VRIJE
UNIVERSITEIT
BRUSSEL

Graduation thesis submitted in partial fulfilment of the requirements for
the degree of de Ingenieurswetenschappen: Computerwetenschappen

GLYCOS

an extensible, resilient and private peer-to-peer online social network

Ruben De Smet

June 2018

Promotors: prof. dr. Ann Dooms prof. dr. Jo Pierson

sciences and bioengineering sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad van
Master of Science in de Ingenieurswetenschappen:
Computerwetenschappen

GLYCOS

een uitbreidbaar, zelfherstellend, besloten peer-to-peer online sociaal netwerk

Ruben De Smet

Juni 2018

Promotors: prof. dr. Ann Dooms prof. dr. Jo Pierson

wetenschappen en bio-ingenieurswetenschappen

Of course Facebook cares about your privacy. They sell it.

---

anonymous IRC user

Abstract

Online privacy typically comes in two forms. At one hand, users can typically choose with whom of their connections to share information, and have plenty of *social privacy controls*. The so-called *privacy problem* is more about *institutional privacy*, whereby the service provider fails to securely store users' data, be it on purpose or not (danah boyd & Hargittai, 2010). When on purpose, these data are often *mined* for profit through resale of profiles; often called profiling.

One way of giving back this institutional privacy to citizens, is by taking away the institution as a whole, by decentralising the application. Care has to be taken as not to make potential "re-centralisation" possible, as is happening to email, where a few large email server providers take up large portions of worldwide email traffic. Mailchimp, a large email marketing company, reports GMail having over 1.8 billion more email delivered than Hotmail (Khan, 2015), and online resources seem to suggest that both Microsoft and GMail are by far the world most popular email service providers (Lewkowicz, 2017; Datanyze, 2018).

By opting for a carefully designed peer-to-peer design, the risk of this "re-centralisation" can be minimised.

Several noteworthy efforts have been made to "re-*de*centralise" online social media platforms, not only academic, also commercial and community projects. These efforts often adapt underlying protocols on a per-feature basis, slowing down the development process, and often scaring off non-domain-specialist developers. This is in contrast with how web development and mobile app development works, where developers have application programming interfaces (APIs) such as cookies, structured query language (SQL) (often with object relational mapping (ORM)), and Representational State Transfer (REST). These APIs offer developers an *abstract method* of reasoning about their application.

We explore a peer-to-peer, fully trustless, obfuscated graph-database model, that is only readable and efficiently traversable by legitimate users. Outsiders only learn a minimal amount of metadata, without revealing content nor structure of the graph database.

This database model is designed as building block for development of online social media, keeping in mind mobile-friendliness, scalability, and efficiency.

Acknowledgements

I would like to thank my promotor prof. dr. Ann Dooms for being open to me suggesting this thesis, her willingness and unfailing energy to supervise, and for providing valuable feedback to the cryptographic and technical aspects of this thesis, and the upcoming paper.

Also as promotor, prof. dr. Jo Pierson too provided me with valuable feedback on the conference paper we are writing for the upcoming *IFIP Summer School on Privacy and Identity Management* in 2018[1]; parts of this thesis will appear on this conference. As a soon-to-be engineer, his feedback on the social part of this thesis was a necessity, since my academic background on this topic was non-existent.

Both prof. Ann Dooms and prof. Jo Pierson are busy people, so I would like to thank both again for freeing up time for meeting with the three of us.

Still in the academic world, dr. Seda Gürses from the KU Leuven provided us with valuable insight in the large world of privacy enhancing technologiess (PETs), and the use and pitfalls of technology in them. I am looking forward to the possibility of working with her and her colleagues in the near future!

On the side of supporters, I would like to thank my parents for enabling me to attend university, for being there when needed, and for having to hear me rattle about what I was doing. I am equally grateful to my girlfriend Ellen, willing to go through a stressful last year with me, for the both of us. What doesn't kill us makes us stronger.

To relieve some of this stress, my friends Thomas, Pieter-Jan, Tom, and Eline were always there for a talk or some pass-time. I would like to thank Tom a second time, for his feedback also from a technical and implementation point of view. His insight in the practical part was and will be valued.

My ever gratitude also goes to Marc, with whom Tom and I had endless insightful discussions. I am sorry he did not live to see me graduate.

Thank you, all of you, for supporting this fight for privacy
Ruben De Smet – 29th June 2018

[1] https://www.ifip-summerschool.org/

Contents

Abstract iii

Acknowledgements v

I Background 1

1 Privacy 3
1.1 Privacy online . . . 4
1.1.1 Privacy as control . . . 4
1.2 Enhancing online privacy . . . 4
1.2.1 Decentralisation . . . 5
1.2.2 Privacy as confidentiality . . . 5
1.3 Privacy as practice . . . 5

2 Decentralising social networks 7
2.1 Federated social networks . . . 7
2.2 Peer-to-peer communication . . . 8
2.2.1 Ad-hoc . . . 8
2.2.2 Friend-to-friend . . . 8
2.2.3 Peer-to-peer . . . 8
2.3 Data storage . . . 9
2.3.1 Friend-to-friend . . . 9
2.3.2 A note on block chains . . . 9
2.3.3 DHT: Kademlia . . . 9
2.3.4 Hybrids and variants . . . 10
2.4 Critique on decentralisation . . . 10

3 Existing online social networks 13
3.1 Classical social networks . . . 13
3.1.1 Mobile apps . . . 13
3.2 Solving the privacy problem . . . 13
3.2.1 Federated systems . . . 15
3.2.2 Supported by cryptographic currencies . . . 15

4 Cryptography **19**
4.1 Elliptic curve cryptography . . . 19
4.1.1 Elliptic curve operations . . . 19
4.1.2 Efficient elliptic curves . . . 21
4.1.3 Discrete logarithm problem . . . 21
4.1.4 Cryptography on elliptic curves: key exchange . . . 21
4.2 Monero: decentralised fungible digital currency . . . 21
4.2.1 System parameters . . . 22
4.2.2 Unlinkability . . . 22
4.2.3 Untraceability . . . 23

5 Resource Description Framework **25**
5.1 Querying the data model . . . 26

II System description **27**

6 High-level system overview **29**
6.1 Peer-to-peer . . . 29
6.2 Generic data storage . . . 29

7 Cryptographic primitives **31**
7.1 Hash functions . . . 31
7.2 Ephemeral public keys . . . 31
7.3 Schnorr signature . . . 32
7.4 The ring signature . . . 34

8 Peer-to-peer access controlled RDF triple-store **35**
8.1 Rationale . . . 35
8.2 Data model . . . 36
8.2.1 Deterring statistical analysis . . . 37
8.3 Operations on the graph . . . 38
8.3.1 Create . . . 38
8.3.2 Read . . . 39
8.3.3 Update . . . 39
8.3.4 Delete . . . 40
8.4 Optimizing the data model . . . 41

9 Implementation **43**
9.1 Cryptography . . . 43
9.1.1 Disinheriting Monero . . . 44
9.1.2 Signatures . . . 44
9.1.3 Off-the-shelve cryptography . . . 46
9.2 Non blocking computations . . . 46
9.2.1 Network stack . . . 47
9.2.2 Notes on asynchronous programming . . . 47
9.3 Graph APIs . . . 48
9.4 Testing . . . 48
9.5 Binaries . . . 49

Epilogue **51**

Future work **53**

Glossary **55**

Part I

Background

Chapter 1

Privacy

Defining what is meant by "privacy" has known several attempts. Zureik, Stalker, and Smith, 2010 take on a multidimensional approach to this definition, wherein privacy is captured as (1) the right to be let alone; (2) limited intrusion to the self; (3) secrecy, confidentiality; (4) control of personal information (w.r.t. third parties); (5) personhood; and (6) intimacy (Hoepman & van Lieshout, 2012).

The same paper by Hoepman and van Lieshout lists a few fallacies when discussion privacy.

One of the more popular fallacies is the "nothing to hide"-argument, which often recited as "If you have nothing to hide you have nothing to fear". Edward Snowden — an ex-NSA contractor well know for disclosing top secret documents about global surveillance practices by NSA and their international partners called the "Five Eyes" — famously (Kleeman, 2015) refuted this in 2015 with

> "Arguing that you don't care about the right to privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say." — Snowden, 2015

Snowden hereby means that revoking a right, just because it is allegedly unused, is unjust. A human right does not need justification; rather the act of infringing the right requires justification.

As Hoepman and van Lieshout state, the premise of the "nothing to hide"-argument is invalid; it assumes that privacy is about hiding the wrong or illegal, while it is about not having to show, rather than to hide; to be let alone, unless urgently required (Warren & Brandeis, 1890).

Research on *privacy engineering* can take several forms. Diaz and Gürses identify three research paradigms, wherein the focus of the research is on different aspects of privacy engineering.

Privacy as control focusses on how to enable users to share, giving them *control* over with whom their data is shared;

Privacy as confidentiality where research focusses on keeping private data confidential by limiting, avoiding, and restricting access, often using cryptographic techniques;

Privacy as practice focusses on how a system can make data flow transparent to the user, by clever user interface design or other techniques. These techniques try to empower the user, increasing their ability to steer their own behaviour.

1.1 Privacy online

Many online platforms nowadays commodify data: companies gather, store and combine users' (and sometimes non users') data — often referred to as *datafication* (Cukier & Mayer-Schoenberger, 2013) — and try to turn it into profit, commodifying these data. The data gathered range from what we tweet on Twitter, to the sub-second timing of how long we look at a post or advertisement, to the websites we visit, to where we take a taxi and how long we stay at that place. From these, marketing companies like Facebook try to infer our interests in products, but also political and sexual (Ryssdal, 2014; Tinker, 2018).

It is clear that these are privacy sensitive data, having them in a single companies hands can lead to abuse. As a prime example, the recent Cambridge Analytica scandal, where *Cambridge Analytica Ltd.* scraped up to 87 million Facebook profiles to target American voters. The design of the Facebook API allowed the company to not only scrape data from users that consented to a survey for "academic use", but also all profiles in those users' social circle (Graham-Harrison & Cadwalladr, 2018).

1.1.1 Privacy as control

It is also interesting to distinguish between *institutional privacy* and *social privacy*. *Institutional privacy* refers to data being shared with one or more organisations, who process the data, and serving it back to the same (or different) users.

The second form of privacy is what Facebook refers to when they claim "we give you control of your privacy" (Facebook, n.d.). *Social privacy* refers to having *control* over with whom, on a social level, one shares information; this is one of the three privacy technology research paradigms identified by Gürses, 2010, called "privacy as control".

1.2 Enhancing online privacy

Attempts to enhance privacy on online platforms may employ different techniques. One way to increase institutional privacy is by *end-to-end encryption*, whereby data is made cryptographically unintelligible by the end-users, before being transmitted through the central organisation. A well-known example is the double-ratchet algorithm, used by instant messaging systems like Signal and WhatsApp (Marlinspike, 2013), based on the earlier Off-the-Record system (Borisov, Goldberg, & Brewer, 2004).

It is difficult however to hide the *metadata* that is generated by these means of communication. Consider the way the Electronic Frontier Foundation illustrates this issue:

> "They know you rang a phone sex service at 2:24 am and spoke for 18 minutes. But they don't know what you talked about.
> "They know you called the suicide prevention hotline from the Golden Gate Bridge. But the topic of the call remains a secret.
> "They know you spoke with an HIV testing service, then your doctor, then your health insurance company in the same hour. But they don't know what was discussed.
> "They know you received a call from the local NRA office while it was having a campaign against gun legislation, and then called your senators and congressional representatives immediately after. But the content of those calls remains safe from government intrusion.
> "They know you called a gynecologist, spoke for a half hour, and then called the

> local Planned Parenthood's number later that day. But nobody knows what you spoke about." (Opsahl, 2013)

Signal makes an extensive effort to remove metadata as soon as possible (Signal, 2018), and to avoid metadata altogether by, for example, using verifiable computation to search for your contacts (Marlinspike, 2017). Other services like WhatsApp do not make this effort[1]. While they claim to delete messages from their servers, they do not make an effort à la Signal to hide contact discovery, they track location in a complicated and extensive way, and they maintain a list of a user's connections.

1.2.1 Decentralisation

A second — usually complementary — way to augment institutional privacy on online platforms is to take away the institution itself, be it partially or as a whole. This is referred to as *decentralisation*. Besides privacy, there are other technical reasons that computer systems get decentralised. We focus on decentralisation with augmenting privacy as a goal, loosely following the authority topology of Troncoso, Isaakidis, Danezis, and Halpin. The next chapter will also describe the network topology.

When partially decentralised, this is referred to as *federation*, where providers assist users (Troncoso et al., 2017). One example of a federated platform is email, where anyone with the technical knowledge can setup an e-mail server. The owner of the server can now invite users to use his server, and email servers across the internet will exchange and route mail among themselves. When one such server is compromised, all users receiving mail from, or sending mail to this server can have their email (and metadata) compromised.

When wholly decentralised, Troncoso et al. distinguish *ad-hoc*, *friend-to-friend* and *peer-to-peer*.

In the ad-hoc paradigm, users' devices communicate directly with each other, without any intermediary participating.

In the friend-to-friend paradigm, nodes assist friends. Users' devices connect to friends' devices, and friends may help each other propagate or host data for other friends. The network uses the social graph to support hosting data.

In the peer-to-peer paradigm, users assist users. Any device on the network will be seen as equal, and nodes provide services and resources to other nodes (Troncoso et al., 2017).

1.2.2 Privacy as confidentiality

Both end-to-end encryption and decentralisation are examples in the "privacy as confidentiality" paradigm. In this paradigm, the act of collecting information is itself considered a threat, and measures are taken against it (Diaz & Gürses, 2012).

1.3 Privacy as practice

In a third paradigm identified by Gürses, 2010, focus is on making data flow transparent to the user to enhance their ability to steer their behaviour.

While designing a platform for online social media, it is important to keep these three paradigms in mind, by (1) handing end-users both course and fine grained *control* over their data; (2) making

[1] *Hint*: if you want to read WhatsApp's privacy policies, use Tor with your browser. They will not know your origin, and will give both their world wide and their European Union version. The E.U. version has been updated on 24 May 2018, the day before the GDPR came into force.

sure their data is sufficiently secured using cryptography; (3) giving end-users visual feedback on their behaviour and the flow of their data on the platform.

Since this thesis focussed on the security and cryptographic aspect of the platform, mostly paradigms (1) and (2) are applied. It is however important to consider the practical, behavioural and empowerment aspects of a platform (3), and would be interesting future work.

Decentralisation and cryptography are thus large part of the means to a private online world. In chapter 2 we will further explore existing decentralisation technologies. Chapter 3 outlines how these technologies can be applied to online social networks (OSN: online social networks). The rest of part I will develop the necessary technical background to understand the system we develop in part II.

Chapter 2

Decentralising social networks

This chapter goes a bit deeper on the technical and practical means to achieve decentralisation, exploring the advantages and disadvantages of several techniques. While the previous chapter focussed on the *authority* topology, this chapter describes the *network* topology (Troncoso et al., 2017).

2.1 Federated social networks

Federated online social networks (OSN: online social networks) are networks that are hosted by interconnected multiple independent authoritative providers. Users pick and connect to one of these providers to use the systems' services. These networks require to install and configure (usually free and gratis) server software, which requires certain technical knowledge.

This kind of system facilitates *optional* privacy, as it is still perfectly possible for the service provider to gather user data of his server instance (and possibly of connected friends of his users). In the extreme case, federated networks become centralised networks, where a single authority has near total control over the whole network.

