Berner
Fachhochschule

Informatikseminar

Bitmessage – Kommunikation ohne Metadaten

Autor Christian Basler
Betreuer Kai Brünnler

Datum 25. Mai 2015

Inhaltsverzeichnis

1 Einführung 3
1.1 Was sind Metadaten? 3
1.2 Wie können wir Metadaten verstecken? 3

2 PyBitmessage 4

3 Protokoll 5
3.1 Nomenklatur 5
3.1.1 message, msg 5
3.1.2 payload 5
3.1.3 object 5
3.2 Ablauf 6
3.3 Meldungen 6
3.3.1 version / verack 6
3.3.2 addr 6
3.3.3 inv 7
3.3.4 getdata 7
3.3.5 object 7
3.3.6 ping / pong / getbiginv 7
3.4 Adressen 7
3.5 Verschlüsselung 8
3.5.1 Signatur 9

4 Probleme 10
4.1 Skalierbarkeit 10
4.1.1 Proof of Work 10
4.1.2 Beschränkung der Meldungsgrösse 10
4.1.3 Streams 11
4.1.4 Präfix-Filterung 11
4.2 Forward Secrecy 12
4.3 Mobile Client 12

5 Diskussion 14

1 Einführung

1.1 Was sind Metadaten?

Verschlüsselungstechnologie wie PGP oder S/MIME ermöglicht es zwar auf sichere Art und Weise Nachrichten vor neugierigen Augen zu schützen, doch seit Edward Snowden den NSA-Skandal aufgedeckt hat wissen wir dass Metadaten — vor allem Informationen darüber wer mit wem kommuniziert – genauso interessant und viel einfacher zu analysieren sind.

Es gibt einige Beispiele wie Sie durch Metadaten in Schwierigkeiten kommen können. Wenn Sie jemandem schreiben der in der IS ist, kann es durchaus sein dass Sie das nächste mal nicht in die USA fliegen können. Die No-Fly-Liste kümmert sich nicht darum dass Sie Journalist sind, oder keine Ahnung hatten dass diese Person ein Terrorist war.

Wenn Samsung erfährt, dass Apple ergiebig mit dem einzigen Produzent eines raffinierten kleinen Sensors ist, brauchen sie keine Details — das S7 wird ebenfalls einen solchen enthalten. (Dass Apple ihn braucht um ein Auto zu bauen haben sie dabei übersehen.)

1.2 Wie können wir Metadaten verstecken?

Mit E-Mail können wir die Verbindung zu unserem Mail-Provider verschlüsseln, und dieser wiederum die Verbindung mit dem Provider unseres Gesprächspartners. Dabei können wir nur hoffen dass unser Anbieter und derjenige des Empfängers sowohl vertrauenswürdig als auch kompetent sind.[1]

Bei Bitmessage senden wir eine Nachricht and eine grosse Anzahl Teilnehmer, darunter den eigentlichen Empfänger. Die Nachricht ist dabei so verschlüsselt, dass nur die Person welche den privaten Schlüssel besitzt diese entschlüsseln kann. Alle Teilnehmer versuchen dies um die für sie bestimmten Nachrichten zu finden.

[1] Gratis sollte er natürlich auch noch sein.

2 PyBitmessage

PyBitmessage ist der Standard Bitmessage Client und, wie der Name andeutet, in Python implementiert. Der Client bietet zwar alle Funktionen von Bitmessage, die User Experience ist allerdings nicht ganz optimal. Insbesondere lassen sich Nachrichten nicht in Ordner verschieben oder mit Labels versehen.

