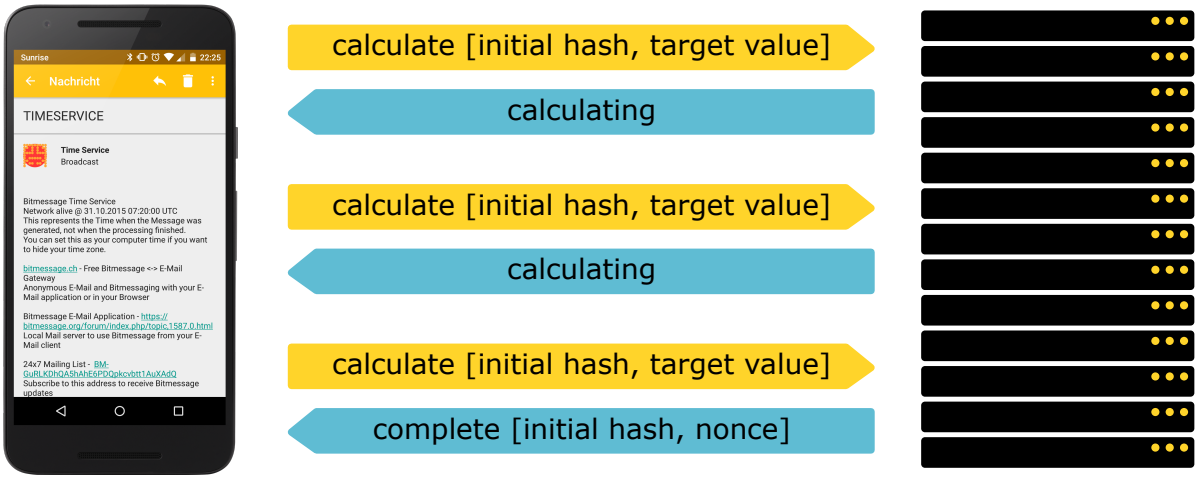
Acknowledgements are sent from the receiving client to notify the writer that the message was received. Think of it as a stamped addressed envelope delivered with a letter. The acknowledgement is part of the encrypted message the server can't read, so it couldn't calculate its POW.

Jabit doesn't support acknowledgements yet, but it would be a pity to prevent it by design.

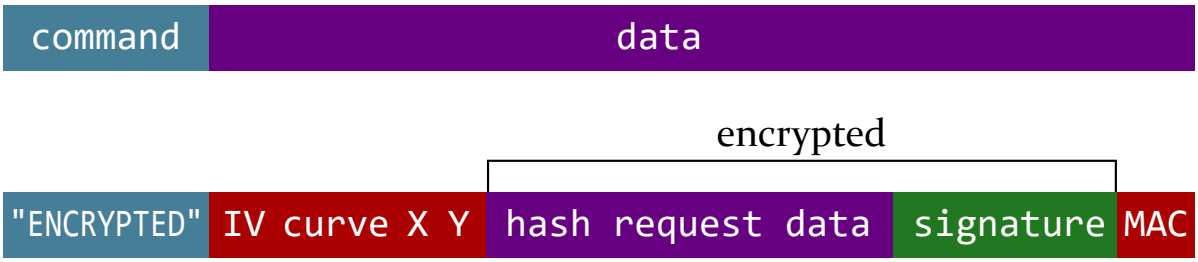
5.2.3 POW Protocol

Much of the planned design can be reused – the encryption used by broadcasts, signing the messages – but we need a wholly separate protocol, with some extra roundtrips to the server. Now the server just receives the parts needed for POW calculation: a hash and a target value.

When the nonce is found, it's saved on the database. The client will regularly poll for it (e.g. during synchronization cycles) until it retrieves the nonce.



For this feature, a new command was introduced, which needs to be implemented on both server and client. It has the same structure as any Bitmessage command, and *custom* in the command field. As payload it contains a signed and encrypted block, containing the following structure:



The custom commands for the proof of work feature are encrypted and signed, including the kind of request or response. The *command* field therefore always says `ENCRYPTED`. Afterwards comes the *encryption header*, including the initialisation vector (IV), the curve type (should normally be 0x2CA), and the X and Y coordinates of a random public key (see 2.3 Encryption, variable R).

The payload always consists of the initial hash (the base for POW calculation) and the kind of request or response. The data field contains the target difficulty for the `CALCULATE` request, and the calculated nonce for the `COMPLETE` response. For the `CALCULATING` response the data contains an emtpy array.

6 Comparison to Bitseal

7 Social and Ethical Consequences

References

[1] The legion of the bouncy castle. https://www.bouncycastle.org.

[2] National security letter. https://yale.app.box.com/NSL-Attachment-Unredacted.

[3] Wikipedia: Elliptic curve cryptography, 2015. https://en.wikipedia.org/wiki/Elliptic_curve_cryptography.

[4] Wikipedia: Integrated encryption scheme, 2015. https://en.wikipedia.org/wiki/Integrated_Encryption_Scheme.

[5] James Ball. Nsa stores metadata of millions of web users for up to a year, secret files show, 2013. http://www.theguardian.com/world/2013/sep/30/nsa-americans-metadata-year-documents.

[6] Jonathan 'Atheros' Warren and 'AyrA'. Bitmessage wiki: Stream, 2015. https://bitmessage.org/wiki/Stream.

[7] Jonathan 'Atheros' Warren and Jonathan Coe. Bitmessage wiki: Protocol specification, 2015. https://bitmessage.org/wiki/Protocol_specification.

[8] Kim Zetter. The nsa is targeting users of privacy services, leaked code shows, 2014. http://www.wired.com/2014/07/nsa-targets-users-of-privacy-services/.