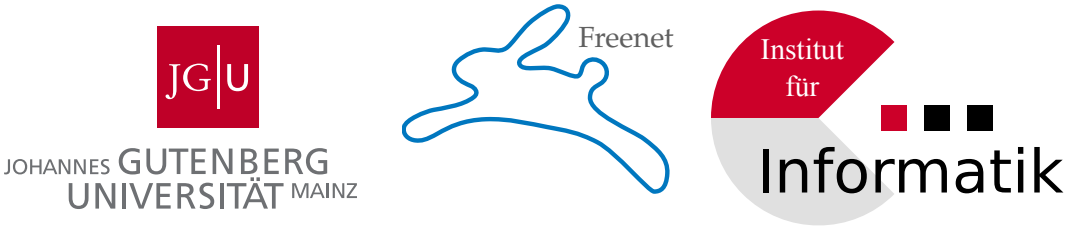


OPTIMIZING A DISTRIBUTED SPAM FILTER FOR FREENET

AKA

THE WEB OF TRUST DEVELOPER'S MANUAL

Bachelor's thesis

to obtain the academic degree Bachelor of Science submitted
to the Faculty Physics, Mathematics and Computer Science
of the Johannes Gutenberg-University Mainz

on 2015-09-01 by

xor@freenetproject.org

Version: 1.2-pdf

Submission date:	2015-09-01
First reviewer:	Univ.-Prof. Dr.-Ing. André Brinkmann *Head of the Center for Data Processing* *and of the Efficient Computing and Storage Group*
Second reviewer:	Univ.-Prof. Dr. Ernst Althaus *Head of Algorithmics*

ABSTRACT

The Freenet Project [1] aims to provide the decentralized, anonymous storage network "Freenet" which is resistant against censorship. It is built on top of the regular Internet, and serves as a foundation for anonymous implementations of a multitude of common web services such as mail, forums and social networking [2]. Due to the fact that potential attackers are also anonymous in Freenet, spam and Denial of Service (DoS) cannot be blocked with conventional measures such as IP blacklists. Thus, a system for creating user identities using cryptography had been implemented. It is called "Web of Trust"(WOT or WoT) [3] and allows prevention of spam by a mechanism where users vote upon how much other users can be trusted to not publish spam.
Due to the growing amount of users and poor choice of algorithms in the initial implementation, the performance of certain functions of WoT has deteriorated to the point where they can block the user interface (UI) for time spans close to a minute (page 49, [4]). These functions are vital for the daily use of WoT, and thereby such delays are not acceptable. The subject of this thesis shall be to improve their performance by replacing the affected algorithms with different ones.

GERMAN ABSTRACT

Das Freenet Projekt [1] hat die Zielsetzung, das dezentrale, anonyme Speichernetz "Freenet" zur Verfügung zu stellen, welches zensurresistent ist. Es baut auf dem regulären Internet auf, und dient als Fundament für anonyme Implementierungen einer Vielfalt an bekannten Web-Diensten wie Mail, Foren und sozialer Netzwerke [2]. Aufgrund der Tatsache, dass potentielle Angreifer in Freenet auch anonym sind, sind Spam und Denial of Service (DoS) mit konventionellen Maßnahmen wie IP-Sperren nicht zu verhindern. Daher wurde einen System zur Erzeugung von Nutzer-Identitäten mittels Kryptographie implementiert. Es heißt "Web of Trust" (WOT oder WoT) [3] und erlaubt das Verhindern von Spam mittels eines Mechanismus bei dem Nutzer eine Wahl durchführen, in der die Stimmen darüber entscheiden, ob andere Nutzern vertraut werden kann keinen Spam zu veröffentlichen.
Aufgrund der wachsenden Anzahl an Nutzern und schlechter Wahl der Algorithmen in der ursprünglichen Implementierung ist die Performanz bestimmter Funktionen von WoT derart gesunken, dass diese das User-Interface für Zeiträume nah an einer Minute blockieren können (Seite 49, [4]). Diese Funktionen sind für die tägliche Nutzung von WoT zwingend notwendig, daher sind solche Wartezeiten nicht akzeptabel. Das Thema dieser Bachelorarbeit soll sein, deren Performanz zu verbessern, indem die betroffenen Algorithmen mit anderen ersetzt werden.

ACKNOWLEDGEMENTS

Sincerest and utmost gratitude shall be expressed to all volunteer developers who have kept the spirit of the Freenet Project alive for 16 years [5] as of Summer 2015.

Equally important are the donors who have helped Freenet core development to have a stable source of development effort for well over 10 years, as well as Matthew Toseland for using these funds to not only dedicate his workforce but also his personal passion to Freenet for this long time [6].

The same importance can also be testified to the ~10 000 users who donate their bandwidth and disk storage of ~140 TB to Freenet [7].

Last but not least, the list of acknowledgments shall be closed with my personal thanks to the staff of the University of Mainz for enduring my slightly stubborn but hopefully bearable continuous ramblings about the aesthetics of the technology behind Freenet. I hope this can be forgiven when considering how captivating the idea behind it can be: A technology such as Freenet may someday return the Internet back to the level of freedom of knowledge which it once seemed so promising to provide, but was deprived of by greed and enforcement of political ideologies.

CONTENTS

LIST OF FIGURES

LIST OF PSEUDOCODE

LIST OF ABBREVIATIONS

API	Application Programming Interface
BFS	Breadth-First Search algorithm [8]
CAPTCHA	Completely Automated Public Turing test to tell Computers and Humans Apart [9]
cRFS	*computeRankFromScratch()*, see page 33
DoS	Denial of Service. Also referred to as “spam”.
ID	IDentifier
SPSP	Single-Pair Shortest-Path problem [10]
SSSP	Single-Source Shortest-Paths problem [11]
UI	User Interface
UCS	Uniform-Cost Search algorithm [12] [13]
USK	Updateable Subspace Key [14]
WoT	Web of Trust
WOT	same as WoT
XML	eXtensible Markup Language [15]

1 Introduction

This thesis aims to be self-contained: No prior knowledge about Freenet or Web of Trust shall be required.
Hence, it is recommended to read the following introduction to get a sufficient overview of what Freenet and Web of Trust are.

1.1 What is Freenet?

Freenet [1] is an anonymous peer-to-peer network which operates on top of the regular Internet.
Unlike the currently more widely known Tor [16] [17], the goal of Freenet is *not* to allow anonymous access to regular websites. Instead, it aims to be a self-contained storage network which hosts content of its own. Each user donates part of their disk space, and Freenet uses it to store content of remote users. The content is stored in an encrypted fashion, and the decryption key is *not* available to the users who store the content.
As users cannot even look into whats in their database, they are believed to not be legally liable for what they store. Like a bank, they provide the service of "safe deposit boxes", and should not be held accountable for what is in them.
Only the person who uploaded a certain piece of content has got the decryption key, and may share this "address" with other people, just like a regular web address, so they may retrieve and decrypt the content.
The motivation behind providing storage instead of merely anonymizing access to regular websites can be understood from the primary goal of Freenet which we will now get to know.