Examples of federated OSNs are Mastodon, GNU Social, friendi.ca, and Diaspora*. Many protocols besides OSNs on the internet are federated; for example email and XMPP.

In case of email, Mailchimp (a large email marketing company) notes in 2015 that more than 70 % of their email targets Google's `gmail.com` domain. Their statistics exclude the "foreign" domains hosted on Google's and Microsoft's mail servers, which suggests an even larger market share (Khan, 2015). Other online resources suggest that both Microsoft and GMail are by far the world most popular email service providers (Lewkowicz, 2017; Datanyze, 2018). This illustrates that federated networks *may* still lead to centralisation, defeating the decentralisation and privacy-related benefits.

The case of email illustrates another drawback of federated networks: the user has to pick a provider. A quick survey on Google, Bing and DuckDuckGo results in `mail.com` and `gmail.com` as top two results, with Microsoft's `live.com` usually third for the keywords "create email account".

2.2 Peer-to-peer communication

When the service provider is taken out as a whole, and the end-user applications communicate among each other without an intermediate party, this is usually referred to as *peer-to-peer*. A *peer* refers to a single user's running software instance. Troncoso et al., 2017 differentiate between three forms.

2.2.1 Ad-hoc

In this paradigm, peers communicate directly with whom they wish to exchange information. No other nodes are contacted. In contexts where only time-local communication makes sense, such as voice calls, this is the simplest approach. Communication can be encrypted between two nodes, and this approach bypasses compromised nodes.

An academic example that makes use of ad-hoc networks is AmbientTalk (Van Cutsem, Mostinckx, Gonzalez Boix, Dedecker, & De Meuter, 2007), which is an actor based programming language for ambient object oriented programming.

2.2.2 Friend-to-friend

In this paradigm, a social graph (e.g., friends, connections, followers) is employed to make logical connections. It often needs manual bootstrapping of connections by adding some initial friends.

Privacy on the other hand can be achieved more easily, since we are only communicating with friends.

2.2.3 Peer-to-peer

In the exact sense of the term *peer-to-peer*, no nodes have more authority than others. In this paradigm, the system is *self organising*, and nodes carry out transactions for all other nodes (Troncoso et al., 2017).

Different techniques have been proposed to organise these nodes, to setup the network topology. Initial naive peer-to-peer networks used *flooding* to spread a message to the whole network, which puts a high load on the system, and has low probability of finding rare data (Shen, Yu, Buford, & Akon, 2010). An example of flooding can be found in the original design of *Gnutella*.

Besides flooding, *gossiping* (also known as *epidemic algorithms*) can lead to better performance (Demers et al., 1987). Here, messages are transmitted to a few random neighbours, modelled after how epidemics spread.

When a peer-to-peer network gets more structure, certain upper bounds can be given for certain operations. The most common form of structure are distributed hash tables (DHT: distributed hash tables) overlay networks, wherein nodes keep a minimal routing table of neighbour nodes, greedily passing messages to 'closer' nodes. Nodes store key-value pairs, and support lookup by key.

They were first introduced with CAN (Ratnasamy, Francis, Handley, Karp, & Shenker, 2001), Chord (Balakrishnan, Kaashoek, Karger, Morris, & Stoica, 2003), Pastry (Rowstron & Druschel, 2001) and Tapestry (Zhao et al., 2004).

Later on, Kademlia (Maymounkov & Mazieres, 2002) was introduced, which was used in the popular BitTorrent protocol. Kademlia has the advantage of fast $\mathcal{O}(\log n)$ lookup (where n is the amount of participating nodes in the network), and a simple routing scheme making it easier to analyse.

When adding more structure to the network topology, we can assign the role of "super-nodes" to powerful nodes (be it in terms of computing capacity, storage, link speed, or different). These super-nodes may become target for an attack, since they gain more authority than others (Troncoso et al., 2017).

Stratified network topologies assign multiple, distinguished roles to nodes. These too have a more obvious attack surface.

2.3 Data storage

When building an online social platform, data *storage* can be studied. After all, data should be stored on a location where it can later be retrieved from. We distinguish three categories of data distribution.

2.3.1 Friend-to-friend

In friend-to-friend (f2f) networks, it is common to store user data with social connections. For example, RetroShare synchronises forums to friends that are subscribed, who in turn can propagate forums to their friends (Fairchild, 2015; Soler, 2017).

One issue with this approach may be the lack of availability when your social circle is down; imagine opening the application when all your friends are asleep, and no connections can be made to update the local data.

However, storing data needs less precautions for malicious actors, since it is assumed that friends are trusted in this model.

2.3.2 A note on block chains

Another way to persistently store data is on a block chain. Usually though, this is regarded as bad practice, since all data is now replicated across all nodes. Also, since block chain is meant to be persistent, deletion of content is at least impractical without additional technology.

A popular way to circumvent this is by storing the social data off chain, only using the block chain to refer to the actual data, or even only as financial support for service providers. An example of the latter approach is the Loki direct messaging application (Jefferys, Harman, Ross, & McLean, 2018).

Of course, when only using the block chain for financial support, other techniques are still required to host the end-user data itself.

2.3.3 DHT: Kademlia

DHTs pseudorandomly distribute data across all nodes. For example, Kademlia (Maymounkov & Mazieres, 2002) is capable of retrieving values in $\mathcal{O}(\log n)$ routing steps. Kademlia is parametrised mainly by a (cryptographic) hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^b$, and an integer k.

Kademlia works by assigning every node an identifier of b bits (classically 160 bits, when using SHA-1 for $\mathcal{H}$), and hashing every object using $\mathcal{H}$. In the sense of a XOR-based metric $d(x,y)$, the k-closest nodes to an object are responsible for having it available.

BitTorrent Mainline DHT is one of the largest Kademlia based DHTs, which resists a daily churn (amount of nodes leaving) of 10 million. Kademlia appears to be *resilient* against this node churn (Wang & Kangasharju, 2013).

What data is stored on the DHT depends on the implementation. Some applications keep all their data in the DHT, while others refer to whom keeps these data.

The latter strategy is used by BitTorrent, where the DHT keeps track of what peers have access to what files. Simplified, the DHT-key is a hash of the file, and the DHT-value is a list of peers.

Another application that uses the latter strategy is PeerSoN (Buchegger, Schiöberg, Vu, & Datta, 2009), where the DHT keeps track of what peer stores a user's data.

2.3.4 Hybrids and variants

It is possible to use a hybrid architecture, for example by storing user data off-chain in block chain based technologies, or by using an extra level of indirection such as in PeerSoN (Buchegger et al., 2009).

MAIDSafe is a commercial example that uses a cryptographic currency *without* blockchain to support (social) application development (Lambert, Ma, & Irvine, 2015).

2.4 Critique on decentralisation

Decentralisation gives back the control over data to the end-user, while taking control away from the developers. It is thus uttermost important to design the applications as such that users get insight in how their data flows, who can access it, and how they can securely use the application. After all, it is impossible for a "support team" to help users that are stuck, since that would imply a (hybrid) form of centralised control. Privacy research that focusses on how technology can be used to enhance the users' understanding of data flow is called the "privacy as practice" paradigm (Diaz & Gürses, 2012).

Agre argues that "decentralised institutions do not imply decentralised architectures, or vice versa", by which he means that concepts in society that decentralisation should not be a goal *an sich*. He also argues that while the peer-to-peer (p2p) community knows that architectures are political, architectures should not be a substitute for politics (Agre, 2003).

On a more technical note, there are a few attacks on decentralised systems. Since in the peer-to-peer paradigm nodes are supposed to be indistinguishable, it is possible to attack the network by generating rogue *Sybil* nodes (Douceur, 2002). This attacks decentralised reputation system, by generating enough (often $\frac{1}{2}N$) nodes to control a vote, or consensus. Loki (Jefferys et al., 2018) requires service nodes to hold a certain amount of currency (a proof of stake) as authentication, while Advogato (Levien & Aiken, 1998) used a trust metric. More general Sybil attack prevention techniques are an ongoing research topic.

Another often mentioned attack is the *eclipse attack*, also known as a *routing table poisoning* attack. The attacker tries to fill the DHT routing table of the target, such that it is unable to communicate with the correct peers. This is easily circumvented by requiring the node id to be pseudorandom, for example by taking the hash of a public key, and attaching a proof of knowledge for the corresponding private key.

Note that some authors refer to federation as distribution (as in distributing users over several service instances). Usually though, it is clear from the context what is meant.

While research on decentralisation has in no way achieved perfect solutions, systems like BitTorrent Mainline DHT have gotten popular and can easily handle large amounts of users and churn. Peer-to-peer networks clearly offer the best privacy guarantees, while federated networks

like email can be easier to maintain. Federation however often leads to *re*centralisation, which can harm privacy. Chapter 3 will explore existing OSNs. It will go in depth on how they can be built on decentralised architectures, and why building OSNs on a peer-to-peer architecture is seemingly difficult.

Chapter 3

Existing online social networks

ONLINE SOCIAL NETWORKs (OSN: online social networks) come with different purposes, but their privacy-related properties can be clearly identified. Different OSNs are compared, both commercial, gratis, academic, free, and non-free, in terms of their privacy related aspects.

3.1 Classical social networks

Classically, OSNs are implemented as a web application. Users authenticate themselves with a username-password combination in their browser, and authentication is persistent throughout the session using one or more cookies.

Data is stored in a central database (typically SQL, or object storage or NoSQL), and the web application guards the database, checking permissions for every user.

Web pages are typically generated *on-the-fly* with PHP or a similar scripting technology, or even dynamically reloaded using AJAX.

Examples of these "classical" networks are Facebook, Instagram, LinkedIn, and Twitter.

3.1.1 Mobile apps

A second category of OSNs are built using mobile "apps", typically targeted to Android, iOS and Windows Phone. Often, these are applications that are tied to their web-counterpart (Facebook, Instagram, and Twitter), although there are also mobile-only apps (Snapchat being a popular example).

They typically still use a centralised (usually private) application programming interface (API), which they contact — often even using HTTP/REST (Fielding & Taylor, 2000) — to exchange data.

3.2 Solving the privacy problem

Different new OSNs have popped up that try to account for the privacy concerns that arise from centralisation (section 1.2). They implement different technologies to overcome those issues. This section enumerates solutions that were proposed in the past.

	Federated	Distributed	Peer-to-peer	e2e encrypted	Free software	OStatus	Offline access	User friendly	Expiring content	Anonymized	Easily extensible
Facebook	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
Instagram	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
LinkedIn	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
WhatsApp	✗	✗	✗	✓	✗	✗	✓	✓	✗	✗	✓
Signal	✗	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓
Twitter	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
Mastodon	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗	✓
PeerSoN	N/A	?	✓	✓	N/A	✗	?	N/A	✗	✗	✗
RetroShare	N/A	✗	✓*1	✓	✓	✗	✗	?	✗	✗	✓
GNU Social	✓	✗	✗	✗	✓	✓	✓	✓	✗	✗	✓
Telegram	✗	✗	✗	✓	✗	✗	✓	✓	✗	✗	✓
SnapChat	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✓
friendi.ca	✓	✗	✗	✗	✓	✓	✓	✓	✓	✗	✓
*Diaspora**	✓	✗	✗	✗	✓	✗	✓	✓	✗	✗	✓
SecuShare	N/A	✗	✓	✓	✓	✓	✓	✓	?	✓	?
MAIDSafe	N/A	✓	✓*2	✓	✓	✗	✓	?	✗	✗	✗
Pandora	N/A	✓	✓	?	✓	✗	✓	?	✗	✗	✗

Table 3.1 – Social networks and their identified properties. Peer-to-peer OSNs typically do not offer an abstraction layer for developers (Easily extensible), while this is a trivial property of centralised or federated OSNs.
✓: The social network has this property.
✗: The social network does not have this property.
?: This property is unknown or unclear.
N/A: Not applicable in the context (e.g. federation does not make sense for a peer-to-peer network).
*1: RetroShare is technically friend-to-friend (section 2.2.2)
*2: MAIDSafe uses a stratified peer-to-peer system, whereby nodes are highly specialised in multiple different roles.

3.2.1 Federated systems

As explained in section 2.1, social networks like Mastodon, and Diaspora* are meant to be installed by many independent service providers. Independent installations (also called "pods") communicate with each other, thus spreading the data.

In this case, the end-user has to put his trust in a single service provider, since the service provider can access all data on a pod.

Note that this service provider can be a family member, a company, or a friend. This way, the end-user can actually choose with whom they put their trust, limiting the effects of a malicious service provider.

Another advantage of these systems, in context of online social networks, is that they can largely employ the same classical web technologies.

The listed federated networks (cfr. also table 3.1) are all free software. This should not surprise, since these are typically developed in the "privacy as confidentiality" paradigm, where "don't trust the developer, trust the code" is often the saying.

OStatus OStatus is a set of protocols that is allows certain (usually federated) OSNs to interoperate table 3.1.

3.2.2 Supported by cryptographic currencies

Where classical OSNs typically rely on data mining (cfr. section 1.1) for profit, private social networks do not, by nature. Several alternative profit models are being developed, and quite a part of them are based on cryptographic currencies. Most cryptographic currencies are based on block chain.

Block chain technology (Haber & Stornetta, 1991) uses a Merkle tree (a Merkle list) of records as a database. Block chains were first conceptualised in Bitcoin (Nakamoto, 2008) as a peer-to-peer, distributed ledger.

Block chains have a high Byzantine fault tolerance, which means it can achieve consensus on the network state.

Originally, in Bitcoin, block chains were developed to overcome double spending (through a consented state).

Blockchain as a database

Blockchains are capable of CRUD-like[1] database operations, and behave like event-sourced databases: instead of storing the database state itself, the full history gets recorded. The database state can then be recovered by replaying history, and history gets checked by digital signatures.

Since the whole history is stored, deletion or alteration of data does not erase nor update data; it merely records a new updated or empty version. Due to preserving the whole history, erasure of privacy sensitive information is nearly impossible.

Blockchain limitations

At time of writing, all current blockchain based systems require a participating node to store the whole blockchain history on his machine. The history of a blockchain will, initially, grow exponentially, while (in the case of the Bitcoin blockchain) being capped at 1 MB every 10 minutes.

[1] CRUD: create-read-update-delete. These operations are considered the four basic functions of persistent storage.

This means that it will become impossible for smartphones, tablets or even computers with a small disk to actively participate in the network.

Blockchain is believed to be a hype (Wilson, 2016) by some, since it is getting used in many applications that do not require network-wide state consensus.

Several solutions to this problem are being developed to overcome this ever-growing system.

For example, MAIDSafe has been developing an OSN, as part of their Autonomous Data Network (Irvine, 2010), based on a non-block chain cryptographic currency (Lambert et al., 2015).

It is of course possible to store the OSN data off-chain, keeping the cryptographic currency as financial incentive for hosts (Jefferys et al., 2018).

Social networks do not have a concept of "double spending", nor do they require network-wide consensus on state. Therefore, the use of block chain technology seems unnecessary for the data of the OSN itself.

Friend-to-friend

Several *fully decentralised*, *peer-to-peer* online social networks have been proposed. Note that strictly spoken blockchain based social networks would fall in this category; this section will describe systems where data is hosted on their own peer-to-peer network.

One category of fully decentralised networks are friend-to-friend networks (f2f), e.g. RetroShare.

Strictly spoken (and RetroShare is a good example thereof) friend-to-friend networks only communicate from one friend to another, as opposed to considering all network nodes equal.

For example, when Alice creates a forum on RetroShare, she announces the existence of the new forum to her friend Bob.

Messages written on the forum get propagated from friends to neighbouring friends, based on their subscription preferences.