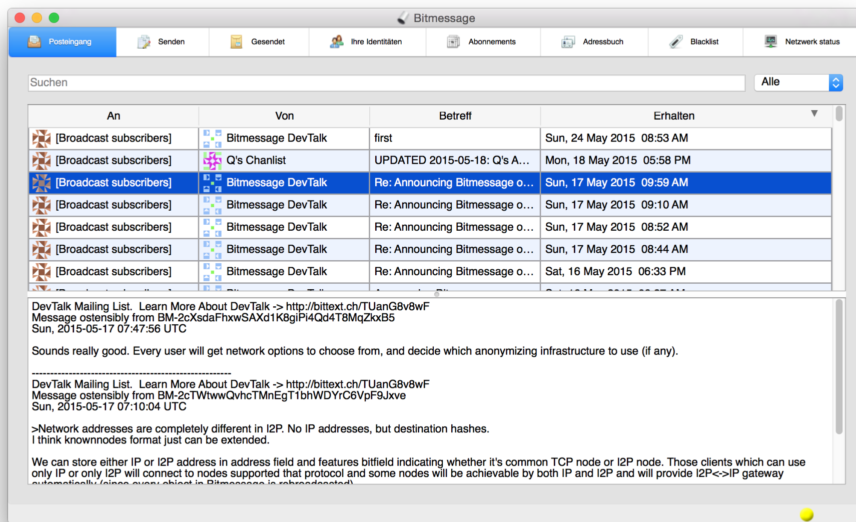

Abbildung 1: Der Posteingang von PyBitmessage.

Der Client ist ausserdem ziemlich langsam im Berechnen des Proof of Work. Auf einem MacBook Pro von 2012 kann es durchaus einige Minuten dauern bis eine Nachricht gesendet werden kann.

Man braucht natürlich nur schon die Idee des Protokolls, um auf die Idee zu kommen Meldungen an mehrere Empfänger gleichzeitig zu senden. Dies geht so, dass die Adresse selbst in einen privaten Schlüssel umgewandelt wird und so zum *entschlüsseln* der Broadcasts benutzt werden kann. Entsprechend kann man beim Erfassen einer Nachricht wählen ob sie an einen bestimmten Empfänger gehen soll, oder an alle, die unsere Broadcasts abonniert haben. Jeder, der die Adresse kennt kann dabei allerdings ihre Broadcasts lesen.

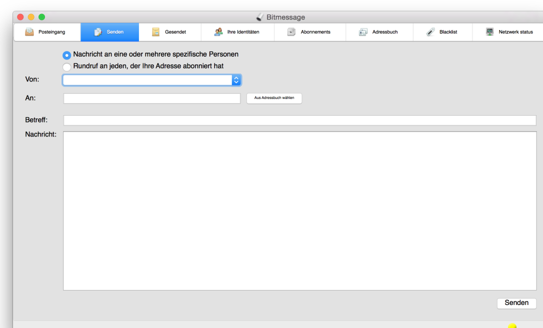

Abbildung 2: Eine neue Nachricht erfassen.

3 Protokoll

Wir benutzen die folgende Konvention um zwischen verschiedenen Bestandteilen der Protokolls zu unterscheiden:

version	für Meldungen zwischen Netzwerkknoten
pubkey	für Objekte welche im Netzwerk verteilt werden
A	für einzelne Netzwerkknoten

3.1 Nomenklatur

Es gibt einige Begriffe welche schnell verwechselt werden können. Es folgt eine Liste der verwirrendsten.

3.1.1 message, msg

Eine Nachricht oder *message* wird von einem Netzwerkknoten zum anderen geschickt, z.B. um neue Objekte anzukündigen oder um die Verbindung aufzubauen.

Ein **msg**-Objekt andererseits enthält die verschlüsselte Nachricht von einem Benutzer an einen anderen.

3.1.2 payload

Payload werden die Nutzdaten eines Protokolls genannt. Es gibt drei Arten von Payload:

1. Der Payload von Meldungen, z.B. *Inventory Vectors*.
2. Der Payload von Objekten. Dieser wird im Netzwerk verteilt.[2]
3. Verschlüsselter Payload, der Chiffretext mit einigen Zusatzinformationen welche für die Entschlüsselung benötigt werden.[3]

3.1.3 object

Ein Objekt ist einer Art von Meldung, deren Payload zwischen allen Netzwerkknoten verteilt wird. Manchmal wird auch nur der Payload gemeint. Um ein Objekt zu senden, wird ein *Proof of Work* benötigt.

[2] Und ist Teil vom Meldungs-Payload.
[3] Dieser wiederum ist Teil vom Objekt-Payload.

3.2 Ablauf

Der neu gestartete Netzwerkknoten **A** stellt die Verbindung zu einem zufälligen Knoten **B** ais seinem Knotenverzeichnis her und sendet eine *version*-Meldung, die die aktuellste unterstützte Protokollversion ankündigt. Falls **B** die Version akzeptiert,[4] antwerted er mit einer *verack*-Meldung, gefolgt von der eigenen *version* mit ihrer neusten unterstützten Protokollversion. Knoten **A** entscheidet nun ob er die Version von **B** akzeptiert und sendet in diesem Fall seine *verack*-Meldung.