1.1.1 Freenet's primary goal: Prevention of censorship

The legacy architecture of regular websites includes a single point of failure: The server at which a website is stored is a central instance which can be disabled. With the Domain Name System (DNS), which provides user-friendly names such as "www.freenetproject.org", the registry which controls the assignment of those names even knows the physical owner of the server.
When thinking about services such as Tor which anonymize access to such regular websites, it becomes apparent that this strategy is not robust against censorship as de-anonymization is not the attack vector for censorship there:
A fascist government may of course have an interest in conducting censorship by de-anonymizing Tor users: By putting them in prison then, they are prevented from consuming certain information. But it may even have a bigger interest in just deleting the whole sites. If the content is gone, even users of Tor cannot access it. And disabling a single website is much less work than tracking down potentially thou-

sands of users of it. Censorship is thus very easy as only the server or even its owner needs to be attacked.
Because of this, Tor does provide the concept of "hidden services" to anonymize the exit point at which a website is hosted. For ease of understanding, this can be summed up as a special ".onion" domain name, where the target Internet address is hidden by Tor.
Unfortunately, this is still a single point of failure: A hidden service is typically controlled by a single person, and resolves to a single target server. As a single person can only provide a limited amount of efforts to host a server [1] , an attacker may just conduct a Denial of Service (DoS) attack to take it down.
And even if no attack is conducted, it might be unwise to store certain content on a central server which may go away if one person shuts off the server: Content which for example reveals major political scandals should probably not only be stored in one place. It is just too important for society to risk its loss, a decentralized storage system should be preferred.
As anonymous networks are a very important retreat for publishing such valuable data, with Freenet it was decided to try to avoid this problem at the network layer already: Instead of storing the data of a Freenet website on the machine of the publisher, it is stored across the whole network on the machines of other users. Once the upload of a site is completed, the publisher may turn off his computer forever.

1.1.2 Self-adjusting redundancy

As we have just learned, Freenet users provide storage for each others. As availability is dependent on remote users now, this raises the questions of when data is deleted:
What happens if users upload more content than there is disk space available?
What happens if a user deletes their store by uninstalling Freenet?
Freenet solves this problem in a quite satisfactory way:
If a user requests the download of data, instead of transferring it directly from the storing peer to the requester, the data is always transported across many peers. While this is already necessary for the reason of providing anonymity, it hereby serves another purpose: The peers along the route will put the data into their cache. Upon low space, a pre-existing slot in the cache will be replaced at random choice to fulfill this operation.
The resulting effect is that popular data, i.e. such which is downloaded frequently, gets stored on more computers. Unpopular data eventually gets deleted due to random replacement [2].
The result is that Freenet is a self-organizing system. There is no librarian or any other central authority which may decide to delete data. Even the author of data

1 Outsourcing hosting to a data center which could provide load balancing would be a bad idea: The administrators might de-anonymize the owner.
2 We may now realize that Freenet is in fact what is commonly referred as a "distributed hash table" (DHT).

cannot delete it once uploaded! Availability of data is solely controlled by the balance between offer of disk-space and the demand of people who download it.
This lack of human control can be hoped to provide freedom of information, speech and knowledge in the most fair way possible.

1.1.3 Address and data types in Freenet

For the purpose of this thesis, we shall only need knowledge of one of the four address types in Freenet: The "Updateable Subspace Key" (USK) [14].
USKs allow an author to upload arbitrary file and directory structures. In turn, the author can provide an address of a file like:

USK@freenet-address/site-name/edition/directories.../file

The author may use USKs to publish whole file and directory structures. No restriction is put upon the content type of files, and their size is practically unlimited[3].
It is possible to update these structures by publishing a new "edition", which is merely an integer counting up from zero.
Control of updating an USK is restricted using asymmetric cryptography. Only the holder of the secret key, typically the author, may update it.
When a remote user accesses the USK, Freenet will validate the authenticity of the retrieved data: The "freenet-address" contains a reference to the public key of the author. The public key will be retrieved and used to validate the cryptographic signature of the data.
The basic use case of an USK are "Freesites": They use the same file formats as regular websites, and are also viewed using a browser - but are accessed through the Freenet web interface which acts as a gateway.
Freenet "client applications", such as Web of Trust which will be our subject, may:

- "subscribe" to an USK: Freenet will periodically query the network for new editions, and automatically ship them to the client application.
- publish USKs on their own, to use Freenet as the network layer for building their own advanced peer-to-peer systems on top of it.

1.1.4 Freenet in numbers

- ~16 years of development [5] as of 2015.
- ~350 000 lines of code [18] - including only some of the sub-projects [2] it consists of.
- ~10 000 users [7].
- ~140 TB of donated disk space [7].
- ~2 months of donation money left as funding for employees [19].

3 There is most likely a limit due to finite length integer arithmetic. Also, the availability of larger files depends more on the total disk space of the network.

1.2 What is Web of Trust?

1.2.1 The spam problem in anonymous networks

The flexibility which USKs offer allows the imagination of a plethora of isomorphisms which can be used to implement regular web services on top of Freenet.
There are implementations of all kinds of popular *communication* systems ranging from individual messaging to global communication [2]:

- Mail
- Social networking
- Blogging
- Micro-blogging (= what "Twitter" provides)
- Forums
- Websites
- Wikis
- Search engines

All these communication systems involve "one-to-many" communication, and so share a common attribute:
One user should be able to send content to many users *without* the remote users requesting it first. Nobody would use mail if it required frequent use of a manual command "download the mails I have received from my friend X" for every communication partner X. Instead, content of remote users should be retrieved without user interaction.
This *automatic* retrieval of content can be a problem in the context of how Freenet works:
An entity which wants to censor content cannot delete the content as Freenet does not allow deletion. It also cannot persecute users in the real world as they are anonymous. Thus, the primary remaining tool of censorship is the **"Denial of Service" attack** (DoS or spam): By polluting communication systems with useless data, they become unusable. The content to be censored will drown in the flood of "spam".
This attack is aggravated the fact that everyone in Freenet is anonymous: The attackers are also anonymous! They cannot be blocked by regular countermeasures such as blacklists of Internet addresses.
A further exacerbation is the scarcity of bandwidth in peer-to-peer networks: As they are powered by their users, not by large data centers, traffic is very expensive. This is even more the case in an anonymous peer-to-peer network: Anonymization usually works by redirecting traffic over many peers, and thus the available bandwidth will be reduced to the slowest link in the chain.
Consequently, there are two desirable features of spam prevention in Freenet:

- Due to the high probability of censors resorting to DoS / spam, spam prevention should be a library which can be used by all kinds of communication systems.
- Because of the high cost of network traffic, spam must not only be filtered out locally after it was downloaded. A spam filter must instead be *proactive*: Spam must not even be downloaded in the first place.

1.2.2 Web of Trust as a distributed, collaborative spam filter

The aforementioned requirements are the job of Web of Trust (WoT). It allows each user to create anonymous "Identities". They can be considered as a public pseudonym which serves as a "user account" to the communication systems built on top of WoT.
The social community of Identities can rate each others like in an election. The result of this poll decides whether a user is considered as trustworthy or as a "spammer". The Freenet client applications such as forums then become client applications of WoT: They will query WoT for a list of trustworthy Identities, and only download content from those. Identities which WoT deems to be spammers will be ignored.

1.3 The aim of this thesis

While WoT seems to do its job of filtering spam, it is yet still a historically grown mixture of work from mostly various volunteers. The principle of choosing and evaluating the used algorithms in a scientific fashion may have been ignored sometimes.
As a result, WoT is often perceived to be a really slow application in terms of excessive CPU usage and excessive execution times of basic operations such as removing ratings. Reports of the users and measurements show that core operations such as changing a vote upon another user can cause execution times in the order of magnitude of almost a minute (page 50).
The goal of this thesis shall therefore be to optimize performance by replacing the affected algorithms.
Albeit the thesis document will mostly focus on the algorithmic theory, a full Java implementation of the changes is also provided (page 67). It has been thoroughly tested, merged into the main Git tree of WoT [20–25], and will be included in the next WoT beta release.