As a drawback, those friend-to-friend networks only work at their best when most members of a friend group are online at the same time. It is possible that certain data will not be accessible when a subgroup of connections are offline (cfr. table 3.1 and section 2.3.1).

One interesting development, published one month after I proposed the topic of this thesis, is RetroShares's generic data exchange system (GXS). This system knows two different types of objects, groups, and messages. Groups are the top level data structure of a *service*, and messages are posted in groups. *Circles* are sets of identities, and groups can be restricted to certain circles. Groups and messages are the data structures that are stored in a decentralised fashion.

Notably, the GXS *takes abstraction* of data storage on a decentralised platform. On top of this abstraction, RetroShare builds forums, and streaming channels, and they plan on further building decentralised websites, version control management, and blogging on top. Identities and circles are recursively implemented as a service themselves (Soler, 2017).

Peer-to-peer based networks

When one lifts the restriction of communicating only with friends, the system can overcome these accessibility issues.

An example OSN that makes use of this property is PeerSoN (Buchegger et al., 2009). This research project uses a distributed hash table (DHT) (cfr. section 2.3.3) to store and retrieve files with information and meta-information about friends and their activities, much how like BitTorrent uses their Mainline DHT.

The DHT is used to retrieve a the *location* of a file, namely, the peers that are online and store the requested file. This request can thus answer with multiple nodes, if multiple copies of a file are available.

PeerSoN also requires a login procedure, much like the DHT announcement in BitTorrent Mainline DHT, where a node "logging in" announces its IP and the list of files stored.

In classical social networks (section 3.1), based on a central web server, the developers use a classical relational database management system (RDBMS), wherein they store their data.

This is partly true for federated online social networks (sections 2.1 and 3.2.1) too. Due to the nature of the service providers being stable and requiring setup, the protocols to interconnect the individual nodes can also be based on HTTP or REST.

In peer-to-peer based systems however, developing of features requires a recurring thought process: storing data for a user requires the developer to think about both the storage format, the cryptography, and usually the remote procedure call (RPC) methods. What most of those projects thus have in common, is the necessity to develop a new protocol (or a protocol extension) on a per feature basis; they lack abstraction layers for developers.

Moreover, this also means that older versions of the OSN software will usually not support storage of data for newer features; they lack forward compatibility. The notable exception is RetroShare, who recently introduced the GXS (Soler, 2017).

The rest of this thesis will build a set of common *building blocks* on which an OSN can be developed. The idea is to provide an abstract data model — just like in SQL — and a coupled authentication model on which private OSNs can be built. Before getting there however, a common notion of certain cryptographic primitives is required. Chapter 4 will introduce the basic cryptographic knowledge on which our abstractions will be built.

Chapter 4

Cryptography

In peer-to-peer networks, data is shared and stored with adversaries. In this context, cryptographic techniques are usually employed to make clear text unintelligible, and non-malleable for those adversaries, in order to guarantee intact delivery to the recipient.

4.1 Elliptic curve cryptography

Koblitz and Miller independently suggested elliptic curves for use in cryptographic systems around 1985 (Koblitz, 1987; Miller, 1986), as an alternative to the classical Galois-field Diffie-Hellman key-exchange (Diffie & Hellman, 1976).

Elliptic curves require smaller keys (e.g. $2b$ bits in Bernstein, 2006) on the same b-bit security level than their Galois(-prime)-field counterparts (for instance 2048 bits for a 112-bit security level). Elliptic curve cryptography is also able to beat classical Diffie-Hellman in terms of performance and ease of implementation; Curve25519 is currently one of the most performant elliptic curve based Diffie-Hellman primitives (Bernstein, 2006).

Elliptic curve cryptography is based on the algebraic structure of elliptic curves. For our purposes, we will be using a Montgomery curve (Montgomery, 1987). These curves consist of the points satisfying the equations

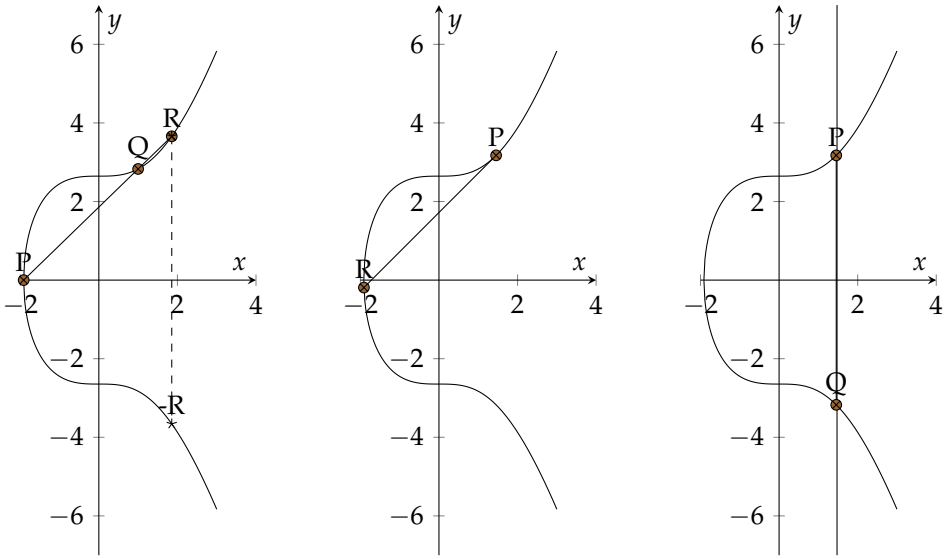
$$E : y^2 = x^3 + ax + b,$$

and the so-called "point-at-infinity" (denoted ∞).

The points (x, y) of these points are most often taken over a *finite* field $\mathbb{F}_q$ of order q. For getting some intuition in them they are often considered over the real numbers, since they can then be easily plotted (for example in fig. 4.1), and the operations we construct can be interpreted geometrically.

4.1.1 Elliptic curve operations

We can define a group operation (called the addition law for elliptic curves, $\cdot + \cdot$) on points on the elliptic curve. Note that elliptic curves are symmetric over the x-axis (cfr. fig. 4.1), thus we denote the inverse of the point P to be $-P$, the opposing point, and $\infty = -\infty = 0$ as neutral element.



(a) A regular addition; $P + Q + R = 0$: using the third intersection of PQ

(b) Point doubling of P; $2P + R = P + P + R = 0$, using the tangent in P.

(c) Addition of opposites; $P + Q = P - P = \infty$, resulting in the point-at-infinity

Figure 4.1 – The elliptic curve $y^2 = x^3 + 7$ over the real numbers, displaying the three-part addition law for elliptic curves.

We geometrically construct the sum of the points P and Q: draw a line intersecting P and Q. The third intersection point of E and PQ is the point R, and we define $-R = P + Q$, and the sum is thus the opposing point of R, as shown in fig. 4.1a.

This construction generally works, except in the following cases:

$P = Q$ When P and Q are equal, we use the tangent of the elliptic curve in P, as shown in fig. 4.1b. This is called point doubling, since $P + Q = P + P = 2P$;

P **or** Q **equals** ∞ Assume $Q = \infty$, then $P + Q = P$. ∞ is the identity of the group. One can also geometrically construct this point, with a vertical line through P as "intersection" between P and ∞;

$P = -Q$ The vertical line does not intersect in a third point, thus $P + Q + R = P - P + R = R = \infty$, shown in figure fig. 4.1c;

Q **is an inflection point** we take $R = Q$, and thus $P + Q = -Q$. This is a limit case of the regular addition law, where Q moves to an inflection point, and R moves as a result to Q.

Now that we constructed an addition operation, we also have a scalar multiplication. Scalars will be denoted with lower case symbols, while group members will be denoted with upper case symbols. For example, $R = rG$ denotes that R is the scalar multiplication of r with G, i.e.

$$R = rG = \underbrace{G + G \cdots + G}_{r \text{ times}}.$$

4.1.2 Efficient elliptic curves

The security related operations on elliptic curves will rely on the scalar multiplication operation. For elliptic curves to be efficient, scalar multiplication needs to be fast. Multiplication of an integer a with the group element G naively takes $a - 1$ addition operations. Knowing that a is usually in the order of 255 bits, this seems inefficient.

A faster way of calculating $A = aG$ is called the double-and-add method.

Algorithm 4.1.1 (Double-and-add method)**.** *Now, set $A \leftarrow \infty$, and $G' \leftarrow G$. Write the binary representation of a: $a = a_0 + 2a_1 + 4a_2 \cdots + 2^b a_b$. Now, for every $i \in [0; b]$,*

1. *if $a_i = 1$, add G to A: $A \leftarrow A + G$.*
2. *double G': set $G' \leftarrow 2G'$*

This algorithm requires, on average, $b + \frac{b}{2} = \mathcal{O}(b)$ additions. This is exponentially less than the naive scalar multiplication, which required $\mathcal{O}(a) = \mathcal{O}(2^b)$ additions. Also note that G' can be precomputed if the same G is used multiple times, which saves another b additions. This does not change the asymptotic complexity, but might introduce cache timing attacks. Since most of the elliptic curve operations in elliptic curve cryptography use a common *basepoint* (often called G), it is worthwhile to precompute Gs 2^i multiples.

4.1.3 Discrete logarithm problem

In groups *without additional structure*, the discrete logarithm problem (DLP) is supposed to be hard: given points P, Q, find the scalar a such that $aP = Q$.

The name of the discrete logarithm problem originates from the original Diffie-Hellman definition, where exponentiation was used instead of scalar multiplication: given $A = g^a$, find a, the (discrete) logarithm of A: $\log_g(g^a) = a$.

4.1.4 Cryptography on elliptic curves: key exchange

Note that the discrete logarithm problem gives us a trapdoor function. Alice and Bob agree on a public *base point* G. Alice has a secret scalar a, and sends $A = aG$ to Bob. Bob has a secret scalar b, and sends Alice $B = bG$. Alice and Bob can now calculate the shared secret $S = aB = bA$, and use this as a shared secret key.

Note that only A and B travel across the (potentially insecure) channel, so an eavesdropper has to solve the discrete logarithm problem to calculate a or b.

The problem of finding $S = abG$ given aG and bG is called the computational Diffie-Hellman problem (CDH). Solving CDH is equivalent with breaking the key exchange (Maurer, 1994), but not with the DLP; computing a discrete logarithm solves CDH, but not necessarily vice versa. This is an open research problem. The problem of distinguishing S from a random group element (which makes S suitable as seed for a key) is called the decisional Diffie-Hellman problem (DDH). It is considered a harder problem than CDH, since CDH can be used as a distinguisher (Boneh, 1998).

4.2 Monero: decentralised fungible digital currency

One technology that requires privacy is digital cash (Okamoto & Ohta, 1991). Bitcoin (Nakamoto, 2008) has been a successful attempt of a decentralised electronic ledger. Unfortunately,

it suffers from several deficiencies with respect to privacy: in order to prevent spending the same cash twice, it relies on the public availability of all transaction data. This includes the amount of money, the source and destination of the money.

Monero, based on the CryptoNote protocol (van Saberhagen, 2013), attempts to solve these problems using some interesting cryptographic primitives, some of which Glycos uses in its construction. Monero attempts to satisfy two conditions (van Saberhagen, 2013; Kumar, Fischer, Tople, & Saxena, 2017):

Untraceability a transaction is (equiprobably) anonymous among several senders;

Unlinkability it is impossible to prove two transactions went to the same destination.

In a later version, Monero also achieves to hide the amount of cash being sent using homomorphic encryption; this technology is called RingCT (Noether, Mackenzie, et al., 2016).

In what comes, parts of the technology behind Monero is described, since it served as inspiration for the anonymisation processes used further on. This section is heavily based on Kumar et al., 2017, which gives a succinct overview of Monero's cryptography.

4.2.1 System parameters

For asymmetric cryptography, CryptoNote uses the Curve25519 elliptic curve by Bernstein (Bernstein, 2006).

q	a large prime number
$\mathbb{F}_q$	a finite field of order q
$E(\mathbb{F}_q)$	an elliptic curve on $\mathbb{F}_q$
G	a base point on $E(\mathbb{F}_q)$
ℓ	the prime order of G
$\mathcal{H}_s$	a cryptographic hash function $\{0,1\}^* \to \mathbb{F}_q$
$\mathcal{H}_p$	a cryptographic hash function $E(\mathbb{F}_q) \to E(\mathbb{F}_q)$

Table 4.1 – Monero's system parameters, using Curve25519 elliptic curve cryptography (Bernstein, 2006)

The cryptographic primitives used in Monero are outlined in table 4.1. The CryptoNote whitepaper (van Saberhagen, 2013) is the reference of choice for further details.

As in Bitcoin, each Monero user has a long-term key pair denoted by ($pk_{\text{LT}}^{\text{user}}$,$sk_{\text{LT}}^{\text{user}}$). These are *two* classical public, private key pairs of an elliptic curve cryptosystem:

$$\begin{aligned} sk_{\text{LT}}^{\text{user}} &= (a, b), \\ pk_{\text{LT}}^{\text{user}} &= (A, B) = (aG, bG). \end{aligned}$$

4.2.2 Unlinkability

To achieve anonymity for the destination of a transaction, Monero employs one-time public keys, derived from Bob's public key (van Saberhagen, 2013; Kumar et al., 2017).

These are referred to as "stealth addresses" in certain literature, since these conceal the actual payment destination. They were first discovered by the user 'bytecoin' on the Bitcoin forums

(user 'bytecoin', 2011), and later improved (van Saberhagen, 2013; Todd, 2014; Courtois & Mercer, 2017).

In order to pay Bob, Alice generates an ephemeral public key (the one-time key), using Bob's $pk_{\text{LT}}^{\text{Bob}}$ = (A, B):

$$\begin{aligned} r &\leftarrow [1, \ell - 1] \\ R &\leftarrow rG, \\ pk_{\text{OT}}^{\text{Bob}} &\leftarrow \mathcal{H}_s(rA)G + B. \end{aligned}$$

R is referred to as the transaction public key (TX public key), and is appended to the transaction. Now note that

$$\begin{aligned} pk_{\text{OT}}^{\text{Bob}} &= \mathcal{H}_s(rA)G + B = \mathcal{H}_s(raG)G + B \\ &= \mathcal{H}_s(aR)G + bG = (\mathcal{H}_s(aR) + b)G \\ &\Downarrow \\ sk_{\text{OT}}^{\text{Bob}} &= \mathcal{H}_s(aR) + b \end{aligned}$$

Since only Bob knows (a, b), only he can derive his secret key $sk_{\text{OT}}^{\text{Bob}}$, using R, the publicly known TX public key. Courtois and Mercer, 2017 gives an extensive review for different methods of deriving stealth addresses.

Note that this can be seen as a variation on elliptic curve Diffie-Hellman key-exchange (Law, Menezes, Qu, Solinas, & Vanstone, 2003) with a random key r:

$$S = rA = raG = arG = aR$$

4.2.3 Untraceability

To achieve anonymity among several senders, Alice selects different source transactions $\mathcal{S} = \{t_1, t_2, \ldots, t_n\}$ containing the same amount of Monero. She then creates a linkable ring signature (Rivest, Shamir, & Tauman, 2001) based on Fujisaki and Suzuki, 2007 over all transactions in $\mathcal{S}$. She uses a non-interactive zero knowledge proof (NIZKP) to commit to a single source transaction. And as such, double spending is made impossible, while preserving the anonymity guarantees the ring signature provides.

Monero is an interesting case study. They don't only use elliptic curve cryptography as a way to sign data, also in a scheme to create anonymity. Moreover, their use in anonymity is twofold: at one hand, they use randomised *ephemeral public key*, and at the other hand they use *ring signatures*.