Wenn beide Knoten die Verbindung akzeptieren, senden sie je eine *addr*-Meldung mit bis zu 1000 bekannten Knoten, gefolgt von einer oder mehreren *inv*-Meldungen, welche alle bekannten gültigen Objekte mitteilt. Danach wird eine *getobject*-Meldung für jedes noch fehlende Objekt gesendet.

Auf *getobject* antwortet der Knoten mit einer *object*-Meldung, welche dann das angeforderte Objekt enthält.

Ein Knoten verbindet sich aktiv mit acht anderen knoten und erlaubt beliebig viele eingehende Verbindungen. Wenn ein Benutzer an Knoten **A** ein neues Objekt erzeugt, wird es mittels *inv*-Meldung bei acht der angebundenen Knoten angeboten. Diese fordern es an und bieten es wiederum bei acht Nachbarknoten an, bis es an alle Knoten verteilt ist.

3.3 Meldungen

Die Meldungen, Objekte und das Binärformat sind im Bitmessage-Wiki gut dokumentiert[3], die Beschreibungen hier sind deshalb darauf konzentriert um was es geht und wie man sie benutzt.

3.3.1 version / verack

Die *version*-Meldung enthält die aktuellste vom Knoten unterstützte Protokollversion, die Streams für die er sich interessiert und die unterstützten Features. Falls der andere Knoten akzeptiert, bestätigt er mitels *verack*. Die Verbindung gilt als initialisiert wenn beide Knoten eine *verack*-Meldung gesendet haben.

3.3.2 addr

Enthält bis zu 1000 bekannte Knoten mit deren IP-Adresse, Port, Stream und unterstützten Features.

[4] Eine Version wird normalerweise akzeptiert, wenn sie höher oder gleich der eigenen höchsten unterstützten Version ist. Knoten welche eine experimentelle Protokollversion implementieren können auch ältere Versionen akzeptieren.

3.3.3 inv

Eine *inv*-Meldung enthält die Hashes von bis zu 50000 gültigen Objekten. Falls das Inventar mehr Objekte enthält können mehrere Meldungen gesendet werden.

3.3.4 getdata

Kann bis zu 50000 Objekte anfordern, indem es deren Hashes sendet.

3.3.5 object

Enthält ein angefordertes Objekt, das eines von folgenden sein kann:

getpubkey	Eine Aufforderung an eine Adresse, deren öffentlichen Schlüssel zu senden. Dieser wird benötigt um die Meldung an diese Adressen zu verschlüsseln.
pubkey	Ein öffentlicher Schlüssel. Siehe 3.4 Adressen
msg	Eine Nachricht an einen bestimmten Benutzer.
broadcast	Eine Nachricht welche so verschlüsselt wird, dass jeder, der die Adresse kennt, sie entschlüsseln kann.

3.3.6 ping / pong / getbiginv

Wer den Source Code von PyBitmessage untersucht, ist vielleicht über einige Meldungen irritiert welche implementiert zu sein scheinen, aber nirgends in der offiziellen Spezifikation zu finden sind. *Ping* bringt einen Knoten (sofern implementiert) dazu ein *pong* zurückzusenden. Verwendet wird das Feature jedoch nirgends. *Getbiginv* scheint dafür gedacht zu sein das ganze Inventar abzufragen, aber soweit ich es verstehe wird es nirgends verwendet.[4]

3.4 Adressen

BM-2cXxfcSetKnbHJX2Y85rSkaVpsdNUZ5q9h: Adressen beginnen mit "BM-ünd sind, genau wie Bitcoin-Adressen, Base58 codiert.[5]

version	Adressversion.
stream	Stream-Nummer.