2 Theoretical Background

To only require common computer science knowledge for understanding this thesis, this chapter will describe how Web of Trust worked before the thesis. It will do so in a way which requires no prior knowledge or further reading about Freenet or WoT technology whatsoever. Having read the previous introduction chapter is recommended though.
Where no further sources are cited, all knowledge in this section is obtained from studying the Freenet [26] and WoT source code [27].
Mathematical proofs of observations about the preexisting state of the WoT codebase will be avoided: It is beyond the time quota of this thesis to test the correctness of the existing code. It shall be taken for granted. [1]

2.1 The data model of Web of Trust

The central features of Web of Trust can be understood just by looking at 4 of its core Java classes.
Their roles shall now be elaborated by simplified Java pseudocode which represents the most important data they store and provide. “Simplified” hereby means that names of those four classes are kept as is, but names of member functions, member variables and class types of member variables are chosen differently to be more self-explanatory for people unfamiliar with the WoT codebase.

1 Correctness of new code written as part of this thesis will nevertheless still be possible to be tested: Unit tests will be used to compare the results of the old and new implementations.

Listing 1: Core classes of Web of Trust (simplified) [28]

```
1  class Identity {
2      FreenetDownloadAddress publicDownloadAddress;
3      String                 nickname;
4
5      String getID() {
6          return publicDownloadAddress.getHashOfPublicKey();
7      }
8
9      int getEdition() {
10         return publicDownloadAddress.getEdition();
11     }
12 }
13
14 class OwnIdentity extends Identity {
15     FreenetUploadAddress    secretUploadAddress;
16 }
17
18 class Trust {
19     Identity    truster;
20     Identity    trustee;
21     byte        value;    // Range: [−100, +100]
22 }
23
24 class Score {
25     OwnIdentity    truster;
26     Identity       trustee;
27     int            rank;     // Range: [0, ∞]
28     int            capacity; // Range: [0, 100] (= percentage)
29     int            value;    // Range: [−∞, +∞]
30 }
```

CLASS IDENTITY (line 1) This class represents a remote user of WoT. The *publicDownloadAddress* is the Freenet "Updateable Subspace Key" (USK) [14] at which the Identity is published as XML [15] and can be downloaded from. The XML which contains an Identity will also further be referred to as "Identity file". A Freenet USK is an address which is defined by a cryptographic public/private key pair. The owner of the private key, which is also the owner of the Identity, is the only person who can publish at the USK. Any data uploaded to the USK will be signed with the private key by the owner. By knowing the USK *publicDownloadAddress*, not only can WoT download a remote Identity - it can also validate that the downloaded version was uploaded by the owner, not by an attacker: The USK contains the hash of the public key, and thus allows Freenet to download the public key and validate the signature of downloaded identity files. This is done automatically by Freenet - it is not possible for a tampered version of an identity to be downloaded from its USK. Freenet will discard data which does not contain a valid cryptographic signature before even delivering it to WoT.
An Identity is uniquely described by its identifier (ID) which is the aforementioned hash of the public key of the USK. Due to the sufficiently high entropy in the way such public keys and therefore hashes are generated, it can be assumed that the

probability of two independent Identities holding the same ID is infinitesimally small. Thus, there can only ever be one Identity globally with a given ID.
The Identity file is published in *editions* (also described as "versions"). The edition is an integer which counts up from 0 inclusive. Each edition is a fully self-contained copy of an Identity, and replaces lower editions. By downloading only the latest edition, WoT has all current data of the Identity. One of the main jobs of WoT is to monitor the network for new editions of identities, and download them. It is also responsible for discovering identities by checking downloaded identity files for links to USKs of yet unknown identities. Thus, the peer-to-peer network layer of WoT is equal to the identity files[2].
Besides the nickname and various preferences of an Identity, an Identity file usually contains a so called *trust list*. A trust list is a set of objects of class Trust, which we will also discuss soon.

CLASS OWNIDENTITY (line 14) Similar to class Identity, this represents a user of WoT. But instead of a remote user, an OwnIdentity represents a local user.
The existence of multiple OwnIdentity objects does *not* mean that there are multiple physical users at the machine. It is neither encouraged nor technically supported to allow multiple users to use a single WoT installation[3].
Instead, single users are encouraged to use multiple OwnIdentities as a measure to improve their anonymity: By restricting each type of activity such as posts in different forums to a single OwnIdentity, the user can limit the amount of correlated information which is available about him.
For our purposes, the quintessence of OwnIdentities is: We will understand them as "a local user of WoT". We will assume their amount to be small.

CLASS TRUST (line 18) Objects of class Trust, also named Trust *values* in reference to their most important member variable, are the main 'input' of the user to the computations of WoT. By storing a Trust object, an Identity referred as *truster* assigns a rating in the range [-100, +100] to another Identity, called the *trustee*. The *truster* may also be considered as the Trust's "giver" or "source", and the *trustee* as the "receiver" or "target". Positive values, *including* zero, are interpreted as "trustworthy", negative values are interpreted as "not trustworthy" (or "distrusted").
For a given pair of (*truster*, *trustee*), there can only be a single Trust - just like in a real election, multiple votes upon the same candidate are not allowed[4]. It *is* however allowed to cast multiple votes by giving multiple Trust values to *different* trustees.

2 Identities also publish so called "introduction puzzles" which are are separate network layer. They are what is commonly known as CAPTCHAs ("Completely Automated Public Turing test to tell Computers and Humans Apart") [9] and prevent attackers from creating an infinite amount of identities to forge Trust votes. The introduction puzzles are not relevant to this thesis though.
3 This is due to a lack of a password mechanism for example. It might be implemented in the future to support public Freenet gateways.
4 The purpose of the CAPTCHA mechanism which was mentioned in a previous footnote now becomes apparent: It attaches a cost to the creation of an Identity to prevent users from creating multiple Identities for the purpose of being able to cast *multiple* votes upon a single trustee.

By convention and encouragement via UI design, users are instructed to use negative ratings *only* in the case where the rated Identity publishes spam or does a DoS attack. Or said inversely, the purpose of the Trust mechanism is *not* to punish other users for having a different opinion.
WoT will accumulate the Trust values of all Identities eligible for voting by downloading their Identity files and importing the contained trust lists. Whether an Identity is eligible for casting a vote is determined by its *capacity*, which the following paragraph about class Score will introduce.

CLASS SCORE (line 24) The Score objects, also named Score *values* in reference to their most important member variable, are the main 'output' of WoT's computations. They are not created manually by the user: The Score computation algorithm, which will be discussed in the following section, produces Score objects from all known and eligible Trust objects.
The Score objects are what is delivered to the actual "client applications" built on top of WoT, for example forum systems. The client apps use the Scores to decide whether content shall be downloaded from an Identity; or whether the Identity is considered as a spammer and thus its content must not be retrieved.
A Score's *value* indicates whether a WoT client application should consider the Identity *trustee* as trustworthy or distrusted from the perspective of the OwnIdentity *truster*. The *truster* may also be considered as the Score's "giver" or "source", and the *trustee* as the "receiver" or "target". If a Score value is positive, *including* zero, content - in our example forum posts - should be downloaded from the Identity trustee and displayed in the UI of the OwnIdentity truster. If it is negative, *or*[5] if there is no Score object for the given Identity, the Identity is considered as distrusted and its content should not be downloaded. It is noteworthy that WoT will not only use the "should this Identity be downloaded?" output of its computations to service client apps, but also for its own purposes: It will not download Identity files from Identities with a bad Score, to prevent Denial of Service / spam upon WoT itself [6].
For every OwnIdentity, WoT will try to compute a Score for all other Identities to rate them from the perspective of the given OwnIdentity. Notably, if an OwnIdentity has assigned a Trust value to another Identity, the Trust value will always overwrite the other Identity's Score value, which would normally instead be computed from remote Trust objects. By observing this fact, we can learn that Score values

5 If WoT does not create a Score object for a certain Identity, this must still be interpreted as the Identity being distrusted: There are so few Trust objects reaching the Identity that it is not possible to assign a rating to it. Thus, for security reasons, in this situation of doubt, the safer approach of considering the Identity as not trustworthy is chosen.