These two primitives will come in handy when constructing our distributed encrypted graph database in part II. Before we go there, we will first introduce the concept of graph databases in chapter 5, the last chapter of part I.

Chapter 5

Resource Description Framework

Resource Description Framework (RDF) is a W3C specification originally designed as a metadata data model[1]. It is similar to classical modelling approaches such as entity-relationship (ER) or object relational mapping (ORM). RDF stores its data in the form of semantic triples: *subject-predicate-object* (s, p, o).

As an example, one can state the fact "Bob is a friend of Alice" as "Alice" (subject o), "hasFriend" (predicate p) and "Bob" (object o). To demonstrate that RDF triples form a graph, one relates every triple to a labelled edge, that connects the subject to the object, and has the predicate as label.

RDF is not tied to a specific serialization format (file format); rather, it has several different serialization formats. For example, if we want to say that Alice has a friend Bob, we can express this in Turtle (Carothers & Prud'hommeaux, 2014) notation (visualised in fig. 5.1) using

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

<#Alice>
  a foaf:Person ;
  foaf:name "Alice" ;
  foaf:knows [
    a foaf:Person ;
    foaf:name "Bob"
  ].
```

These seven lines contain five triples; they state

1. `#Alice` is a Person;
2. `#Alice`'s name is Alice;
3. `#Alice` knows a person (without an identifier);
4. that person is called Bob;

[1]This description of RDF is based on Wikipedia's description at https://en.wikipedia.org/wiki/Resource_Description_Framework, licensed under the Creative Commons Attribution-ShareAlike License

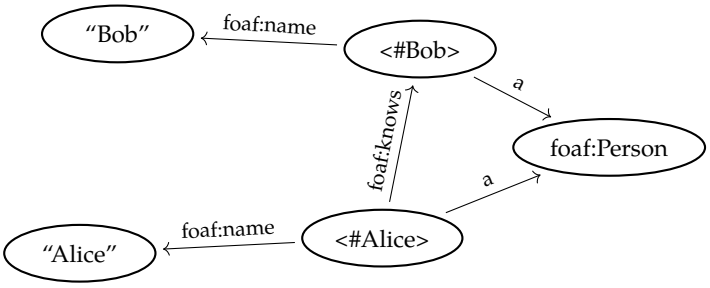


Figure 5.1 – An example RDF graph. This graph signifies that Alice, a person, knows Bob, another person. Both persons have a name and type.

5. that person is a Person, obviously.

In the preamble of the Turtle document, it is stated that several schema's, or *ontologies* are used in the thereafter following document. Among those are basic RDF, and RDF/S, the basic RDF Schema constructs (Guha & Brickley, 2014). More importantly, FOAF (Brickley & Miller, 2014) is used, which is a standard ontology for social networks and friendship concepts.

5.1 Querying the data model

Queries can be ran on RDF data models using SPARQL. If Eve, for example, wants to know about Bob's current acquaintances, she might use a query like

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/> .
SELECT ?friends
WHERE {
  ?bob a foaf:Person ;
       foaf:name "Bob" ;
       foaf:knows ?friends.
}
```

This query returns the `?friends` variable, for which the three given triples evaluate to `true`:

1. `?friends` must be the object of `foaf:knows`, of which the subject is a variable `?bob`;
2. `?bob` must be a person;
3. The name of the Person `?bob` must be "Bob".

RDF has been used in the context of social network modelling before, cfr. e.g. San Martín and Gutierrez, 2009, who use a social network modelled using RDF for analytic purposes.

We will however combine the database-like structure of an RDF-graph with cryptography and a peer-to-peer network, to create an *abstraction* for building a private online social network (OSN). Chapter 6 gives a global overview of how we combine RDF graphs and peer-to-peer networks, while subsequent chapters describe how to protect the graph using cryptography.

Part II

System description

Chapter 6

High-level system overview

Recall that we are building a peer-to-peer and extensible online social network (OSN). Before going into detail on how it works, let us look at a high-level overview of the whole system. This chapter will therefore introduce the high-level procedures and design considerations, as to give intuition in how the system functions as a whole, and how the system may be used in an OSN.

6.1 Peer-to-peer

The system uses a structured peer-to-peer networking model (section 2.2.3, nodes assist other nodes), based on the Kademlia DHT (cfr. section 2.3.3, and Maymounkov and Mazieres, 2002). Data stored on the system is uniformly distributed over all participants, and all participants are equivalent nodes on the network.

This means that Alice's data are potentially stored with a malicious participant Mallory, and access to this data needs protection. Chapter 7 elaborates on the cryptography used to protect, anonymise, and in other ways shield these data from Mallory.

6.2 Generic data storage

Glycos is designed to be *trivially extensible*, by providing the developer with a generic data storage interface, in the form of a graph database.

This is conceptually the same as how web applications are developed: usually, those applications are backed by an relational database management system (RDBMS), which provides generic storage of application developer defined objects and their relations, and keeps track of relation definitions.

Relational databases offer a layer of abstraction that is useful to the application developer. This is contrasted by how state-of-the-art (SOTA) peer-to-peer (p2p) systems like PeerSoN (Buchegger et al., 2009) or SecuShare are developed; every applications feature mandates a corresponding protocol or data format. Designing these protocols requires consideration of the adversary model, a cryptographic implementation, an approach to integrate the data model into the network, and all clients are required to switch to the newest version for better compatibility.

RetroShare has seen this problem, and made an effort to partially solve this in friend-to-friend context (section 2.3.1), by introducing abstract concepts like *objects*, *groups*, and *messages* (Soler, 2017).

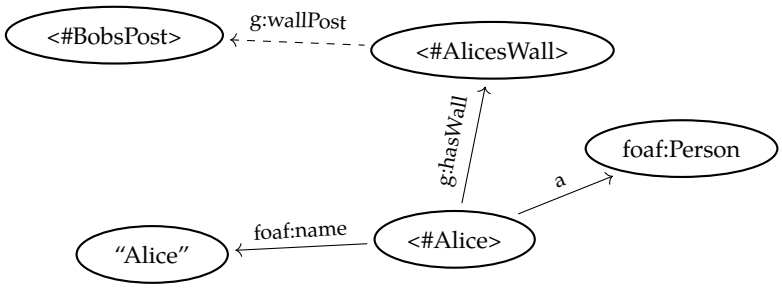


Figure 6.1 – Bob writes a message on Alice's wall. This is only possible if Alice has granted Bob the rights to do so; otherwise, the network will not accept Bob's post (`<#BobsPost>`).

Glycos also takes abstraction at the protocol level of both the data model, and the privacy attributes. Data is stored in a graph database (Angles and Gutierrez, 2008 for an overview), with privacy attributes attached to every vertex. Note that this approach is more generic than RetroShare's GXS, since RetroShare's GXS data structures can be trivially embedded in a graph database.

Privacy and security related attributes include

ownership the creator of a vertex (the *owner* Alice) has the power to update or delete the vertex;

append rights Alice can assign rights to Bob or Carol to append to a vertex, without revealing their identities to Mallory. Bob can now, without revealing his identity to Mallory, associate vertices through predicates to Alice's vertex.

read rights Alice can hand out read rights to Bob or Carol, without revealing their identities. They can now read the data associated with a vertex; i.e. the predicate and object identifier.

Vertex relationships (through predicates/edges) are hidden among a group of n equiprobable vertices, to combat statistical attacks on the graph structure. Vertex relations are, to an outsider, indistinguishable, but easily filtered by a party with read rights.

Append rights can be verified by an outsider, while this outsider solely learns the correctness of the connection, not its origin nor the destination.

In the following examples, consider Alice and Bob to be the respective creators, and thus owners of the vertices `<#Alice>` and `<#Bob>`.

Example 6.2.1 (Bob posts on Alice's wall)**.** *If, w.r.t. fig. 6.1, Bob is listed in the access control list of vertex* `<#AlicesWall>`*, only then, Bob can write* `<#AlicesWall> g:wallPost <#BobsPost>`*.*

Note that the access control list of `<#AlicesWall>` is anonymised; only Alice and Bob know that Bob is listed. An adversary cannot distinguish entries in the access control list.

Additionally, if Bob makes another post to Alice's wall, the adversary cannot link these posts; the adversary does not know that both posts were written by the same author.

On a high level, Glycos consists of a graph database and access control. Vertices have owners, in our model. These owners can assign rights to their peers to "grow" the graph, thus creating social interactions through the database.

We are still left with the actual implementation of this model. Chapter 7 will first lay down the exact cryptographic primitives, and then finally chapter 8 will explain how the cryptography, graphs and networking comes together.

Chapter 7

Cryptographic primitives

For anonymising the graph, for providing authentication and integrity, and for confidentiality, several different cryptographic techniques are employed. This chapter explains in detail which cryptographic implementations are used, however, they can conceptually be replaced with equivalent primitives.

7.1 Hash functions

A typical hash function maps an arbitrary length n-bit string $\{0,1\}^*$ to a fixed length output bit string $\{0,1\}^n$. One can also consider hash functions into a finite field $\mathbb{F}_q$. This is useful when output of a Diffie-Hellman key exchange is to be used in a secret key, such as in Monero's $\mathcal{H}_s$ (cfr. section 4.2).

The CryptoNote whitepaper (van Saberhagen, 2013) does not clearly define $\mathcal{H}_s$, beyond a cryptographic hash function $\{0,1\}^* \to \mathbb{F}_q$. In NIST SP 800-185, Appendix B ("Hashing into a Range (Informative)", Kelsey, 2016) describes a way to hash into an integer in a range. For our uses, $\mathcal{H}_s(X)$ is defined according to that appendix:

1. Let $k = \lceil \log_2 q \rceil + 128 = \lceil \log_2 2^{255} - 19 \rceil + 128 = 383$
2. Call cSHAKE 256$(X, 384)$[1] = SHAKE 256$(X, 384)$
3. Interpret the cSHAKE output as an integer N, let $\mathcal{H}_s(X) = N \mod q$

For all other standard hash function, we are using SHA 3 (Dworkin, 2015) based on the Keccak sponge function (Bertoni, Daemen, Peeters, & Van Assche, 2009).

7.2 Ephemeral public keys

Glycos uses as part of the anonymisation process *ephemeral* public keys. Just like symmetric ephemeral keys as used in perfect forward secrecy algorithms, these keys are used for one "session", and just like regular public keys, only the holder of the corresponding private key can carry out signatures.

[1] Note that we choose $384 > k = 383$, for clean byte-alignment

In principle, this is based on ByteCoin's Stealth Addresses (user 'bytecoin', 2011; Todd, 2014); we use Monero's one-time address variation (van Saberhagen, 2013), without their tracking key (cfr. section 4.2).

All users on the network have a *single* (as opposed to Monero's double) long-term key-pair:

$$\begin{aligned} sk_{\text{LT}}^{\text{user}} &= a, \\ pk_{\text{LT}}^{\text{user}} &= A = aG. \end{aligned}$$

Algorithm 7.2.1 (Generate an ephemeral public key)**.** *Bob generates an ephemeral public key for Alice as follows:*

$$\begin{aligned} r &\leftarrow [1, \ell - 1] \\ R &\leftarrow rG, \\ pk_{OT}^{alice} &\leftarrow \mathcal{H}_s(rA)G + A, \\ sk_{OT}^{alice} &\leftarrow \mathcal{H}_s(aR) + a. \end{aligned}$$

Adding A ensures that Bob cannot impersonate Alice, while Bob can reveal to his friends that the private key belongs to Alice by revealing r, if he would want to.

This key clearly belongs to Alice: only Alice knows the integer a required to construct her secret key. She can recognise that this key belongs to her by checking whether A' equals $pk_{\text{OT}}^{\text{alice}}$ in

$$A' = \mathcal{H}_s(aR)G + A.$$

These ephemeral public key system will be employed to anonymise key ownership; Alice can publicly send a message to Bob, without Mallory knowing the identity of the targeted party Bob.

7.3 Schnorr signature

Digital signatures are a means to authenticity, non-repudiation and integrity to a message. They will be used where we want to identify and validate a specific user.

We will require digital signatures to be compatible with generated ephemeral public keys (section 7.2). This implies we cannot use standard Ed25519 signatures, since the secret key p is used as a master secret, and not as scalar multiplier, when associated with the public key. This would prohibit Bob from constructing an ephemeral public key from Alice's key, without her knowing the corresponding secret key.

Schnorr signatures (Schnorr, 1991a) on the other hand, do not need a key derivation step. Since the patent (Method for identifying subscribers and for generating and verifying electronic signatures in a data exchange system, 1991b) expired in 2008, they are gaining popularity. Our signature is a variation on the original Schnorr signature, proposed in Bellare, Namprempre, and Neven, 2009, and adapted for elliptic curves. In the paper by Bellare et al., this scheme is called $\mathcal{BNN} - \mathcal{IBS}$, an *i*dentity-*b*ased *s*ignature scheme named after the authors.

Algorithm 7.3.1 (Generate a Schnorr signature)**.** *Bob signs the message M with his secret key $x = sk^{Bob}$. His corresponding public key is $Y = pk^{Bob} = xG$.*

1. *Choose a random k from $\mathbb{F}_q$.*

2. *Let* $R = kG$
3. *Let* $e = \mathcal{H}_s(R||M)$
4. *Let* $s = k - xe$
5. *The signature now is the tuple* (R, s)

In classical Schnorr signatures, the signature would be (s, e). Note that R can be calculated from s, e, and Y; vice versa, e can be calculated from R, and s:

The $a||b$ notation here means the result of concatenating a and b. We might want to use Keccaks TupleHash construct in the future for this (Bertoni et al., 2009; Kelsey, 2016).

$$\begin{aligned} R &= sG + eY \\ e &= \mathcal{H}_s(R||M) \end{aligned}$$

Alice now does not need the public key Y from Bob; she can derive it from the signature (R, s). This saves 32 bytes of storage, since Y does not require storing.

Theorem 7.3.2 (Public key recovery)**.** *Given a correctly signed message M with signature (R, s), Alice correctly recovers the public key Y*

$$Y' = \mathcal{H}_s(R||M)^{-1}(RsG)$$

Proof.

$$\begin{aligned} Y' &= \mathcal{H}_s(R||M)^{-1}(RsG) \\ &= e^{-1}(kG - (k + xe)G) \\ &= e^{-1}xeG \\ &= xG \\ &= Y \end{aligned}$$

□

Alice verifies Bob's signature (R, s) as follows:

Algorithm 7.3.3 (Validating a Schnorr signature)**.** *Bob transmits his signature (R, s) to Alice. Alice runs*

1. *Let* $R_v = sG + \mathcal{H}_s(R||M)Y$
2. *Return* VALID *if* $R_v = R$

Theorem 7.3.4 (Schnorr correctness)**.** *A correctly signed message will verify correctly*

Proof.

$$\begin{aligned} R_v &= sG + eY \\ &= (k - xe)G + eY \\ &= kG - xeG + exG \\ &= kG \\ &= R \end{aligned}$$

and $e_v = \mathcal{H}_s(R_v||M) = \mathcal{H}_s(R||M) = e$ □

Of course, correctness is not the only property we require from a digital signature. More importantly is perhaps unforgeability. These proofs get a bit more involved though, and are out of scope for this thesis. They can be read in the original paper (Bellare et al., 2009).