[5]Dieses verwendet die Zeichen 1-9, A-Z und a-z ohne die leicht verwechselbaren Zeichen I, l, 0 and O.

ripe	Hash der aneinandergefügten öffentlichen Schlüssel zum signieren und verschlüsseln. Wichtig: in **pubkey**-Objekten werden die Schlüssel ohne führendes 0x04 gesendet, doch um den Ripe zu berechnen muss dieses Byte vorangestellt werden. Da dies für fast alle Verwendungszwecke der Schlüssel nötig ist, lohnt es sich dies gleich beim erstellen des Objekts zu machen. `ripemd160(sha512(pubSigKey + pubEncKey))`
checksum	Die ersten vier Bytes eines doppelten SHA-512-Hashs der vorangehenden Daten. `sha512(sha512(version + stream + ripe))`

3.5 Verschlüsselung

Bitmessage benutzt Elliptische-Kurven-Kryptographie zum signieren als auch zum verschlüsseln. Während die Matematik hinter elliptischen Korven sogar noch komplizierter zu verstehen ist all der ältere Ansatz, riesige Primzalen zu multiplizieren, so basiert es doch auf dem gleichen Prinzip, eine Mathematische Operation durchzuführen welche in eine Richtung sehr schnell, jedoch sehr schwierig umzukehren ist. An Stelle von zwei grossen Primzahlen werden hier ein Punkt auf der elliptischen Kurve mit einer sehr grossen Zahl multipliziert.[6]

Die Benutzerin, nennen wir sie Alice, benötigt ein Schlüsselpaar, welches aus dem privaten Schlüssel

$$k$$

besteht, der eine riesige Zufallszahl darstellt, und einem öffentlichen Schlüssel

$$K = Gk$$

der einen Punkt auf der vorher definierten Kurve repräsentiert.[7] Beachten Sie bitte dass dies keine einfache Multiplikation ist, sondern die skalare Multiplikation eines Punktes auf der elliptischen Kurve. G ist der Startpunkt für alle Operationen auf einer spezifischen Kurve.

Ein anderer Benutzer, Bob, kennt den öffentlichen Schlüssel. Um eine Nachricht zu verschlüsselt erstellt er das temporäre Schlüsselpaar

$$r$$

und

$$R = Gr$$

Danach berechnet er

$$Kr$$

[6]Bitte fragen Sie mich nicht wie das genau geht. Falls Sie es wirklich wissen möchten, beginnen Sie auf http://de.wikipedia.org/wiki/Elliptische_Kurve und http://de.wikipedia.org/wiki/Elliptic_Curve_Cryptography. Falls Sie etwas machen möchten das funktioniert, verwenden Sie lieber eine Bibliothek wie Bouncy Casle welche die harte Arbeit übernimmt.

[7]Bitmessage benutzt eine Kurve namens *secp256k1*.

benutzt den daraus folgenden Punkt um die Meldung zu verschlüsseln[8] und sendet K zusammen mit der verschlüsselten Nachricht.

Wenn Alice die Meldung empfängt, benutzt Sie die Tatsache dass

$$Kr = Gkr = Grk = Rk$$

alse berechnet sie einfach Rk um die Meldung zu entschlüsseln.

Die genaue von Bitmessage verwendete Methode wird Elliptic Curve Integrated Encryption Scheme oder ECIES genannt, welche auf Wikipedia detailliert beschrieben wird (http://de.wikipedia.org/wiki/Elliptic_Curve_Integrated_Encryption_Scheme).

3.5.1 Signatur

Um Objekte zu signieren verwendet Bitmessage Elliptic Curve Digital Signature Algorithm oder ECDSA. Dies ist etwas komplizierter als ECIES. Wenn Sie Details wissen möchten ist Wikipedia einmal mehr eine gute Anlaufstelle: http://de.wikipedia.org/wiki/Elliptic_Curve_DSA.

Ein interessantes Detail für potentielle Entwickler von Bitmessage-Clients — vor allem wenn sie es mit einem objektorientierten Ansatz machen möchten: die Signatur geht über alles aus dem Objekt-Header ohne das Nonce und alles aus dem Objekt-Payload ohne die Signatur selbst. Natürlich sind nicht alle Objekte signiert.[9]

[8] Genaugenommen wird ein doppelter SHA-512-Hash über der X-Koordinate benutzt um den symmetrischen Schlüssel zu erzeugen.

[9] Mein Ansatz: zuerst denken, dann falsch implementieren, dann viel umschreiben.

4 Probleme

4.1 Skalierbarkeit

Bitmessage skaliert nicht.[10] Gitb es sehr wenige Benutzer, so gibt es auch keine Anonymität. Mit nur einer handvoll Benutzern ist es einfach (beispielsweise für die NSA), den Verkehr zwischen Knoten zu analysieren um herauszufinden wer wem schreiben könnte. Oder man überwacht einfach mal alle.