6 Identities may be downloaded upon a special condition even if they have a negative Score value: If they are eligible for casting Trust votes, which they are when their Score *capacity* is above 0. This is to ensure *stable* behavior of the algorithm in spite of random permutations of the order in which Identities are downloaded: Changes in the order of download of Identities, which can at any time happen due to the random nature of networking, should not change the output Score values of the algorithm.

could also be called "computed Trusts": Where the user himself was not able decide himself how high the Trust value for an Identity should be, WoT jumps in and does the job of *inferring* the trustworthiness by computing a Score value from other users' Trust values.
The Score value then is computed as a *weighted average* of the Trust values of the other users.
The weight of each Trust included in a Score is decided from the *capacity* field of another Score: The Score of the Identity which gave the Trust. For example when computing a Score given by OwnIdentity O to Identity A, and considering a Trust T from Identity B to Identity A, then the Trust's value is weighted by the Score *capacity* of the Score given from O to B. The capacity serves as a percentage which weights the Trust value.
The capacity itself is computed from the *rank*. The rank of an Identity who received the Score is a measurement of the "distance" to the OwnIdentity who is assigning the Score. The distance is measured by a function of Trust steps, with some limitations which will be discussed further. For now, an example can be given as: OwnIdentity O trusts Identity A trusts Identity B -> *rank* = 2.
The computation and meaning of rank and capacity shall become more apparent when we now proceed to discuss the Score computation algorithm.

2.2 The Score computation algorithm

Before this thesis was written, there were two Score computation algorithms present in WoT:

1. The reference implementation *computeAllScoresWithoutCommit()* which recomputes *all* Scores from scratch, i.e. works upon an empty Score database, just by considering the Trust objects.

2. An optimized *incremental* Score computation algorithm which is able to only update those Scores which need to be updated after a single Trust object changed.

We will now consider those existing implementations to eventually learn about their insufficiencies.

2.2.1 Reference implementation: Full recomputation from scratch

The function *computeAllScoresWithoutCommit()* [7] [29] is sufficiently self-contained to allow full understanding of how Scores are computed, and is able to recompute all Score objects from Trust objects, not relying upon pre-existence of any Scores [8].

7 Notice: The "withoutCommit" part of the function name only refers to the fact that the function does not commit the database transaction, it is not of interest to us.

8 To be factually correct, it shall be noted that actually the function does need a certain type of special Score objects to pre-exist: For each OwnIdentity, a Score assigned from the OwnIdentity to itself must

Thereby, the JavaDoc labels it as the reference implementation of Score computation, and thus it is critically important for this thesis to understand how this algorithm works.
Hence, the function shall now be summarized by Java pseudo-code which shows its core computations while ignoring handling of errors, boundary conditions, etc.
It is split into three stages, which we will attempt to understand now:

1. Computing ranks
2. Computing capacities
3. Computing Score values

2.2.1.1 Computing ranks

For understanding rank computation, let us consider the following isomorphism: Where I is the set of all Identity objects, and T is the set of all Trust object, we consider them as:

$$\text{Graph } G = (V = I, E = T)$$

Thus, where the Identities are the vertices, giving a Trust from an Identity A to an Identity B yields a directed edge A -> B, with the Trust value as the weight of the edge. We will call this the **Trust graph** from now on.
Notice: What is considered as the weight of a Trust edge will often be different in the following elaborations. But usually it will still be a *function* of the Trust value.
The goal of the rank computation stage is to compute **shortest paths** upon this graph in a special manner. The shortest paths are what will be called "rank". Let us now have a look at how those shortest paths are computed, in a Java pseudocode which was simplified for ease of understanding [9] :

exist. The Score will have special constant values of *value*, *rank* and *capacity*. This seems to be a trick to make the algorithm yield the same results with empty databases as the incremental implementation: If incremental computation is compared to the mathematical method of induction, then the self-Score is the base-case to get it started. It was not investigated in detail why full Score computation does not automatically create those objects if they do not exist, but it can be speculated that this is only needed in a unit test situation: In normal operation, *createOwnIdentity()* will always create such a Score object, and it will persist forever. Thus, we can ignore this special "self-Score" for now.

9 Parts of the pseudocode are more inefficient compared to the actual implementation. This was solely done for readability. It will not cause any issues when comparing performance as we will do that by benchmarks of the original code, not using the pseudocode here.

Listing 2: Rank computation reference implementation (simplified) [30]

```
for(OwnIdentity source : getAllOwnIdentities()) {
    Map<Identity, Integer> ranks              = new HashMap<>();
    Queue<Identity>        unprocessedVertices = new LinkedList<>();

    ranks.put(source, 0)
    unprocessedVertices.addLast(source);

    while(unprocessedVertices.isEmpty() == false) {
        Identity vertex     = unprocessedVertices.removeFirst();
        Integer  vertexRank = ranks.get(vertex);

        if(vertexRank == ∞) {
            // If an Identity has a rank of ∞ it is not allowed to give a rank
            // to its trustees.
            // Notice: The Map is initialized to "null" ranks, not ∞
            continue;
        }

        for(Trust edge : getGivenTrusts(vertex)) {
            int weight;

            if(edge.value > 0)
                weight = 1;
            else
                weight = ∞;

            Identity neighbourVertex  = edge.getTrustee()
            int      neighbourRank    = vertexRank + weight;
            Integer  neighbourRankOld = ranks.get(neighbourVertex);
            boolean  neighbourSeen    = ranks.get(neighbourVertex) != null;

            if(!neighbourSeen) {
                ranks.put(neighbourVertex, neighbourVertexRank);
                unprocessedVertices.addLast(neighbourVertex);
            } else if(neighbourVertexRank < neighbourVertexRankOld) {
                if(getTrust(source, neighbourVertex) == null) {
                    ranks.put(neighbourVertex, neighbourVertexRank);
                } else {
                    // The old rank must have come from a Trust of the source
                    // OwnIdentity since its Trusts are added to the queue
                    // first.
                    // If an OwnIdentity assigns a Trust, this is a mandatory
                    // decision: It shall overwrite all effects of Trusts of
                    // remote Identities. Thus, the rank received from other
                    // Identities is not used in that case.
                }
            }
        }
    }

    for(Identity target : getAllIdentities()) {
        Integer rank = rankValues.get(target);

        if(rank == null)
            getScore(source, target).delete();
        else
            getScore(source, target).setRank(rank);
    }
}
```

We shall notice that this aims to solve the so-called "SSSP" problem as defined by [11]:

> "**single-source shortest-paths problem**: given a graph $G = (V, E)$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$."