7.4 The ring signature

As ring signature, the scheme from Appendix A from Abe, Ohkubo, and Suzuki, 2002 is chosen, since it is easy to implement, and we do not require linkability or traceability, like Monero's ring signatures provide (cfr. section 4.2). Ring signatures have "weaker" security properties; they provide equiprobable authentication among a set of senders (called a ring), non-repudiation over the same set, and integrity over the same set. This means that a verifier can only learn that the signer possesses (at least) one private key in the set, every key has the same probability of being used.

The translation from standard discrete logarithm to elliptic curve DLP is straightforward, almost verbatim from Abe et al., 2002:

Algorithm 7.4.1 (Generate ring signature)**.** *A signer with secret key x_k signs message m with public-key list $\mathcal{R}_s$*

1. *Select $\alpha, c_i \leftarrow [0, \ell - 1]$ for $i = 0, \ldots, n-1, i \neq k$, and compute $z = \alpha G + \sum_{i=0, i\neq k}^{n-1} c_i Y_i$*
2. *Compute*

$$c = \mathcal{H}_s(\mathcal{R}_s||m||z)$$
$$c_k = c - \sum_{i=0, i\neq k}^{n-1} c_i \mod q$$
$$s = \alpha - c_k x_k \mod q$$

3. *Return $\sigma = (s, c_0, \ldots, c_{n-1})$*

Algorithm 7.4.2 (Verify ring signature)**.** *A verifier verifies signature σ. (L, m, σ) is valid if*

$$\sum_{i=0}^{n-1} c_i \cong \mathcal{H}_s\left(\mathcal{R}_s||m||\left(sG + \sum_{i=0}^{n-1} c_i Y_i\right)\right) \mod q$$

These signatures will be used where the signer needs to prove authenticity without giving away their (pseudonymous or real) identity.

We constructed two techniques for anonymisation. The first one are the *ephemeral public keys*, where, given Bobs public key, Alice derives a random "stealth key". This stealth key can be recognised by Bob, and he can derive the associated private key.

The second anonymisation technique are ring signatures. When Alice wants to contact Bob and needs to *prove a certain membership*, and does not want Mallory to know which member she is, she will use a ring signature.

Both these anonymisation techniques will be used when constructing the peer-to-peer graph database in the next chapter; the ring signature will be useful to hide which one in an access control list can alter data, while the ephemeral public keys can be used to setup anonymous access control lists. The Schnorr signature scheme will be used where anonymisation is not required.

Chapter 8

Peer-to-peer access controlled RDF triple-store

Where state-of-the-art (SOTA) peer-to-peer solutions have the need to develop a secure protocol and data model on a per-feature basis — with the notable exception of RetroShare — section 3.2, our aim is to build generic *abstract* building blocks for peer-to-peer online social network (OSN).

Glycos tries to solve this problem by storing its data in a Resource Description Framework (RDF) based triple-store (Lassila & Swick, 1997; Lanthaler, Cyganiak, & Wood, 2014). RDF is based on a graph database model. For an in depth introduction on graph databases, refer to e.g. Angles and Gutierrez, 2008; a basic overview is given in chapter 5.

Storage of those triples takes place in a public DHT-like structure, based on Kademlia (cfr. section 2.3.3, and Maymounkov and Mazieres, 2002), so we need to take care to keep data and metadata confidential.

8.1 Rationale

As an example, we will consider the case of Bob making a birthday post on Alice's *wall*. Alice's wall is a virtual place where Alice's friends can share information with both Alice and her friends; i.e. a place where all Alice's loved ones come together.

```
<#Alice>
  g:hasWall <#AliceWall>.
<#AliceWall>
  g:hasPost <#BobsPost>.
<#BobsPost>
  a g:WallPost;
  rdfs:comment "Happy birthday, Alice!".
```

This requires several non-trivial confidentiality and privacy considerations.

1. Alice does not want Charlie, who's not a friend of Alice, to be able to post on Alice's wall;
2. Alice nor Bob want Charlie to be able to read Alice's wall, since her wall is a place for Alice's friends;

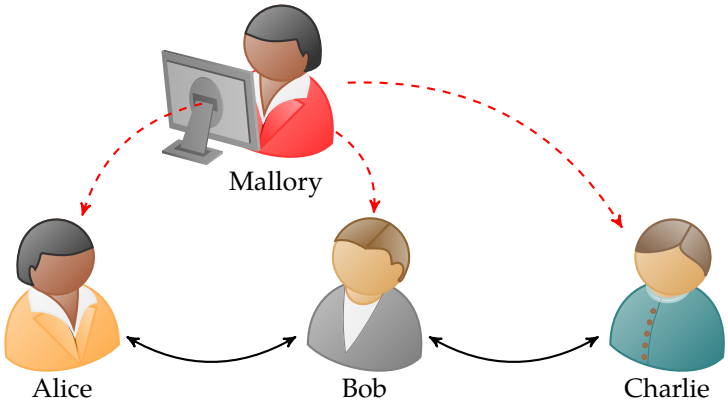


Figure 8.1 – In what follows, Alice and Bob are friends. Mallory is not a friend of neither Alice nor Bob; she tries to intercept and alter communication between Alice and Bob. We take the Alice's viewpoint: Charlie may be a friend of Bob, but Alice does not necessarily want to share with him.

3. Mallory, as a complete outsider, must not be able to learn anything about the network; This includes any content *or even the structure* of the graph;
4. Both Alice and Bob should be able to remove Bob's post from Alice's wall. This makes Alice a kind of "administrator", while Bob is subordinate to the wall;
5. Alice should be able to grant and retract permissions for people on her wall.

8.2 Data model

Recall that a graph database contains *vertices* connected by *edges*. We split the concept of vertices and edges into two distinct objects on the Kademlia DHT. Vertices are identified by their owner (and put on the DHT using a SHA 3 hash).

Definition 8.2.1 (vertex). *A vertex V is a 7-tuple $(O, R, ACL, R_{ACL}, v, c, S)$. O is an ephemeral, unique public key derived from the private key held by the owner of this vertex, the* owner key. *R is the recogniser used to generate O. ACL is the* access control list, *listing all ephemeral public keys that are allowed to link other vertices from this vertex using edges. R_{ACL} is the recogniser used to generate the ephemeral public keys in ACL. v is the optional, associated value. c, the* clock, *is a positive integer. S is a* signature *of (R, ACL, R_{ACL}, v, c) generated by O.*

When Alice creates a vertex V, and wants to grant her friend Bob the right to append other vertices to it, she adds an ephemeral key derived from Bobs key to ACL. This way, Bob can recognise (using R) that he can append vertices, while Mallory cannot.

As signature S, an elliptic curve Schnorr signature (Schnorr, 1991a) is used, as defined in section 7.3. This digital signature certifies the integrity of the message; Mallory — as a woman-in-the-middle — cannot alter the contents R, ACL, R_{ACL}, v, c without forging the signature. Note that altering O does not make sense, since it would only allow Mallory to make invalid copies of the vertex, since the combination O, R is tied to the Alice's public key. Secondly, O will not be in the final serialisation of V, since our signature scheme recovers the associated public key.

By only having the owner public key as part of the Kademlia identifier, we allow the contents of the vertex to be updated. The location on the Kademlia network is thus fixed for a specific vertex. The location is also random (in the random oracle model), which thwarts some poisoning

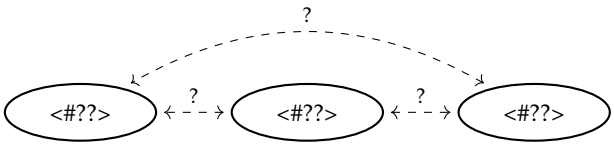


Figure 8.2 – By expanding the ring signature over multiple vertices, an outsider does not learn the connections between vertices. For an outsider, connections between vertices are equiprobable over a set, while the target vertex is fully hidden, to counter statistical attacks.

attacks (Locher, Mysicka, Schmid, & Wattenhofer, 2010): an adversary would need to find a near-second-preimage for O.

We define a *total order* on vertices using the clock c; this way, older versions of vertices can be distinguished from newer updates, and older versions can be dropped.

Definition 8.2.2. *For two vertices $V_1 = (O, ACL_1, v_1, c_1, S_1), V_2 = (O, ACL_2, v_2, c_2, S_2)$, V_2 is said to be* newer *or* more recent *than V_1 (note that the owner keys are equal) if $c_2 > c_1$.*

The k-closest nodes (section 2.3.3) to O can now distinguish which version of both to store, and which to prune from their memory.

It only rests to connect a vertex to another via an edge. We have to take care about two things: we want the user drawing the edge to rest pseudonymous among the ACL, and to counter statistical analysis, we do not want Mallory to be able to count edges departing from, or arriving at a vertex[1].

Definition 8.2.3 (edge)**.** *An edge E between two vertices $V_s = (O_s, ACL_s, v_s, c_s, S_s)$ (the subject) and $V_o = (O_o, ACL_o, v_o, c_o, S_o)$ (the object) is a 5-tuple*

$$E = (O_s, ACL_E, \mathcal{E}_k(\ell, O_o), \mathcal{R}_s, S).$$

$\mathcal{R}_s = \{O_1, O_2, \ldots, O_i = O_s, \ldots, O_n\}$ is a set of public keys, called the ring*, containing all public keys in ACL_s*

n is the ring-size. ACL_E contains the encrypted symmetric key k for the symmetric cipher $\mathcal{E}$, encrypted with each public key in ALC_s.

ℓ is the label of the edge*.*

S is a ring signature (Abe et al., 2002) over the ring $\mathcal{R}_s$ of $(ACL_E, \mathcal{E}_k(O_s), \mathcal{E}_k(\ell), \mathcal{E}_k(O_s))$

Encrypting (ℓ, O_o) hides the contents, and where the edge arrives from Mallory, while all users in the access control list can decode the edge.

By using a ring signature, the actual creator of the edge can stay anonymous. Recall that the ACL_s contains pseudonymous (ephemeral) public keys. These keys are not linkable to the owners of their respective private keys; however, two edges with the same creator would be linkable under a regular signature scheme.

8.2.1 Deterring statistical analysis

We can go further than this; by extending the ring signature over *several* source vertices, the ring signature would prevent Mallory to infer the actual subject vertex V_s. It prevents her from

[1]When Mallory can count those edges, she might be able to infer which vertices represent what kind of objects. In conjunction with traffic analysis, this may thwart the sought after confidentiality properties.

counting the edges that leave a specific vertex V_s, while it still allows for anyone to infer the location of the edge E on the DHT (cfr. fig. 8.2), which is a required property to retrieve the edge by a legitimate user.

While this may practically prevent Mallory from using statistical attacks, this also poses some challenges. Assume that Alice owns a vertex V_a. Mallory can now carry out a denial of service attack, which will potentially floods Alice and her peers (Bob) when retrieving V_a and its associated edges. Mallory can construct a vertex V_m, and then create many valid edges from V_m. By taking the ACL of V_a in the ring signature, Alice and Bob will be required to filter out these edges when retrieving V_a, potentially creating a significant load on Alice's and Bob's device.

A second problem arises when Alice wants to delete V_a; edges that contain V_a in their ring will require an update which Alice cannot carry out. This might be less of a problem, if we would just allow the edge to contain invalid vertices in the ring.

To overcome the denial of service, one could require the ring of vertices to be somehow deterministic; it is however unclear at this point *which* vertices to pick. A simple deterministic set of elements in this ring would be the k-closest vertices. These are easily located, but they *may change over time*. This would require a notion of causality ("happened before" relations) in the graph (Lamport, 1978).

Moreover, it may be possible to infer the real vertex V_a from this set, as the ring would somehow be centred around V_a.

Note that carrying out this statistical analysis requires access to a significant portion of the graph, since Alice's data is "spread" over the whole Kademlia DHT. This process would thus only be possible when running a significant portion of nodes, *and* eavesdropping on communication lines.

In any case, we think this method of anonymising the graph demands further research, and a deep analysis in the anonymising properties of any proposed solution. Note that this anonymisation can be carried out independently from the application level, since the data model loosely couples the application with the storage, i.e. the application can be developed independently of the back-end storage model.

8.3 Operations on the graph

Ideally, we want to support CRUD operations; i.e., *create*, *read*, *update*, and *delete*. These are the standard operations supported by any relational database. In this context however, we also have to take into account access control rules. We will thus see that users can carry out those CRUD operations only when authorised to do so.

8.3.1 Create

There are two ways to look at the create operation. Alice may want to insert a *triple* (s, p, o). What operations she has to carry out depends on the existence of the subject vertex; if it exists, she only has to post an edge, while if it does not exist she has to post both an edge and a vertex.

The other way to look at this operation makes more sense w.r.t. the data structure on the Kademlia DHT. In this view, Alice can create *vertices* or *edges*; i.e. there are two *create* operations.

Example 8.3.1 (Alice creates her wall)**.** *Alice is a new Glycos user, and she wants to create a profile with a* wall*, and grant Bob the rights to post there.*

She creates a new vertex, by deriving a random ephemeral public key (the *owner* key O) from her user key. She adds Bob to the ACL, by deriving another random ephemeral public key from his user key. She announces the vertex by transmitting it to the k-closest nodes to this vertex.

Example 8.3.2 (Bob posts on Alice's wall)**.** *Bob posts a birthday message on Alice's wall.*

He thus wants to write a triple `<#AlicesWall> g:wallPost <#BobsPost>`, as in fig. 6.1. Of course, the vertex V_a `<#AlicesWall>` already existed, and is owned by Alice. However, Bob is listed in the ACL. He first creates his post as a new vertex V_b — potentially with other attributes such as time, date, text, and others — and then publishes a new edge. He does this by announcing the edge to the k-closest nodes to V_a.

8.3.2 Read

For Bob to read a vertex, he has to know the *hash* of the owner key. This has may have been exchanged *a priory* between Alice and Bob (when they first talked with each other about Glycos), or discovered while reading other vertices and following edges.

Example 8.3.3 (Bob reads Alice's wall)**.** *Bob wants to read posts on Alice's wall.*

Bob asks the k-closest nodes to V_a to search for V_a. Bob thereby receives the vertex V_a, and all associated edges. Bob recursively requests vertices through these edges, thereby receiving all posts on Alice's wall that he is allowed to read.

Bob can read those edges that were meant for him — after all, his (ephemeral) public key was listed in the ACL of V_a, and all edges are required to encrypt for that key — and can thus traverse the three.

8.3.3 Update

Updating is a more complex subject; Alice may want to update her wall V_a to accommodate more friends, or remove friends from the ACL. Bob may want to update his wall post, because he made a typo.

In both cases, the owner can compute the private key associated with the owner key O, and create a new vertex with an updated clock value $c' = c + 1$. The owner can sign this vertex with his private key.

When the ACL does not require an update, this new vertex can be published as-is. When the ACL however *does* require an update, all edges associated with V_a needs an update.

Let us first consider altering the ACL. There are two operations: a new member can be added, and a member can be removed. While strictly unnecessary[2], we we completely reconstruct the ACL even on deletion, to avoid that Mallory observes the key that was deleted. This means that the recogniser R_{ACL} will be replaced, and that all keys in associated edges will be invalidated. Note that the owner of a vertex is always considered included in the ACL.

Example 8.3.4 (Alice has become friends with Charlie)**.** *Bob has introduced Alice to Charlie, and she wants to grant Charlie right to her wall.*

[2]Note that removing a key from ACL does not require knowledge of r, the discrete logarithm of $R = rG$, while adding a key $K = kG$ requires knowledge of r or k since the ACL entry equals $\mathcal{H}_s(rK)G + K = \mathcal{H}_s(kR)G + K$. Thus, adding a key to the ACL requires *or* reconstruction of the ACL, or storing the discrete logarithm r.

Alice has received Charlie's master key (for example through Bob). She also remembers the other master keys that were previously put in ACL. She regenerates the ACL, and transmits it to the k-closest nodes.