Mit vielen Benutzern wächst der benötigte Traffic und Speicherplatz quadratisch. Dies geschieht, weil mit mehr Benutzern um eine Nachricht zu schreiben es auch mehr mögliche Gesprächspartner für bestehende Benutzer gibt.

4.1.1 Proof of Work

Proof of work hat zwei zwecke. Es hilft, das Netzwerk zu schützen indem es verhindert dass einzelne Knoten es mit Objekten fluten, aber auch den einzelnen Benutzer vor Spam zu bewahren. Es gibt einen minimal nötigen Proof of Work um Objekte im Netzwerk zu verteilen, doch der Benutzer kann für seine Adressen höhere Anforderungen stellen, falls er mit Angeboten für billiges Viagra™ zugeschüttet wird. Der für eine Adresse nötige Proof of Work wird im **pubkey**-Objekt mitgeteilt. Absender, welche in der Kontaktliste eines Benutzers sind sollten normalerweise kein höherer Proof of Work machen müssen.

Die Schwierigkeit wird mittels Nachrichtenlänge und Lebensdauer berechnet, das heisst eine grössere Meldung oder eine welche länger im Netzwerk gespeichert wird ist kostet mehr beim senden.

$$d = \frac{2^{64}}{n(l + \frac{tl}{2^{16}})}$$

d zielschwierigkeit
n nötige Versuche pro Byte
l Payload-Länge + Extra-Bytes (um es nicht zu einfach zu machen viele winzige Meldungen zu versenden)
t Lebensdauer

Um den Proof of Work durchzuführen, muss ein Nonce[11] gefunden werden, so dass die ersten acht Bytes vom Hash des Objekts (inklusive Nonce) eine kleinere Zahl repräsentieren als die Zielschwierigkeit.

4.1.2 Beschränkung der Meldungsgrösse

Um zu verhindern dass bösarige Benutzer einzelne Knoten blockieren, dürfen Meldungen nicht grösser als 256 KiB sein. Wegen des Proof of Work sind grössere Nachrichten für den Normalgebrauch

[10] Noch nicht.
[11] Number used once.

sowieso nicht praktikabel, aber sie könnten benutzt werden um Knoten mit Müll-Meldungen zu beschäftigen.

4.1.3 Streams

Die vorgesehene Lösung für das Skalierungsproblem ist, den Traffic – genau genommen Adressen – in Streams aufzuteilen. Ein Knoten liest nur auf den Streams, welche seine Adressen betreffen. Wenn er ein Objekt an einen anderen Stream schicken möchten, verbindet er sich einfach mit einem Knoten im gewünschten Stream, sendet sein Objekt und schliesst die Verbindung wieder. Wenn alle aktiven Streams voll sind, wird für neue Adressen ein neuer Stream verwendet.

Das ungelöste Problem ist, herauszufinden wann ein Stream voll ist. Ein weiteres Problem ist die Tatsache dass, währen das Netzwerk wächst, der Traffic auf den vollen Streams mitwächst, da es mehr Benutzer gibt welche jemandem auf dem vollen Stream schreiben möchten. Der Traffic auf dem vollen Stream wächst also linear mit der Netzwerkgrösse.

4.1.4 Präfix-Filterung

Jonathan Coe schlägt diesen interessanten Ansatz vor, den Traffic aufzuteilen. Dies würde ein Protokoll-Update erfordern, würde aber eine viel genauere Kontrolle darüber erlauben, wie viel Traffic ein Knoten verarbeiten will.[1]

Anstelle von Streams stellen wir uns eine Adresse als Blatt eines Binärbaums der Höhe 65 vor. Die Position wird über die ersten 64 Bits des Ripe einer Adresse. Eine Präfix-Lenge n definiert den Teilbaum ab welchem wir Meldungen lesen. Ein sendender Client setzt ein 64-Bit-Nonce bei welchem die ersten n Bits vom Ripe der Empfängeradresse kopiert und der Rest zufällig gesetzt wird.