The weight of edges is chosen to be ∞ if a Trust value is negative or zero, and 1 otherwise.
It could be considered a mixture of the concepts behind the breadth-first search algorithm (BFS) [8] and less prominently Dijkstra's algorithm [31].
Edges are walked in a breadth-queue like in BFS.
Unlike with graphs supported by BFS, the edges do not only use the single weight of 1, but also the weight of ∞. This is why a mechanism of comparing against the current best known distance seems to have been added to be able to handle the two possible edge weights.
The Queue is not a PriorityQueue like in Dijkstra, which seems like a design issue. Because the goal of this thesis is not to improve the full recomputation, but to get rid of it completely in favor of an incremental one, this discovery has not been attempted to be fixed. It was documented in the Freenet bugtracker at [32].
To understand the output of the algorithm, please observe the two possible value ranges of the computed rank values:

1. $[0, \infty)$
2. ∞

Case 1 can be interpreted as: The rank is the number of Trust steps from the source OwnIdentity to the trustee. This aims to be a metric for "how good the source knows a trustee", i.e. how socially close they are: If Alice trusts Bob, and Bob trusts Charlie, it would be 2. If instead Alice trusts Bob, Bob trusts Charlie, Charlie trusts David, and David trusts Eric, a human would intuitively say that Alice and Eric know each other more distantly than Alice and Charlie as described in the former situation. And in fact, in the new situation, the rank will be 4 and therefore higher than 2. Thus, the rank is considered as an acceptable metric for measuring how close two Identities are socially.
Case 2 can be interpreted as: If an Identity has only received negative Trusts, it is to be considered as distrusted. Thus, it should not have a rank, as rank measures social closeness, and a distrusted Identity is rather isolated. For program logic reasons, to mark an Identity as distrusted, a Score has to exist - but the Score needs a rank, otherwise it would be deleted as seen in line 54. Thus, to be both able to give a rank, but also mark it as "not really a good one", the special value of ∞ is chosen. It could be interpreted as "the identity is socially ostracized to the point where it is infinitely far away from society"[10].

10 The author is unable to stop himself from hereby stating that this definition is a beautiful choice of the numeric value for the "no rank"-rank. It fits quite well!

2.2.1.2 Computing capacities

The goal of capacities is to be a factor which is used to weigh Trust values with when computing Score values from them.
Where rank computation was a way to define a metric how close an OwnIdentity and a remote Identity are in the social graph, capacities are a function of ranks to bend this metric to a usable 0 to 100% scale for the aforementioned purpose of weighting Trusts.
The way capacities are computed can be described by the following Java pseudocode which was simplified for ease of understanding [11] :

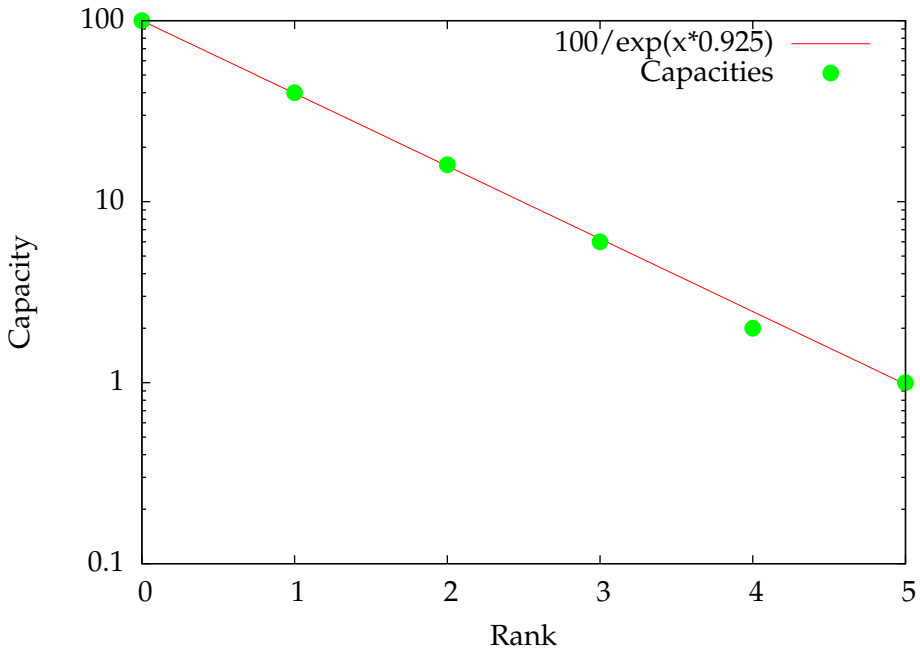
Listing 3: Capacity computation reference implementation (simplified) [33]

```
int computeCapacity(OwnIdentity source, Identity trustee, Integer rank) {
    if(rank == null) {
        // Notice: This case is not used here.
        // The rank not existing is represented by the Score object already
        // being deleted during rank computation. Nervertheless, for
        // understanding, it good to know the interpretation
        // "no rank = no capacity = no Score".
        return 0;
    }

    switch(rank) {
        case    0: return 100;
        case    1: return  40;
        case    2: return  16;
        case    3: return   6;
        case    4: return   2;
        case < ∞: return   1;
        case   ∞: return   0;
    }
}

assert(allScoreRanksHaveBeenComputed());

for(OwnIdentity source : getAllOwnIdentities()) {
    for(Identity target : getAllIdentities()) {
        Score score = getScore(source, target);

        if(score == null)
            continue; // no Score = no rank = no capacity

        score.setCapacity(computeCapacity(source, target, score.getRank()));
    }
}
```

Without any actual proof in the documentation of the WoT source code, the following loose observations were made for understanding why the capacity steps were chosen like that:
By plotting the atomic capacity steps using an exponential y-axis, we are able to conclude that they follow an **exponential decay** model.

11 The code which was removed in comparison to the official version of this function is likely not needed anyway, which was documented at bugtracker entry [32].

Figure 2.1: Capacity value decay with growing rank



The fact that the lowest possible value is reached after rank >= 5 would be possible to guess to be related to the so called **"six degrees of separation theory"**. As the capacity values will not be subject to any work in this thesis we settle for citing the Wikipedia article [34] to define it:

> "Six degrees of separation is the theory that everyone and everything is six or fewer steps away, by way of introduction, from any other person in the world, so that a chain of "a friend of a friend" statements can be made to connect any two people in a maximum of six steps. "

For further reading, please consult more reliable sources. Notably, the WoT source code states that the values were inspired by "Advogato's Trust Metric" [35].
Some non-scientific naive comments can nevertheless be made:
It shall be noted that the choice of capacity decay seems sound: The fact that every human being is reachable in only 6 "Trust" steps shows that most decay of capacity should happen before rank 6 to be soon enough to not give full capacity to everyone. The chosen boundary of 5 is the next logical choice.
Also, the chosen function of exponential decay is a "natural" choice, i.e. a standard fast decay function used in many decay processes.
Thus, it was decided to leave the capacity steps tuned as is during this thesis.

2.2.1.3 Computing Score values

Up to now, we have understood how the reference implementation of Score computation computes the fields Score.*rank* and Score.*capacity*. What is missing is the actual "output" of Score computation: The *value* field.
When an OwnIdentity O assigns a Score to a remote Identity X, the value of the Score is supposed to be a **weighted average** of the Trust values which X has received. The value shall be delivered to applications built on top of WoT so they can decide whether the content which X has published is trustworthy or not. Positive values, **including** zero, indicate trust, negative ones indicate distrust. (For further explanation, go back to page 10.)
Now that we have understood how the capacity uses the rank to define a percentage which is a measurement for how socially close an Identity is to an OwnIdentity, we will be at ease to understand how Score values are computed using the capacity. Thus, let us look at a simplified pseudocode:

Listing 4: Score value computation reference implementation (simplified) [36]

```
1  assert(allScoreRanksHaveBeenComputed());
2  assert(allScoreCapacitiesHaveBeenComputed());
3
4  for(OwnIdentity source : getAllOwnIdentities()) {
5      for(Identity target : getAllIdentities()) {
6          Score targetScore = getScore(source, target);
7
8          if(targetScore == null)
9              continue; // No rank = no Score = no Score value
10
11         Trust sourceTrust = getTrust(source, target);
12
13         if(sourceTrust != null) {
14             // A Trust decision of an OwnIdentity overwrites all remote
15             // Trust decisions.
16             targetScore.setValue(sourceTrust.getValue());
17             continue;
18         }
19
20         int value = 0;
21
22         for(Trust trust : getReceivedTrusts(target)) {
23             Score trusterScore = getScore(source, trust.getTruster());
24             int   capacity;
25
26             if(trusterScore != null)
27                 capacity = trusterScore.getCapacity();
28             else
29                 capacity = 0; // No score = no rank = no capacity
30
31             value += (trust.getValue() * capacity) / 100;
32         }
33
34         targetScore.setValue(value);
35     }
36 }
```

As we see, value computation is trivial: For each OwnIdentity source, and each Identity target, sum up all trust values the target has received, and weight them with the capacity of the Truster from the perspective of the OwnIdentity.
By understanding this, we now have learned how the core algorithm of WoT works. To consolidate this knowledge, it is recommended to re-read the chapter about the WoT data model at page 7.