Alice regenerates all edges starting from V_a; she can sign all those edges, since she is considered part of the ACL.

Note that using the above procedure, deletion of a key from the ACL does not delete the actual data.

Example 8.3.5 (Alice revokes Charlies rights)**.** *Alice does not want to be friends with Charlie any more, she wants to remove him from the ACL of her wall.*

Obviously, if Alice uses the same procedure as above, any potential posts by Charlie on Alice's wall will still exist. Note though that she can choose not to regenerate Charlie's edges, which effectively removes his posts from the wall.

8.3.4 Delete

Delete, just like create, can be seen from different perspectives. There are two ways in which Bob's birthday post on Alice wall can be removed.

Example 8.3.6 (Alice deletes Bobs birthday post)**.** *Alice may not like the post, and should be able to delete it from her space.*

We introduce an "anti-edge" to remove this post.

Definition 8.3.7 (Anti-edge)**.** *An anti-edge for the edge $E = (O, ACL_E, C, \mathcal{R}_s, S)$ is the 3-tuple (O, S, S'). S' is a signature of S signed by O. The anti-edge (O, S, S') is said to annihilate E.*

Alice sends out an *anti-edge* for the edge to V_b to the k-closest nodes to V_a. This effectively unlinks the data. If Bob notices the removal of the link, he should remove the vertex, as a clean-up step.

The k-closest nodes to V_a are required to store this anti-edge, as proof for the deletion of this edge, may Mallory ever try to restore it[3] through an replay attack.

Another option for Alice is to regenerate the ACL for V_a, and while regenerating the edges, not to include the edge to V_b.

Definition 8.3.8 (Anti-vertex)**.** *An anti-vertex for $V = (O, R, ACL, R_{ACL}, v, c, S)$ is the 3-tuple (O, v, S'). S' is a Schnorr signature of v signed by O. The anti-vertex (O, v, S') is said to annihilate V.*

Example 8.3.9 (Bob deletes his own post on Alice's wall)**.** *Bob may not like his post, and may want to remove it from Alice's wall.*

Bob sends out an *anti-vertex* for V_b to the k-closest nodes. This effectively removes the data for this post, not the link. When Alice notices the removal of the post, she should remove the edge.

To counter the same kind of replay attack, the k-closest nodes to V_b shall store this anti-vertex.

Storing anti-vertices and anti-edges may accumulate. It is thus important to represent them as compact as possible.

[3]Mallory can just retransmit a copy of the edge as "new", without needing Alice's or Bob's intervention, effectively reinstantiating the edge.

8.4 Optimizing the data model

When Bobs vertex has many read-only attributes, e.g. a wall post with time, date, author, and a title, this may generate a lot of recursive requests on the graph.

When it does not make sense to grant second parties write access to those attributes, it might be worthwhile to encapsulate part of the graph in the vertex itself, in our example, embed all those attributes into V_b. This will enhance responsiveness, decrease network usage, and require less cryptographic operations.

In the future, it should be possible to remove these from storage, reclaiming the space. One possibility would be — when Glycos somehow gains a notion of "happened before" such as in section 8.2.1 — using a notion of causality to deter the mentioned replay attacks. This too requires additional research.

Another possibility is to refrain from deletion at all, and only updating vertices and their ACL. In the Bob deletion scenario, he can empty the ACL of V_b, and remove the contents v of V_b, effectively rendering his post deleted. This will however be more difficult to garbage collect when the above replay attacks are mitigated.

Using the example of a wall, we constructed a data model wherein Alice can delegate read/write access to her peers. Those peers can post to her wall, read messages, and delete their own. Alice can remove messages by others from her wall, as sort of an administrative privilege. We demonstrate how the create-read-update-delete operations (CRUD) operations can be implemented, effectively mimicking properties of classical relational databases.

Moreover, we demonstrate how the database model can be further anonymised by deterring statistical analysis when a significant portion of the database would be collected. This technique however requires further research[4], since it is not clear how updating or deletion would work. The exact anonymising properties of this technique also require deeper future analysis.

Now that we designed the actual data model, the next chapter is devoted to a proof-of-concept implementation, and discussion of benchmarks ran on that implementation.

[4]It is also unclear whether statistical analysis would actually yield deanonymisation, or any other significant data or metadata.

Chapter 9

Implementation

A proof of concept implementation was written in Rust[1]. The main reason to pick Rust over another language was the cross-platform nature: Rust code — just like C++ — compiles to fast machine code, its powerful static analysis, and compilers exist for all major (and minor) mobile and desktop platforms. Bindings to the target platforms can be implemented trivially, since the Rust compiler optionally implements the native C calling convention for all target platforms.

Two major components had to be implemented: the networking part, and the graph and cryptography part. Figure 9.1 visualises of how these work together.

The rest of this chapter is dedicated to the design choices and challenges in the host code, the tests that were ran, and some benchmarks.

9.1 Cryptography

A few cryptographic building blocks were not available off-the-shelve. First, there is the ephemeral private key derivation (section 7.2), inspired on the Monero (section 4.2) primitives. Second, there are the Abe et al., 2002 ring signatures, and the generic Schnorr, 1991a signatures had to be implemented. The popular Ed25519 signatures (Bernstein, Duif, Lange, Schwabe, & Yang, 2012) were available off-the-shelve, but as explained in section 7.3, they could not be used in our context.

[1] "Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety." https://www.rust-lang.org/

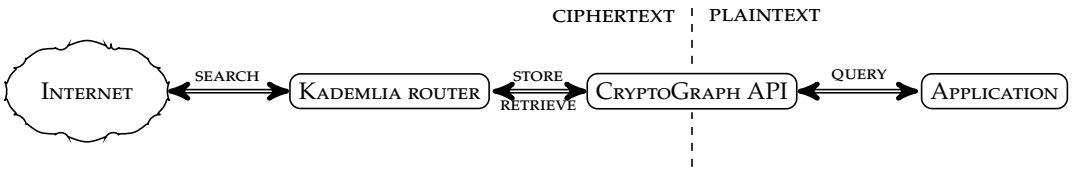


Figure 9.1 – The two main components to be implemented are the application programming interface (API) for the cryptographic graph, and the custom Kademlia router. The Kademlia router is responsible for connection with other peers, storing and retrieving encrypted graph data on the network. The graph API implements the encryption and decryption of the graph, feeding it to the application.

Third, existing cryptographic primitives (hashes, Curve25519, authenticated encryption with associated data (AEAD), key expansion, and message authentication codes) had to be sourced from existing libraries.

9.1.1 Disinheriting Monero

Implementing the ephemeral secret key algorithm from section 7.2 and both signatures schemes from sections 7.3 and 7.4 was at first done by adding the Monero source code as a Git submodule[2] of the project. Monero's Curve25519 implementation is a slight update of the original Bernstein, 2006 reference implementation. A few C methods implemented the small differences between Monero's protocol and ours, and glues itself to the Rust code base[3]. C code contributed just over 100 lines of source.

This approach based on the Monero source code worked, but was hard to debug and improve. The Monero source code was only used for Curve25519 operations and the Diffie-Hellman key exchange, and not for ring signatures or ephemeral key derivation, since Glycos' primitives differ slightly (sections 7.2 and 7.4). This coincided with discovering the high quality `curve25519-dalek` Rust library (de Valence & Lovecruft, 2016–2018), which provides all necessary operations on this elliptic curve in a safe and secure API. Porting the primitives proved easy enough; the source code line count actually shrank by 230 lines, since the C based glue code was not required any more.

9.1.2 Signatures

`curve25519-dalek` made implementing the Abe et al., 2002, and Schnorr, 1991a signature algorithms quite simple. The type safe design of Rust allows for example to use a "builder pattern" to incrementally construct a ring signature. A snippet from the unit test code as example:

```
132 let mut signer = AbeRingSigner::default();
133 signer.push(&alice_pk, &rng).unwrap();
134 signer.push(&carol_pk, &rng).unwrap();
135 let signature = signer.finish(&bob, &rng, data).unwrap();
136
137 assert!(signature.verify(data));
```

This example constructs a ring signature with three keys. Bob is the actual signer of the `data`, Alice and Carol are in the ring as the other members.

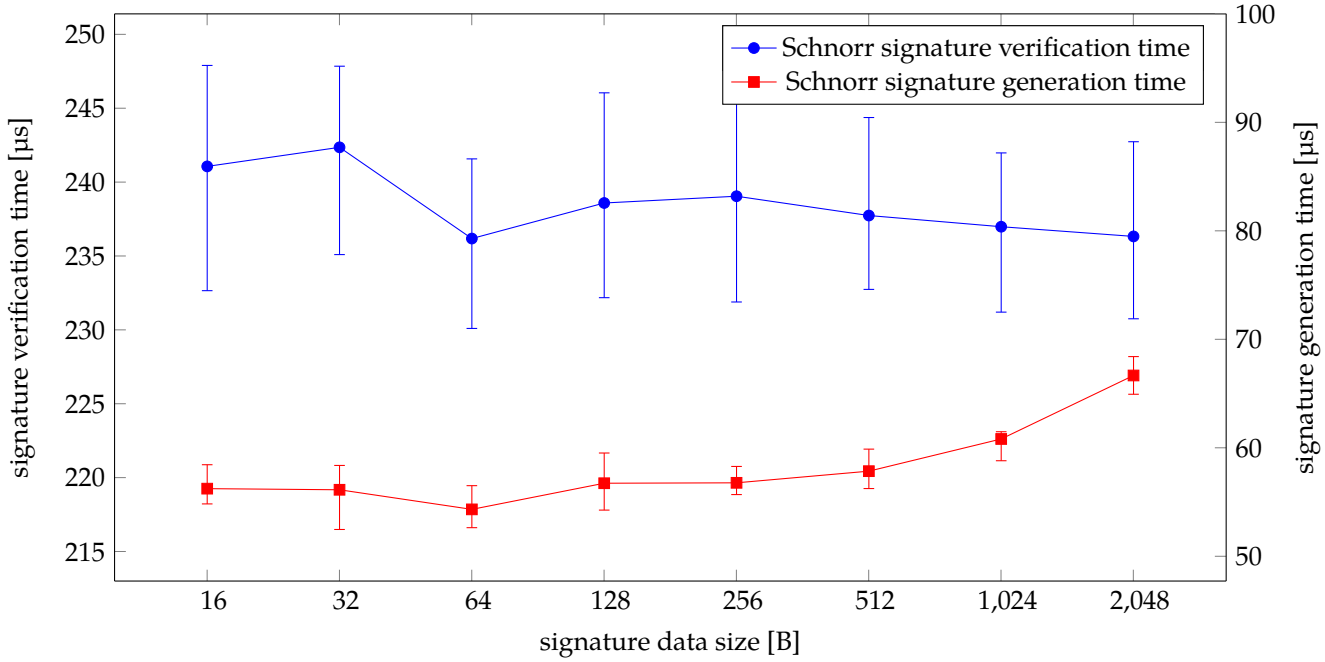
Since there is not a lot to "build" to a regular signature, that operation is implemented in a more direct way:

```
82 let data = b"Hello world";
83 let signature = sk.sign(&rng, data).unwrap();
84
85 assert!(pk.verify(&signature, data));
```

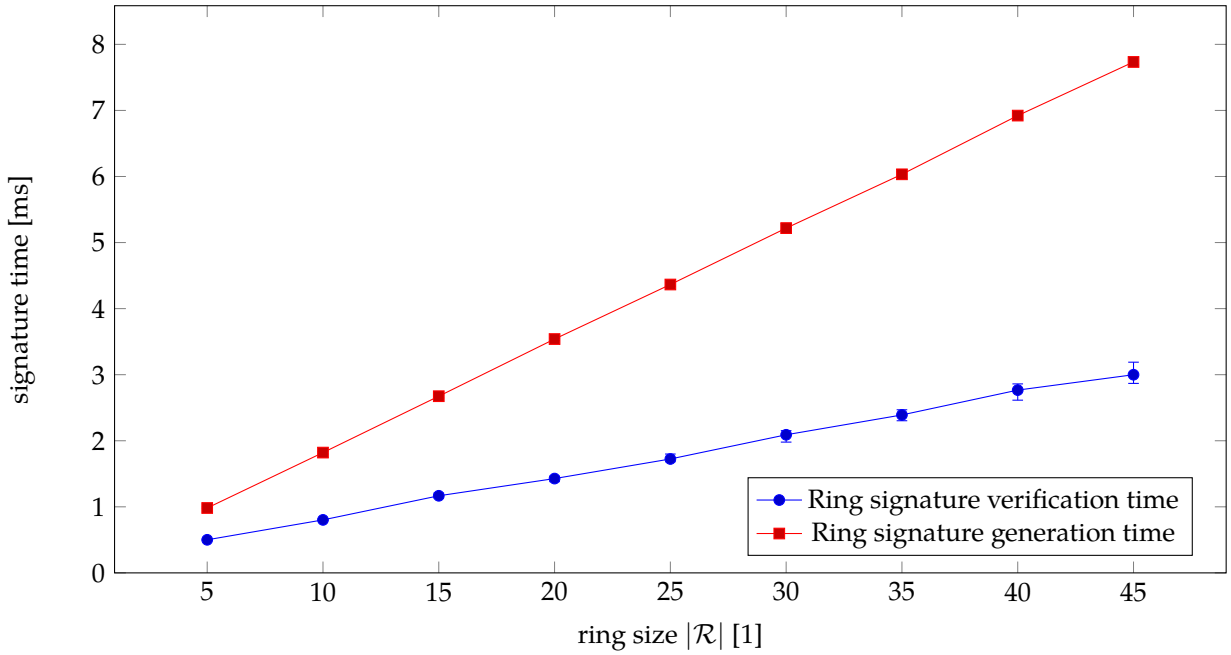
The implementation of these signatures was benchmarked on a laptop. All relevant specifications can be found in table 9.1. The smartphone is consistently around a factor 10 slower than

[2] From `man gitsubmodules`: "A submodule is a repository embedded inside another repository. The submodule has its own history; the repository it is embedded in is called a superproject."

[3] Mixing Rust and C is possible since Rust can link to external C libraries



(a) Signature verification is slower than signature generation. For the small data sizes used here, the (input independent) elliptic curve operations take significantly more time than the (input dependent) underlying hash function.



(b) Signature generation is slower than ring signature verification. These signatures scale with ring size $\mathcal{R}$, which is apparent from the plot.

Figure 9.2 – Both signature schemes run in millisecond to sub-millisecond timings. This means that several hundreds to thousands of signatures can be generated per second, which is a necessary condition for efficient implementation on mobile devices, where battery and processing power are scarce. Error bars signify 95 % confidence interval on the displayed median.

Table 9.1 – Specifications of the devices used for benchmarks. All benchmarks were ran on the laptop, except where otherwise noted.

	Laptop	**Smartphone**
Brand	Lenovo Thinkpad X250	Lenovo Moto Z Play
CPU	Intel Core i5-5200U (Broadwell)	ARM Cortex A53 (MSM8953)
Core count	2 cores, 4 threads	8 cores
Clock frequency	2.20 GHz	2.0 GHz
RAM	16 GB DDR 3 at 1600 MT/s	3 GB DDR 3
Operating system	Arch Linux	SailfishOS 2.1.3.7 armv7hl
Rust compiler	1.28.0-nightly (`b907d9665` 2018-06-13)	

the Intel CPU. Both the performance on laptop and mobile should increase when `curve25519-dalek` fully implements vector instructions[4]. Data for these benchmarks was gathered using the `criterion` library (Aparicio & Heisler, 2014–2018).