Nehmen wir nun an, der Ripe von Bobs Adresse starte mit `00101001...` und hat eine Präfix-Länge von 3. Alice sendet ihre Meldung mit dem Tag `00110100...`. Die ersten drei Bits müssen gleich sein, aber der Rest ist zufällig gewählt. Bobs Client verarbeitet nun alle Meldungen welche seinem Präfix entsprechen, er muss also nur $^1/_8$ des Gesamttraffics lesen.[12]

Wie Bitmessage populärer wird, wird es auch mehr und mehr Traffic generieren. Bob möchte deshalb möglicherweise seine Präfix-Länge auf 4 erhöhen, was den zu verarbeitenden Traffic weiter auf $^1/_{16}$ des Gesamtvolumens reduziert. Um dies zu tun, publiziert er einfach seinen **pubkey** mit seiner aktualisierten Präfix-Länge. Das heisst natürlich auch dass entweder immer ein **pubkey** publiziert sein muss, oder Alice muss wenigstens einmal online sein während der **pubkey** publiziert ist. Andernfalls gibt es in unserem Szenario eine 50% Chance dass die Nachricht Bob nicht erreicht.

Dies würde es zwar einem Smartphone-Client erlauben nur seine eigenen Meldungen zu verarbeiten,[13] aber damit würde man auch seine Anonymität beinahe komplett aufgeben.

[12] Im Moment ist der Traffic insgesamt etwa 1 GiB im Monat.
[13] Ein Präfix von 64 würde höchstwahrscheinlich bedeuten dass man auf dem Stream aleine ist.

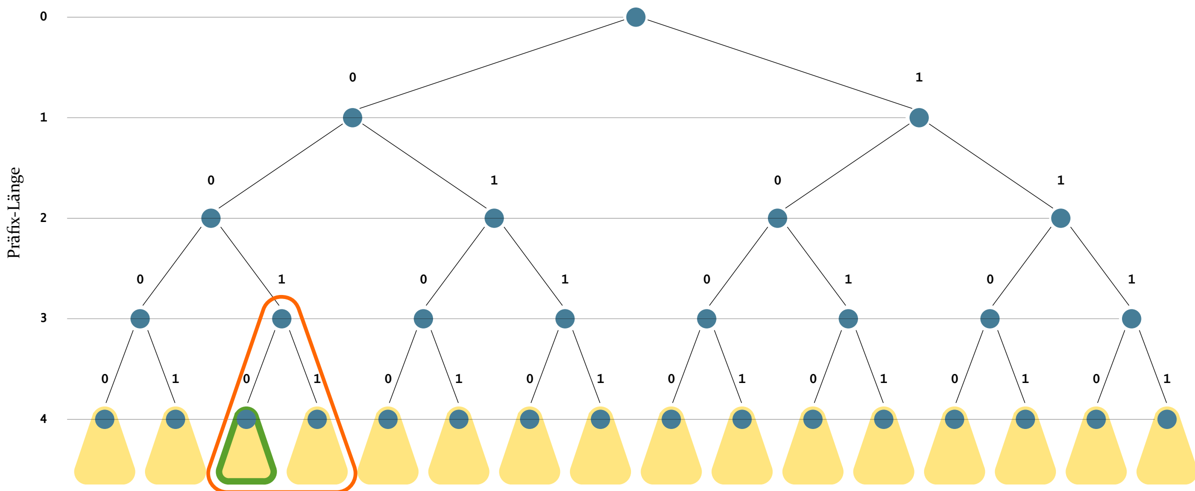

Abbildung 3: Die Pefix-Länge geht bis 64, jedes der gelben Dreiecke stellt folglich einen Teilbaum der Höhe 61 dar.

4.2 Forward Secrecy

Offensichtlich ist es für einen Angreifer trivial alle (verschlüsselten) Objekte zu sammeln welche durch das Bitmessage-Netzwerk verteilt werden — sofern Speicherplatz kein Problem ist. Sollte dieser Angreifer irgendwie an den privaten Schlüssel eines Benutzers kommen, kann er alle gespeicherten Meldungen entschlüsseln welche für diesen Benutzer bestimmt sind und sich ausserdem als diesen ausgeben.[14]

Glaubhafte Abstreitbarkeit (plausible deniability) kann, in einigen Szenarios, dagegen helfen. Bei dieser Aktion, auch ëine Adresse atomisieren"[15] genannt, wird der private Schlüssel anonym veröffentlicht.[16]

Perfect Forward Secrecy scheint nicht praktikabel implementierbar zu sein, da man dazu vor dem Senden der eigentlichen Nachricht Informationen austauschen muss. Diese brauchten wiederum Proof of Work um das Netzwerk zu schützen, was für den Sender die doppelte Arbeit bedeutet und dreimal solange dauert um sie zu senden — das heisst, falls beide Clients online sind. Der Austausch von Nachrichten würde so gut wie unmöglich wenn beide Benutzer nur sporadisch online sind.