2.2.2 Incremental Score computation upon Trust change

Let us remember what was said earlier: Before this thesis was written, there were two Score computation algorithms present in WoT:

1. The reference implementation *computeAllScoresWithoutCommit()* which we just discussed in depth. It will recompute **the whole** Score database from scratch.

2. An optimized **incremental** Score computation algorithm which is able to only update those Scores which need to be updated after a single Trust object changed.

We will now learn to understand the existing incremental algorithm.
The need for such an algorithm becomes apparent if we remember how Identities are pushed across the network as explained in detail on page 8:
WoT does *not* download the full graph of all Trust values in a single file. Instead, small sets of trust values are downloaded in so-called "trust lists" which are published in periodically released new editions of Identity files.
Thus, as the Trust graph is discovered in small pieces, an algorithm which handles small pieces is needed: If the full Score computation algorithm was run for every single changed Trust value, it would very likely recompute many Scores which were not even affected by the changed Trust.
Hence, WoT contains a function *updateScoresWithoutCommit(Trust oldTrust, Trust newTrust)* which handles changes upon a single Trust. It can be simplified to the following pseudocode:

Listing 5: Old incremental Score computation implementation (simplified) [37]

```
1  void updateScoresWithoutCommit(Trust oldTrust, Trust newTrust) {
2      if(newTrust == null) {
3          // Trust deleted
4          computeAllScoresWithoutCommit(); // Explained after this code
5          return;
6      }
7
8      if(oldTrust != null && oldTrust.value > 0 && newTrust.value <= 0) {
9          // Given rank could change
10         computeAllScoresWithoutCommit(); // Explained after this code
11         return;
12     }
13
14     for(OwnIdentity source : getAllOwnIdentities()) {
15         Queue<Trust> unprocessedEdges = new LinkedList<>();
16         unprocessedEdges.addFirst(newTrust);
17
18         while(!unprocessedEdges.isEmpty()) {
19             Trust    edge   = unprocessedEdges.removeFirst();
20             Identity target = edge.getTrustee();
21
22             if(target == source)
23                 continue;
24
25             Score newScore = getScore(source, target).clone();
26             Score oldScore = newScore != null ? newScore.clone();
27                                               : new Score(source, target);
28
29             // Compute ranks from the ranks of the trusters, including the
30             // new Trust.
31             newScore.rank = computeRankFromExistingData(source, target);
32
33             // computeCapacity() does not query Trusts/Scores, it only uses
34             // the passed rank
35             newScore.capacity = computeCapacity(source, target, newScore.rank);
36
37             // Compiute value from the received Trusts inlcuding the new Trust.
38             newScore.value = computeValueFromExistingData(source, target);
39
40             boolean rankCannotBeGivenAnymore =
41                    (oldScore.rank >= 0  && oldScore.rank < ∞)
42                 && (newScore.rank == −1 || newScore.rank == ∞);
43
44             if(rankCannotBeGivenAnymore) {
45                 computeAllScoresWithoutCommit(); // Explained after this code
46                 return;
47             }
48
49             boolean capacityCannotBeGivenAnymore =
50                 oldScore.getCapacity > 0 && newScore.getCapacity == 0;
51
52             if(capacityCannotBeGivenAnymore) {
53                 computeAllScoresWithoutCommit(); // Explained after this code
54                 return;
55             }
56
57             // We're sure that we won't abort now so we can update the database
58             getScore(source, target).setRankCapacityValue(newScore);
59
60             boolean rankChanged = oldScore.rank != newScore.rank;
61             boolean capacityChanged = oldScore.capacity != newScore.capacity;
```

```
62
63              if(!rankChanged && !capacityChanged)
64                  continue; // Don't process trustees, they cannot be affected.
65
66              boolean canGiveCapacity = newScore.capacity > 0;
67              boolean canGiveRank = newScore.rank >= 0 && newScore.rank < ∞;
68
69              if(!canGiveCapacity && !canGiveRank)
70                  continue;
71
72              for(Trust givenTrust : getGivenTrusts(target))
73                  unprocessedEdges.addLast(givenTrust);
74          }
75      }
76  }
77
78  int computeRankFromExistingData(OwnIdenity source, Identity target) {
79      Trust sourceTrust = getTrust(source, target);
80      if(sourceTrust != null) {
81          // OwnIdentity decision always wins.
82          return sourceTrust.value > 0 ? 1 : ∞;
83      }
84
85      int bestTrusterRank = -1; // No rank
86
87      for(Trust trust : getReceivedTrusts(target)) {
88          Score trusterScore = getScore(source, trust.getTruster());
89
90          if(trusterScore == null || trusterScore.rank == ∞)
91              continue; // No rank to give, continue
92
93          // By assigning a positive Trust, a Truster is willing to give his rank
94          // By assigning a Trust <= 0, he is only willing to give a rank of ∞
95          int giveableRank = trust.value > 0 ? trusterScore.rank + 1 : ∞;
96
97          if(bestTrusterRank == -1 || giveableRank < bestTrusterRank)
98              bestTrusterRank = giveableRank;
99      }
100
101     return bestTrusterRank;
102 }
103
104 int computeCapacity(OwnIdentity source, Identity trustee, Integer rank) {
105     // Same as on page 15
106 }
107
108 int computValueFromExistingData(OwnIdentity source, Identity target) {
109     Trust sourceTrust = getTrust(source, target);
110     if(sourceTrust != null)
111         return sourceTrust.value; // OwnIdentity decision always wins.
112
113     int value = 0;
114     for(Trust trust : getReceivedTrusts(target)) {
115         Score trusterScore = getScore(source, trust.getTruster());
116         int capacity = trusterScore != null ? trusterScore.capacity : 0;
117
118         value += (trust.value * capacity) / 100;
119     }
120     return value;
121 }
```

Let us not be deterred by the large amount of code which constitutes this algorithm. Similar to the full score computation function we have discussed earlier, it operates in a BFS-manner:
A deleted, created or modified Trust value can cause the rank, capacity or value of the trustee to change, for example:

- If the trustee had not received a rank yet, if someone with a good rank starts to trust the trustee, this will allow the trustee to obtain a rank. This also applies inversely: If a Trust is removed, the rank its trustee had may disappear if it came from the removed Trust.
- Capacities are computed from rank, and thus change along with it.
- Score values are computed from Trust values, and thus a changed Trust may cause Score values to change.