9.1.3 Off-the-shelve cryptography

All other used primitives are off-the-shelve components. As such, the `ring` library by Smith (Smith, 2014–2018) is used where AEAD ciphers (Bellare & Namprempre, 2000), Hash based message authentication code (MAC) (HMAC)[5], or cryptographic hashing algorithms are needed. For SHA 3 (Dworkin, 2015) support, an external library (aptly called `sha3`) is used, since `ring` has not yet implemented Keccak (Bertoni et al., 2009), and as such SHA 3 is still lacking at the time of writing[6].

9.2 Non blocking computations

Glycos performs many operations on the network. Those operations are prone to latency, and may block the running thread.

Instead of *waiting* for a computation to finish, Glycos employs *futures* (often referred to as promises), which are proxy objects representing an initially unknown value (Friedman & Wise, 1976). A variation of a future is a *stream*, which represents a conceptually unlimited amount of return values of the called computation.

Futures are used on all levels in Glycos' design, from the network (UDP stack) to the high-level graph traversal procedures. This is possible since futures can be chained: it is possible to — instead of awaiting the result — register further processing functions that will be called when the future is resolved.

Rusts support for futures is profound[7], and is being considered for (partial) adoption in the standard libraries[8]. What follows will describe bottom-up where futures reside and what data

[4]The implementation of vector instructions is still on its way as of June 16 2018, cfr. https://github.com/dalek-cryptography/curve25519-dalek/issues/142

[5]We use HMAC-SHA512/256 instead of the more modern Keccak MAC (KMAC) (Kelsey, 2016), because of the current availability in `ring`. When `ring` implements KMAC, this will be changed.

[6]ring's Keccak tracking issue: https://github.com/briansmith/ring/issues/59

[7]For an overview of all features of the `futures` library, cfr. https://docs.rs/futures

[8]As of 16 June 2018, the `futures` library is in the *Rust language nursery*, cfr. https://github.com/rust-lang-nursery/futures-rs/, and an additional syntax for futures has been accepted for inclusion in the Rust compiler, cfr. https://github.com/rust-lang/rfcs/blob/master/text/2394-async_await.md.

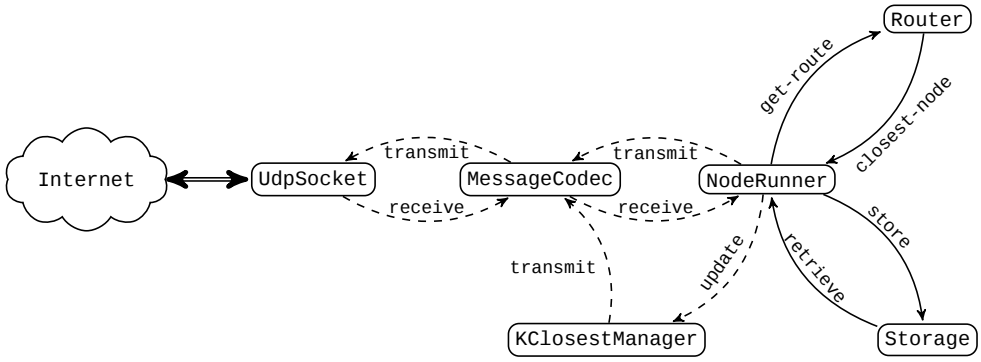


Figure 9.3 – Communication diagram of the different components of the network stack. The `UdpSocket` is the operating system API to connect to the internet. It receives datagrams through Tokio, which parses the raw bytes through the `MessageCodec`. The `NodeRunner` routes the messages to the different components. The `Router` keeps track of the Kademlia buckets, the `KClosestManager` keeps track of *k*-closest searches, and finally the `Storage` stores data (vertices and edges) in memory or on disk. Dashed arrows depict *asynchronous* communication using *futures* or *streams*.

they represent.

9.2.1 Network stack

At the lowest level, futures are used on UDP sockets. Tokio (Crichton, Lerche, Roman, & et al., 2016–2018) provides the abstractions hereto. It provides a UDP socket which represents a `Stream` of UDP datagrams. These datagrams get parsed by the `MessageCodec` (which also validates the Kademlia ID, as shortly described in section 2.4), and are in turned routed through the application via the `NodeRunner`, as depicted in fig. 9.3.

Part of the communication in the network diagram is synchronous, i.e. the calls to the `Router` and to the `Storage`. These calls are low latency calls, since the router is completely in-RAM, and storage is heavily cached on both read and write.

Things become interesting when looking at the `KClosestManager`. This component implements Kademlias *k*-closest search, and the front-end is also implemented using futures. This means that *k*-closest search is implemented completely asynchronously from the socket on, without spawning additional threads, using blocking I/O, or using locks. When the `KClosestManager` receives an update from the `NodeRunner`, it will update the state of the associated search request, optionally fulfilling the future when ready.

The search state has an associated timeout on every request, such that every *k*-closest search eventually returns.

9.2.2 Notes on asynchronous programming

A completely asynchronous design enables the programme to sleep, only waking up when the operating system has an update (usually a socket or a timer). It makes locking in general unnecessary, avoiding deadlocks altogether.

Asynchronous programming has not been fully accepted by everyone however. One well known counter argument is posed in the "What Color is Your Function?" blog post by Nystrom.

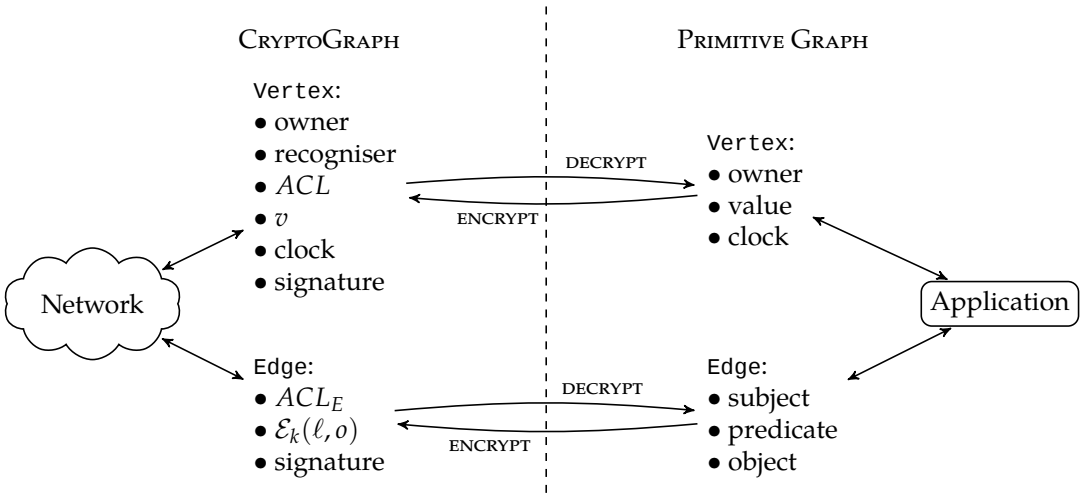


Figure 9.4 – The graph API. Clear text operations are clearly separated from cipher text domain operations, and conversion between the two domains happens through an explicit encrypt or decrypt method.

He justly writes that a non-async program cannot easily invoke asynchronous programs, since the latter require the setup of an event loop (Nystrom, 2015).

9.3 Graph APIs

The graph API is designed to be easy to use, and difficult to misuse. Retrieved edges and vertices have a different type (`cryptograph::{Edge, Vertex}`) than decrypted edges and vertices (`primitives::{Edge, Vertex}`). Since Rust is a strongly typed language, conversion between these types is only possible through the relevant methods. Those `decrypt` and `encrypt` methods take the required keys as argument, and return an error or the decrypted/encrypted vertex or edge. This follows the same principle as a red-black architecture (where `cryptograph::` is "black" and `primitives::` is "red") from the U.S. military (MIL-HDBK-232A, 1987). This process is graphically represented in fig. 9.4. The network API only accepts encrypted vertices and edges to be transmitted.

9.4 Testing

With a total line count of 2979 spread over 29 source files, rigorous testing is required.

Besides the around 25 unit tests, integration tests called "torture test" starts 128 (local) nodes to test various things. The first torture test starts 128 nodes, and checks whether the routing table gets populated with at least $k = 4$ nodes.

The second torture test starts 128 nodes, and stores and re-retrieves a vertex and an edge from the network. This effectively tests whether the `KClosestManagers` asynchronous APIs function correctly.

Since both torture tests run in parallel (which is the default in Rusts testing framework), 256 nodes run concurrently. An approximate 2 GB of RAM at peak is consumed during these tests, which is around 8 MB per node.

9.5 Binaries

Beside unit and integration tests, two other binaries were built too. One of these is a UNIX *daemon* called `glycosd`, which is meant to be deployed on a long running node. Long running nodes can be used by short running nodes as initial contact point.

The `glycosd` daemon is also available as a Docker image, for easy nightly deployment.

A second *interactive* binary is the `glycos-explore` tool. This command-line tool provides an interactive prompt, and provides a set of commands to interact with the graph database.

An example session could go as follows, starting from a new or empty account:

```
Glycos explore <#Vertex 1172c29c> % ls
Glycos explore <#Vertex 1172c29c> % append is foaf:Person
Glycos explore <#Vertex 1172c29c> % ls
 -> is foaf:Person
Glycos explore <#Vertex 1172c29c> % append foaf:lastName Smith
Glycos explore <#Vertex 1172c29c> % append foaf:givenName John
Glycos explore <#Vertex 1172c29c> % ls
 -> is foaf:Person
 -> foaf:lastName Smith
 -> foaf:givenName John
```

In this session, John creates his account. The first time `glycos-explore` starts, the application generates a *master key*[9]. The first four bytes of the SHA 3 hash of his master public key are shown in the explorer console as a hexadecimal string. He can then *list* the outbound edges using the `ls` command, and *append* a new one using `append`. He thus creates a profile with his first and last name.

The presented implementation is far from complete, however. Update and delete operations are not implemented, although they are easy enough to implement. Further more, local storage is not pruned or resynchronised when closer nodes are discovered. An even more high-level ORM-like interface should also be built, which can for example compile a database model in a domain specific language (DSL) into Rust structures, and integrate with other target languages (Java or Kotlin on Googles Android, Swift or Objective C on Apples iOS). These issues are implementation-level details however; the given implementation is aimed to be an extensive proof-of-concept.

[9] Subsequent starts of the application load the key from disk.

Epilogue

Glycos is meant to be an extensible, resilient and private peer-to-peer online social network (OSN). We'll let this title guide us through these concluding words.

The main two goals were to provide *privacy* and *extensibility*. These two properties took a large part of the work. Chapters 1, 2, 4 and 7 were all devoted to making Glycos a private system. The privacy of Glycos stood on two main pillars: *decentralisation* and *cryptography*. Through decentralisation, we can enforce that no single central point of control can hold the data. Fast cryptography further anonymises and encrypts user data.

On the *extensibility* side of Glycos, the data model is inspired on a graph database. Chapters 5 and 8 explained that just like in regular Web 2.0 applications, this enables the application developer to focus on the application; it puts designing secure peer-to-peer applications on the same level as classical application development. Moreover, the data model is decoupled from the interface, which means that the underlying model can be updated and replaced without heavily impacting application development. This was our main contribution.

Glycos is a *resilient* and self-healing system, fulfilled through the use of the *peer-to-peer* Kademlia DHT. Chapters 2 and 3 explained how Kademlia offers routing logarithmic in the network node count, also contributing to an efficient design.

Maybe not in the title, but equally important would be *efficiency* or performance. Efficient applications drain less battery, are more responsive to users. While the benchmarks of the proof-of-concept implementation indicate a performant base application, there is still room left for improvement, e.g. by using vectorised instructions.

While the work is not over, Glycos seems to be a solid and efficient base on which a potential new OSN can be built. And just maybe, it can bring the world a little more online privacy.

Future work

Throughout the text we already stumbled upon interesting future research questions. What follows summarises these, and points out additional future research questions.

Testing anonymity

We did make some claims with respect to anonymity. Claiming the use of ring signatures over an access control list of users enhances anonymity at least seems like a mathematical consequence. However, when expanding the ring over several vertices as proposed in section 8.2.1 this consequence becomes less clear, especially when this ring of vertices becomes a deterministic output (to counter denial-of-service, cfr. section 8.2.1). Further research is required to look at the consequences of this proposal.

Enhancing the data model

There are many possible ways in which we can possibly enhance the data model. One thing that comes to mind is the large data structure of the edge representation (section 8.2). It should be possible to reduce the size of the ring signature and the edge significantly, since the ring members are already stored elsewhere.

The update operation (section 8.3.3) is currently not atomic, i.e. Bob can append an edge to V_a while Alice is updating V_a, which will invalidate Bobs addition. By clever protocol design or by redesigning the model using commutative operations, it should be possible to make updates atomic.

Post-quantum cryptography

Our constructions with asymmetric cryptography (ring signatures, ephemeral public keys, Schnorr signatures, chapter 4) relies heavily on elliptic curve cryptography. These primitives assume that the discrete logarithm problem (DLP) is difficult, i.e. that computing the discrete logarithm takes an exponential time. It is postulated however that a sufficiently large quantum computer can compute the discrete logarithm in polynomial time (Shor, 1999).

It might be interesting to research whether it is possible to construct this database model in a *post-quantum* safe way, i.e. by substituting DLP-based cryptography with primitives that are not broken using a quantum computer.

Privacy as practice

Recall the three privacy engineering research paradigms by Diaz and Gürses (chapter 1): (1) privacy as *control*, where users can decide on their privacy on a social level; (2) privacy as *confidentiality*, where users keep private data confidential by technical means; (3) privacy as *practice*, focussing on how to give the user sufficient feedback and control.

The design of Glycos focussed mostly on privacy as *confidentiality*. When actual end-user applications are developed, research will shift more towards privacy as *control*. The design of the user interface, the ontological requirements for the data model, the actual controls given to the end-user fall in that paradigm.

When a user interface is built, and in the (hopeful) case of having some active users, Glycos might be used as a research tool for privacy as *practice*.

Glossary

AE: authenticated encryption A symmetric cipher together with a message authentication code (MAC). 55

AEAD: authenticated encryption with associated data A symmetric encryption cipher that includes authentication (authenticated encryption (AE)), and optional unencrypted associated data. 44, 46

API application programming interface. iii, 13, 43, 44

client-server a network structure where users contact a central authoritative system, often a single server or a data centre. 56

CRUD: create-read-update-delete operations These are considered the four basic operation on relational databases. 41

DHT: distributed hash table a peer-to-peer key-value store based on a hash function. 8–10, 16, 17, 35

DLP: discrete logarithm problem the problem of computing a in g^a over a finite field. 53

DSL domain specific language. 49

ER: entity-relationship A database abstraction where data is modelled using "relations" between data. 25, 56

f2f: friend-to-friend a peer-to-peer (p2p) like system where peers only contact social relations. 9

FOAF Friend Of A Friend, an Resource Description Framework (RDF) ontology describing persons and their relation with each other and the world. 26

forum a place for people to leave messages; also called a discussion board. 16

GXS: generic data exchange system A system developed for RetroShare where a certain level of abstraction is given to the RetroShare developers (Soler, 2017). 16, 17

HMAC: Hash based MAC A MAC based on potentially any hash. A HMAC calls its associated hash function twice to counter length-extension attacks. Keccak MAC (KMAC) does not suffer from this inefficiency. 46

KMAC: Keccak MAC A message authentication code based on Keccak (Bertoni et al., 2009). 46, 55

MAC: message authentication code A short piece of information that is used the authenticity and integrity of a message. 46, 55, 56

metadata data describing the data; for example, the message "Hello Bob" has as metadata 'sent by Alice', 'destined to Bob' and 'sent on 17 June 2018'. iii, 41

NIZKP non-interactive zero knowledge proof. 23

ORM: object relational mapping A programming abstraction where database relations are mapped onto objects or structures in an object oriented programming language. iii, 25

OSN: online social network an online service where users make social connections and interactions. 6, 7, 11, 13–17, 26, 29, 35, 51

p2p: peer-to-peer a network structure where nodes contact other nodes. Often contrasted with client-server. 10, 29, 55

PET privacy enhancing technologies. v

RDBMS: relational database management system a database system modelled after the entity-relationship (ER) paradigm. 17, 29

RDF: Resource Description Framework A W3C standard for data and metadata modelling in a semantic web context. 25, 26, 35, 55

REST: Representational State Transfer an extension of HTTP for machine readable data transfers. iii

RPC remote procedure call. 17

SOTA state-of-the-art. 29, 35

SPARQL query language for graph databases. 26

SQL structured query language. iii

wall Alice's wall is a virtual space where her connections can post messages. Those messages are usually visible for Alice and her connections. 35, 36, 38–41

Bibliography

Abe, M., Ohkubo, M., & Suzuki, K. (2002). 1-out-of-n signatures from a variety of keys. *Advances in Cryptology—Asiacrypt 2002*, 639–645.