4.3 Mobile Client

Es gibt zwar inzwischen einen Client für Android,[17] dieser benötigt jedoch sehr viel Traffic, der Proof of Work dauert ewig und die Batterie hält nicht sehr lange.

[14] Das letztere ist schwieriger wenn der Schlüssel durch eine Bruteforce-Attacke erworben wurde.
[15] "Nuking an address."
[16] Siehe https://bitmessage.ch/nuked/ für ein Beispiel.
[17] Bitseal: https://play.google.com/store/apps/details?id=com.github.bitseal

In PyBitmessage gibt es die Option, den Empfänger eindeutig identifizierbar zu machen. Dies würde es einem Server erlauben nur die relevanten Meldungen weiterzuleiten. Zwar gibt man dabei seine Anonymität auf, aber das ist immer noch besser als die Ansätze der E-Mail-Relays, wo der private Schlüssel gleich beim Server liegt.

Ein Ansatz, der Anonymität trotz unterstützendem Server erlaubt, schlug Dan Smith vor.[2] Dabei wird ein drittes Schlüsselpaar generiert, mit dem der String ÏDENTIFICATIONünd einige zufällige Bytes verschlüsselt wird. Der Relay-Server muss dabei den privaten Identifikationsschlüssel erhalten — soweit muss man ihm also noch vertrauen. Wenn er also mit Hilfe des Identifikationsschlüssels den Text ÏDENTIFICATIONëxtrahieren kann, ist die Meldung für einem bestimmt und wird weitergeleitet.

Der Proof of Work schliesslich lässt sich sehr einfach auf einem Server erledigen. Mit einer optimierten Implementation sollte er aber auf einem modernen Smartphone in vernünftiger Zeit machbar sein. Die Geräte haben inzwischen oftmals mehr als vier Prozessorkerne und einen dedizierten Grafikchip, der sich ausgezeichnet zum parallelen Berechnen von Hashes eignet.

5 Diskussion

Anonymität hat ihren Preis. Bei Bitmessage ist es Traffic, Speicherplatz und Rechenpower. Bei E-Mail ist es Vertrauen. Wenn wir unserem E-Mail-Provider nicht vertrauen können (wer kann das?), ist Bitmessage eine alternative, wenn auch nicht vollständig ausgereift.

Ich finde die Idee eines Trustless-Protokolls[18] und Peer-To-Peer Netzwerke die üblicherweise soche Protokolle verwenden äusserst interessant. Zu Beginn war das Internet ein riesiges Netzwerk gleichberechtigter Teilnehmer, aber heutzutage scheint alles zu Google oder Facebook zu führen. P2P und Trustless-Protokolle geben uns ein Stück dieser Freiheit zurück welche in der Cloud verlorengegangen ist.

Dass es nun einen P2P E-Mail-Ersatz gibt finde ich absolut grossartig.

Ja, es skaliert nicht so gut wie E-Mail. Aber dank Proof of Work denke ich, dass wir kein mit E-Mail vergleichbares Spam-Problem haben, was den Traffic schon mal gut halbiert. Falls Bitmessage E-Mail jemals ersetzt, würde ein grosser Teil des E-Mail-Verkehrs wohl eher durch Instant-Messaging ersetzt. Vor allem Firmenintern, wo eine eigene Infrastruktur aufgebaut werden kann, muss nicht unbedingt Bitmessage verwendet werden.

[18] Ein Protokoll, das kein Vertrauen benötigt.

Literatur

[1] Jonathan Coe. Bitmessage wiki: Scalability through prefix filtering, 2015.
https://bitmessage.org/wiki/Scalability_through_Prefix_Filtering.

[2] Dan Smith. A tweak to enable trust-less bm relay servers (and lightweight mobile clients), 2014.
https://bitmessage.org/forum/index.php?topic=3725.msg7871.

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

[4] Jonathan 'Atheros' Warren and ISibbol. Biginv and ping/pong, 2015.
https://github.com/Bitmessage/PyBitmessage/issues/112.

Abbildungsverzeichnis

1 PyBitmessage: Inbox 4
2 PyBitmessage: Senden 4
3 Präfix-Filter: Binärbaum 12