Because of those possible changes, the incremental algorithm starts with updating the Score rank/capacity/value of the receiver of the changed Trust. This induces a chain of events though: If the rank or capacity of the Trust receiver has changed, his trustees might have their Score rank, capacity and value affected as well. They might have originally inherited the now changed rank or capacity from the Trust receiver which will change theirs now. Or due to having been assigned a Trust from him, they could also have their Score value changed due to his changed capacity which weights his Trust.
Consequently, the same will apply to further trustees. So the algorithm deals with all reachable trustees via BFS.
Now if we look more closely, we can see a problem though: Negative trust changes such as removal of Trust values, disappearance of capacity and worsening of rank all cause the algorithm to abort and fall back to full score recomputation by *computeAllScoresWithoutCommit()*.
This is obviously a huge performance issue (bugtracker: [38]): The full score recomputation code will recompute *all* Scores. It will ignore the knowledge that only a single Trust value has changed, and thus fail to use the opportunity to optimize its actions by only recomputing relevant Scores. With this observation, you have now learned the specific goal of this thesis: To amend the incremental recomputation algorithm to be able to handle those cases of disruptive Trust changes.
But before we move on to designing this optimization, let us first find out *why* the existing incremental algorithm aborts upon negative Trust changes:
Look at the case where the algorithm does *not* abort: If we consider giving a Trust value to someone as a "forward" arrow, the Queue will walk through the Trust edges in "forward" BFS-manner. First the changed Trust is walked along forward, then the given Trusts of the trustee, and so on. They are processed in waves of equal rank, like walking from the core of an "onion" outwards to its shell.
This behavior is able to trickle down "positive" Trust changes, i.e. capacity increases, correctly since they only affect precisely the "outer" circles of the Trust-

"onion". The giver of a Trust is not affected by the Trust, only the receivers, and their receivers.
Now what about negative Trust changes? If a Trust is changed to distrust Identity X, and the Trust was source of the Identity's capacity, then the capacity will *decrease* (= worsen) to 0 instead of growing. Naively, we would say that the algorithm could propagate this forwards just as positive Trust changes. But there is a problem with the subroutine *computeRankFromExistingData()* because it uses the *old* ranks of the trusters:
The distrusted Identity X may have previously received other Trust values than the changed one - which might also have been giving a rank $< \infty$ and thus a capacity of > 0.
So when calling *computeRankFromExistingData()* to compute the new rank from the *old* ranks of the received Trusts, their *old* given ranks $< \infty$ would win against the new rank of ∞ which the changed Trust value would give in theory. Or in other words: The old, good capacities of > 0 would win against the new bad capacity of 0 which the changed Trust would give in theory.
But: These higher capacities might actually not originate from a Trust chain whose direct source is an OwnIdentity. Instead, they might actually come from Trusts given by the distrusted Identity *itself*, i.e. could be the result of a *Trust circle*. The circle would route back the *old* capacities which the trustees had received from the truster and give the truster capacity which came from himself. Notice how this is not a problem with positive Trust changes: They cause growth of capacity. As capacity decreases with each Trust step, if the capacity of an Identity is grown, it will always be greater than the capacity of a pre-existing circle which starts at it, and thus win against the circle. So old circles can only win the capacity race if the capacity decreased due to the negative Trust changes which we just discussed.
Therefore, a new algorithm which would be able to handle the described situation could avoid walking in circles by doing a *search* for a valid path "backwards" to an OwnIdentity. OwnIdentities are the proper source of ranks / capacities, so if we search a full path to them, the resulting rank / capacity will be valid.
Finally, as the incremental algorithm walks forward, and merely distributes *existing* ranks, not computes them freshly by finding complete Trust paths, we can conclude that this algorithm is not suitable for doing what was just described as necessary. Thereby, our goal in the actual thesis work shall now be to find a replacement algorithm for filling in the cases where the current algorithm falls back to full Score recomputation.

3 Results and Discussion

3.1 Presuppositions of the optimized incremental Score computation

As we've just discussed, the pre-existing incremental Score computation algorithm refuses to handle certain special cases. Lets reconsider what is common among those to find out assumptions a fill-in algorithms could make.

Listing 6: Abort conditions of old incremental Score computation (simplified) [37]

```
1  void updateScoresWithoutCommit(Trust oldTrust, Trust newTrust) {
2      boolean trustDeleted
3          = newTrust == null; // Given rank is gone
4      boolean givenRankCouldChange
5          = oldTrust != null && oldTrust.value > 0 && newTrust.value <= 0;
6
7      if(trustDeleted || givenRankCouldChange) {
8          computeAllScoresWithoutCommit(); return;
9      }
10
11     ...
12     Score newScore = // Score of the trustee or one of his trustees;
13     Score oldScore = newScore.clone();
14
15     newScore.rank = ...; newScore.capacity = ...; newScore.value = ...;
16
17     boolean rankCannotBeGivenAnymore =
18             (oldScore.rank >= 0  && oldScore.rank < ∞)
19          && (newScore.rank == −1 || newScore.rank == ∞);
20
21     boolean capacityCannotBeGivenAnymore =
22         oldScore.getCapacity > 0 && newScore.getCapacity == 0;
23
24     if(rankCannotBeGivenAnymore || capacityCannotBeGivenAnymore) {
25         computeAllScoresWithoutCommit(); return;
26     }
27 }
```

The common factors here are at least:

- The rank given to the trustee has possibly disappeared since the Trust changed to not allow giving a rank anymore.
- The capacity given to the trustee has possibly disappeared since the Trust changed to not allow giving a capacity anymore.

This is identical to the problem with rank circles we have identified in the previous chapter. Thus, the core goal of a replacement algorithm shall be to search a new rank without running into circles (and then compute capacities from that). The replacement then also has to be propagated to the reachable trustee subgraph.
Additionally, the Score.value of the trustee might need updating due to the changed value of the Trust; and Score values can also change where capacities to weigh them have changed. But these parts are rather trivial to handle.

We shall now proceed by amending the algorithm to call our new replacement instead of *computeAllScoresWithoutCommit()*, which then shall handle the cases which we just described:

Listing 7: Refactored old incremental Score computation to call new algorithm (simplified) [39]

```
1 void updateScoresWithoutCommit(Trust oldTrust, Trust newTrust) {
2     if(wouldFallbackToComputeAllScores...) {
3         Identity trustee = newTrust != null ? newTrust.trustee
4                                             : oldTrust.trustee;
5 
6         updateScoresAfterDistrustWithoutCommit(trustee);
7         return;
8     }
9 }
```

Notice how the Trust value itself does not have to be passed to the new algorithm: The information which we just discussed about changed rank/capacity is all it needs to know, and will be conveyed by the "AfterDistrust" in the function name. (As usual, "WithoutCommit" is merely to meet the WoT programming style of marking functions which do not commit the database transaction on their own.)

3.2 The optimized incremental Score computation algorithm

The heart of the optimized algorithm can be simplified to the following Java pseudocode:

Listing 8: New optimized incremental Score computation algorithm core (simplified) [40]

```
1  class ChangeSet<T> {
2      T beforeChange;
3      T afterChange;
4  }
5
6  void updateScoresAfterDistrustWithoutCommit(Identity distrusted) {
7      Map<ScoreID, ChangeSet<Score>> rankChanged
8          = updateRanksAfterDistrustWithoutCommit(distrusted);
9
10     Map<ScoreID, ChangeSet<Score>> capacityChanged
11         = updateCapacitiesAfterDistrustWithoutCommit(rankChanged.values());
12
13     Set<ScoreID> valueChanged
14         = updateValuesAfterDistrustWithoutCommit(distrusted, capacityChanged);
15
16     // The Score database is up to date now.
17     // The sets can be used for generating event notifications for client
18     // applications for example.
19 }
```

This core part of the algorithm has a control flow which follows the presuppositions we made:
The distrusted Identity will suffer from a worsened rank, and its trustees may in turn suffer from inheriting worse ranks. Thus, line 8 will compute the set of Scores with changed ranks.
As capacities are computed from ranks, the capacities of the set of Scores which was just computed will be updated by line 11.
Finally, Score values are the sum of Trust values an Identity has received, each multiplied by the capacity weight factor. Thus, all Score values have to be updated where in the sum either a Trust value changed or the capacity changed. The job of 14 shall be to deal with his.
We may now proceed to discuss each of those three functions in detail.