Agre, P. E. (2003). P2P and the Promise of Internet Equality. *Commun. ACM*, *46*(2), 39–42.

Angles, R. & Gutierrez, C. (2008). Survey of graph database models. *ACM Comput. Surv. 40*(1), 1:1–1:39.

Aparicio, J. & Heisler, B. (2014–2018). criterion.rs: Statistics-driven micro-benchmarking library. Retrieved June 17, 2018, from https://github.com/japaric/criterion.rs

Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., & Stoica, I. (2003). Looking up data in P2P systems. *Communications of the ACM*, *46*(2), 43–48.

Bellare, M. & Namprempre, C. (2000). Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Advances in Cryptology—ASIACRYPT 2000*, 531–545.

Bellare, M., Namprempre, C., & Neven, G. (2009). Security proofs for identity-based identification and signature schemes. *Journal of Cryptology*, *22*(1), 1–61.

Bernstein, D. J. (2006). Curve25519: new Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, & T. Malkin (Eds.), *International workshop on public key cryptography* (pp. 207–228). Berlin, Heidelberg: Springer Berlin Heidelberg.

Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., & Yang, B.-Y. (2012). High-speed high-security signatures. *Journal of Cryptographic Engineering*, 1–13.

Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G. (2009). Keccak sponge function family main document. *Submission to NIST (Round 2)*, *3*, 30.

Boneh, D. (1998). The Decision Diffie-Hellman problem. In J. P. Buhler (Ed.), *Algorithmic number theory* (pp. 48–63). Berlin, Heidelberg: Springer Berlin Heidelberg.

Borisov, N., Goldberg, I., & Brewer, E. (2004). Off-the-Record Communication, or, Why Not To Use PGP. In *Proceedings of the 2004 acm workshop on privacy in the electronic society* (pp. 77–84). WPES '04. Washington DC, USA: ACM.

Brickley, D. & Miller, L. (2014). *FOAF vocabulary specification 0.99*. FOAF Project. Retrieved from http://xmlns.com/foaf/spec/20140114.html

Buchegger, S., Schiöberg, D., Vu, L. H., & Datta, A. (2009). PeerSoN: P2P social networking - early experiences and insights. In *Proceedings of the Second ACM Workshop on Social Network Systems Social Network Systems 2009, co-located with Eurosys 2009* (pp. 46–52). Nürnberg, Germany.

Carothers, G. & Prud'hommeaux, E. (2014). *RDF 1.1 turtle*. W3C. Retrieved from http://www.w3.org/TR/2014/REC-turtle-20140225/

Courtois, N. T. & Mercer, R. (2017). Stealth Address and Key Management Techniques in Blockchain Systems. In *Proceedings of the 3rd international conference on information systems security and privacy - volume 1: Icissp* (pp. 559–566). INSTICC. SciTePress.

Crichton, A., Lerche, C., Roman, & et al., A. T. (2016–2018). Tokio: A runtime for writing reliable, asynchronous, and slim applications with the rust programming language. Retrieved June 17, 2018, from https://tokio.rs/

Cukier, K. & Mayer-Schoenberger, V. (2013). The Rise of Big Data: How It's Changing the Way We Think About the World. *Foreign Affairs*, *92*, 28–40.

danah boyd & Hargittai, E. (2010). Facebook privacy settings: Who cares? *First Monday*, *15*(8). Retrieved May 28, 2018, from http://journals.uic.edu/ojs/index.php/fm/article/view/3086

Datanyze. (2018, June 12). Email hosting market share report. Datanyze. Retrieved June 14, 2018, from https://www.datanyze.com/market-share/email-hosting

de Valence, H. & Lovecruft, I. (2016–2018). curve25519-dalek: A pure-Rust implementation of group operations on Ristretto and Curve25519. Retrieved June 17, 2018, from https://github.com/dalek-cryptography/curve25519-dalek

Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., … Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual acm symposium on principles of distributed computing* (pp. 1–12). PODC '87. Vancouver, British Columbia, Canada: ACM.

Diaz, C. & Gürses, S. (2012). Understanding the landscape of privacy technologies. *Extended abstract of invited talk in proceedings of the Information Security Summit*, 58–63.

Diffie, W. & Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, *22*(6), 644–654.

Douceur, J. R. (2002). The Sybil Attack. In P. Druschel, F. Kaashoek, & A. Rowstron (Eds.), *Peer-to-peer systems* (pp. 251–260). Berlin, Heidelberg: Springer Berlin Heidelberg.

Dworkin, M. J. (2015). SHA-3 standard: Permutation-based hash and extendable-output functions. *Federal Inf. Process. Stds.(NIST FIPS)-202*.

Facebook. (n.d.). Facebook's Privacy Principles. Retrieved May 30, 2018, from https://www.facebook.com/about/basics/privacy-principles

Fairchild, J. (2015, December 30). RetroShare – Documentation: Forums. Retrieved May 31, 2018, from https://github.com/RetroShare/documentation/wiki/Documentation:-Forums

Fielding, R. T. & Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation, University of California).

Friedman, D. P. & Wise, D. S. (1976). *The impact of applicative programming on multiprocessing*. Indiana University, Computer Science Department.

Fujisaki, E. & Suzuki, K. (2007). Traceable ring signature. In *Public key cryptography* (Vol. 4450, pp. 181–200). Springer.

Graham-Harrison, E. & Cadwalladr, C. (2018). Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach. *The Guardian*. Retrieved from https://www.theguardian.com/news/2018/mar/17/cambridge-analytica-facebook-influence-us-election

Guha, R. & Brickley, D. (2014). *RDF schema 1.1*. W3C. http://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

Gürses, S. (2010). *Multilateral privacy requirements analysis in online social network services* (Doctoral dissertation, KU Leuven, Heverlee).

Haber, S. & Stornetta, W. S. (1991, January). How to time-stamp a digital document. *Journal of Cryptology*, *3*(2), 99–111.

Hoepman, J.-H. & van Lieshout, M. (2012). Cyber Safety: An Introduction. (Chap. Privacy, pp. 75–87). The Hague: Eleven International Publishing.

Irvine, D. (2010). *MaidSafe Distributed File System*.

Jefferys, K., Harman, S., Ross, J., & McLean, P. (2018, March 1). Loki: Private transactions, decentralised communication. Retrieved from https://loki.network/

Kelsey, J. (2016). SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash. *NIST Special Publication*, *800*, 185.

Khan, O. (2015, July 15). Major Email Provider Trends in 2015: Gmail's Lead Increases. Mailchimp. Retrieved from https://blog.mailchimp.com/major-email-provider-trends-in-2015-gmail-takes-a-really-big-lead/

Kleeman, S. (2015, May 29). In One Quote, Snowden Just Destroyed the Biggest Myth About Privacy. Retrieved June 14, 2018, from https://mic.com/articles/119602/in-one-quote-edward-snowden-summed-up-why-our-privacy-is-worth-fighting-for

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*, *48*(177), 203–209.

Kumar, A., Fischer, C., Tople, S., & Saxena, P. (2017). A Traceability Analysis of Monero's Blockchain. *IACR Cryptology ePrint Archive*, *2017*, 338.

Lambert, N., Ma, Q., & Irvine, D. (2015). *Safecoin: The Decentralised Network Token*. MaidSafe, Tech. Rep.

Lamport, L. (1978). Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, *21*(7), 558–565.

Lanthaler, M., Cyganiak, R., & Wood, D. (2014). *RDF 1.1 concepts and abstract syntax*. W3C. Retrieved October 9, 2017, from http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/

Lassila, O. & Swick, R. R. (1997). *Resource Description Framework (RDF): Model and syntax*. W3C. Retrieved October 20, 2017, from https://www.w3.org/TR/WD-rdf-syntax-971002/

Law, L., Menezes, A., Qu, M., Solinas, J., & Vanstone, S. (2003). An efficient protocol for authenticated key agreement. *Designs, Codes and Cryptography*, *28*(2), 119–134.

Levien, R. & Aiken, A. (1998). Attack-resistant trust metrics for public key certification. In *Usenix security symposium* (pp. 229–242).

Lewkowicz, K. (2017, March 21). Here's What We Learned After Tracking 17 Billion Email Opens [Infographic]. Retrieved June 14, 2018, from https://litmus.com/blog/2016-email-client-market-share-infographic

Locher, T., Mysicka, D., Schmid, S., & Wattenhofer, R. (2010). Poisoning the Kad Network. *ICDCN*, *10*, 195–206.

Marlinspike, M. (2013, November 26). Advanced cryptographic ratcheting. Retrieved May 30, 2018, from https://signal.org/blog/advanced-ratcheting/

Marlinspike, M. (2017, September 26). Technology preview: Private contact discovery for Signal. Retrieved May 30, 2018, from https://signal.org/blog/private-contact-discovery/

Maurer, U. M. (1994). Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms. In *Annual international cryptology conference* (pp. 271–281). Springer.

Maymounkov, P. & Mazieres, D. (2002). Kademlia: A peer-to-peer information system based on the XOR metric. In P. Druschel, F. Kaashoek, & A. Rowstron (Eds.), *Peer-to-peer systems* (Vol. 2429, pp. 53–65). Lecture notes in Computer Science. 1st International Workshop on Peer-to-Peer Systems, Cambridge, Massachusetts, Mar 07-08, 2002. Microsoft Res. Heidelberger Platz 3, D-14197 Berlin, Germany: Springer-Verlag Berlin Heidelberg.

MIL-HDBK-232A. (1987). Red/Black Engineering – Installation Guidelines.

Miller, V. S. (1986). Use of elliptic curves in cryptography. In H. C. Williams (Ed.), *Advances in cryptology — crypto '85 proceedings* (pp. 417–426). Berlin, Heidelberg: Springer Berlin Heidelberg.

Montgomery, P. L. (1987). Speeding the pollard and elliptic curve methods of factorization. *Mathematics of computation*, *48*(177), 243–264.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

Noether, S., Mackenzie, A. et al. (2016). Ring confidential transactions. *Ledger*, *1*, 1–18.

Nystrom, B. (2015, February 1). What Color is Your Function? Retrieved June 16, 2018, from http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

Okamoto, T. & Ohta, K. (1991). Universal electronic cash. In *Annual international cryptology conference* (pp. 324–337). Springer.

Opsahl, K. (2013, June 7). Why Metadata Matters. Electronic Frontier Foundation. Retrieved May 30, 2018, from https://www.eff.org/deeplinks/2013/06/why-metadata-matters

Ratnasamy, S., Francis, P., Handley, M., Karp, R., & Shenker, S. (2001). A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev. 31*(4), 161–172.

Rivest, R., Shamir, A., & Tauman, Y. (2001). How to leak a secret. *Advances in Cryptology—ASIACRYPT 2001*, 552–565.

Rowstron, A. & Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing* (pp. 329–350). Springer.

Ryssdal, K. (2014, November 18). Uber's data makes a creepy point about the company. *Marketplace*. Retrieved May 19, 2018, from https://www.marketplace.org/2014/11/18/business/final-note/ubers-data-makes-creepy-point-about-company

San Martín, M. & Gutierrez, C. (2009). Representing, querying and transforming social networks with RDF/SPARQL. In *European semantic web conference* (pp. 293–307). Springer.

Schnorr, C.-P. (1991a). Efficient signature generation by smart cards. *Journal of cryptology*, *4*(3), 161–174.

Schnorr, C.-P. (1991b). *Method for identifying subscribers and for generating and verifying electronic signatures in a data exchange system*. 4995082.

Shen, X., Yu, H., Buford, J., & Akon, M. (2010). *Handbook of Peer-to-Peer Networking*. Springer Science & Business Media.

Shor, P. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, *41*(2), 303–332.

Signal. (2018). Signal – Privacy Policy. Retrieved May 30, 2018, from https://signal.org/signal/privacy/

Smith, B. (2014–2018). ring: Safe, fast, small crypto using Rust. Retrieved June 17, 2018, from https://github.com/briansmith/ring

Snowden, E. (2015, May 21). Reddit – Just days left to kill mass surveillance under Section 215 of the Patriot Act. We are Edward Snowden and the ACLU's Jameel Jaffer. AUA. Retrieved May 29, 2018, from https://www.reddit.com/r/IAmA/comments/36ru89/just_days_left_to_kill_mass_surveillance_under/crglgh2/

Soler, C. (2017). *A generic data exchange system for friend-to-friend networks*. INRIA Grenoble-Rhone-Alpes.

Tinker, B. (2018, April 11). How Facebook 'likes' predict race, religion and sexual orientation. *CNN*. Retrieved May 30, 2018, from https://edition.cnn.com/2018/04/10/health/facebook-likes-psychographics/index.html

Todd, P. (2014). [bitcoin-development] Stealth addresses. Retrieved December 2, 2017, from https://www.mail-archive.com/bitcoin-development@lists.sourceforge.net/msg03613.html

Troncoso, C., Isaakidis, M., Danezis, G., & Halpin, H. (2017). Systematizing decentralization and privacy: Lessons from 15 years of research and deployments. *Proceedings on Privacy Enhancing Technologies*, *2017*(4), 404–426.

user 'bytecoin'. (2011). Untraceable transactions which can contain a secure message are inevitable. Retrieved December 2, 2017, from https://bitcointalk.org/index.php?topic=5965.0

Van Cutsem, T., Mostinckx, S., Gonzalez Boix, E., Dedecker, J., & De Meuter, W. (2007). AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. (pp. 3–12).

van Saberhagen, N. (2013). Cryptonote v2.0.

Wang, L. & Kangasharju, J. (2013). Measuring large-scale distributed systems: case of BitTorrent Mainline DHT. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on* (pp. 1–10). IEEE.

Warren, S. D. & Brandeis, L. D. (1890). The right to privacy. *Harvard law review*, 193–220.

Wilson, S. (2016, May 3). Blockchain: Almost everything you read is wrong. Retrieved October 9, 2017, from https://www.constellationr.com/blog-news/blockchain-almost-everything-you-read-wrong

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., & Kubiatowicz, J. D. (2004). Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on selected areas in communications*, *22*(1), 41–53.

Zureik, E., Stalker, L. H., & Smith, E. (2010). *Surveillance, Privacy, and the Globalization of Personal Information: International Comparisons*. McGill-Queen's Press-MQUP.