3.2.1 Updating Score ranks

The entry-point of the new algorithm is determining whether the rank of the distrusted Identity changed, and which further rank changes this induces. It can be summarized by this Java pseudocode:

Listing 9: New optimized incremental Score computation algorithm: updateRanksAfterDistrust...() (simplified) [41]

```
Map<ScoreID, ChangeSet<Score>>
        updateRanksAfterDistrustWithoutCommit(Identity distrusted) {

    Queue<Score> scoreQueue = new LinkedList<>();
    Set<ScoreID> scoresQueued = new HashSet<>();
    Set<ScoreID> scoresCreated = new HashSet<>();

    // Add all Scores of the distrusted Identity to the queue.
    // We do this by iterating over all OwnIdentities instead via getScores():
    // There might *not* have been an existing Score object from every
    // OwnIdentity to the distrusted Identity if it had not received a Trust
    // value yet; and by the distrust it could now be eligible for having one
    // exist.
    // Thus, we must check whether we need to create a new Score object. We do
    // this by just creating all possible Score objects now, then trying to
    // compute a rank in the next loop, and  deleting Score objects again if
    // they cannot be justified by a rank.
    // (We do not have to do this for trustees of the distrusted Identity: A
    // distrusted Identity's Trusts cannot create Scores since it is not allowed
    // to give a rank to its trustees.)
    for(OwnIdentity treeOwner : getAllOwnIdentities()) {
        Score outdated = getScore(treeOwner, distrusted);
        if(outdated == null) {
            outdated = new Score(treeOwner, distrusted);
            scoresCreated.add(outdated.getID());
        }

        scoreQueue.add(outdated);
        scoresQueued.add(outdated.getID());
    }

    Map<ScoreID, ChangeSet<Score>> rankChanged = new HashMap<>(); // Result set

    Score score;
    while((score = scoreQueue.removeFirst()) != null) {
        // computeRankFromScratch() will be explained in a following pseudocode
        // starting at page 33.
        // It solves the "single-pair shortest-path problem" [10]
        // without relying upon pre-existing rank values, i.e. only from Trusts.
        int newRank = computeRankFromScratch(score.truster, score.trusee);

        if(score.rank == newRank)
            continue;

        if(newRank == -1) { // No rank
            score.delete();
            // If we created the Score ourself, don't tell the caller about the
            // deleted rank: There was no rank before, we had only created the
            // Score to cause an search for a possibly newly existing rank.
            if(!scoresCreated.contains(score.id))
```

```
51                  rankChanged.put(score.id, new ChangeSet<Score>(score, null));
52          } else {
53              Score oldScore
54                  = scoresCreated.contains(score.id) ? null : score.clone();
55
56              score.rank = newRank;
57
58              rankChanged.put(score.id, new ChangeSet<Score>(oldScore, score));
59          }
60
61          final OwnIdentity treeOwner = score.truster;
62
63          for(Trust edge : getGivenTrusts(score.trustee)) {
64              Identity neighbour = edge.trustee;
65
66              if(scoresQueued.contains(new ScoreID(treeOwner, neighbour)))
67                  continue;
68
69              Score touchedScore = getScore(treeOwner, neighbour);
70              if(touchedScore == null)
71                  // No need to create a Score: This function is only called upon
72                  // distrust. A distrusted identity cannot create Scores for its
73                  // trustees since ranks can not be given if the giver is
74                  // distrusted. With regards to creating Scores for the
75                  // distrusted Identity itself, we have already done this in
76                  // a separate loop.
77                  continue;
78              }
79
80              scoreQueue.add(touchedScore);
81              scoresQueued.add(touchedScore.id);
82          }
83      }
84
85      return rankChanged;
86  }
```

The function for updating ranks operates in a BFS-manner again.
We start at the distrusted Identity by recomputing its rank via *computeRankFromScratch()*, and then trickle down the changed rank to its trustee-subgraph.
computeRankFromScratch() is the most important difference to the old algorithm: This function is an optimized version of a standard graph algorithm to solve the **"single-pair shortest-path problem"** (SPSP) [10] to find a new rank *without* causing the problem of running into circles which was described at page 21. The problem was triggered by re-using old data instead of searching a new path, and a SPSP algorithm can search a completely new path "*from scratch*" as the function name indicates[1].
The optimization which was applied to the standard SPSP algorithm can also be considered as quite interesting, which is why its code will not be discussed now but in a section following on page 33.
If the concept of rank circles was too complex to understand, the problem can also be explained in an incomplete but more simple manner like this:

1 Notably, an algorithm which searches a shortest path cannot yield circles as walking in a circle is longer than the shortest path. But our problem really was more the creation of data which includes circles by mixing old and new data.

A shortest path search is necessary as a new negative Trust value means that an identity is *not* allowed to give a rank to its trustees anymore. Thus, the other Trust values which the trustees have received must be considered as potential entry points for a shortest path to obtain a new rank. So the rank which came from the disappeared Trust edge is attempted to be "re-routed" by finding a new shortest path to a source OwnIdentity.
If the rank of the distrusted Identity changed, we then continue with updating the ranks of its trustees: They got their ranks from the distrusted Identity, so if its rank changed, theirs will as well. The problem then additionally propagates to their trustees, and so forth.

PERFORMANCE While the performance of the whole new incremental Score computation algorithm will be measured at page 42, we nevertheless shall already observe *why* its performance may be improved due to the pseudocode which we just discussed:
The old algorithm did fall back to using *computeAllScoresWithoutCommit()*, which will recompute the ranks of all Identities, i.e. solve the single-source shortest *paths* problem [11]; while the new algorithm solves the single-pair shortest *path* problem [10] multiple times. At best, it is even only solved once: If the rank of the distrusted Identity did not change, the rank of its trustees will not be recomputed. This allows us to observe:

Lemma 1 *Let* $N = |\mathtt{Identities}|$ *and let* $|\mathtt{OwnIdentities}| = \mathtt{const}$.
In the case where the old algorithm had to try to find $\Omega(N)$ *paths, the new algorithm will only have to try to find* $\Omega(1)$ *paths* [2].

Notice how this Lemma specifies the amount of *output* we request from the search algorithm, not its runtime. Nevertheless, the fact that in the best case the algorithm only is requested to find one shortest path instead of plenty can be assumed to greatly reduce the efforts it has to invest.

2 While we hereby specified a *minimum*, i.e. best case, the worst case is however that the new algorithm runs the SPSP algorithm N times - once for *every* Identity! The probability of this worst case and possible solutions to avoid it will be discussed in the chapters about measurements and possible future work, see pages 42 and 54.

3.2.2 Updating Score capacities

Once changed ranks are determined, the new incremental Score computation algorithm will proceed to update capacities. The input dataset to the function which provides that is the set of changed ranks, as capacities are computed from ranks.

Listing 10: New optimized incremental Score computation algorithm: updateCapacitiesAfterDistrust...() (simplified) [42]

```
1  Map<ScoreID, ChangeSet<Score>> updateCapacitiesAfterDistrustWithoutCommit(
2          Collection<ChangeSet<Score>> rankChanged) {
3
4      Map<ScoreID, ChangeSet<Score>> capacityChanged = new HashMap<>();
5
6      for(ChangeSet<Score> changeSet : rankChanged) {
7          Score score = changeSet.afterChange;
8          if(score == null) {
9              // Existing Score deleted -> Rank deleted -> Capacity deleted
10             capacityChanged.put(changeSet.beforeChange.id, changeSet);
11             continue;
12         }
13
14         // Same function as on page 15
15         score.capacity
16             = computeCapacity(score.truster, score.trustee, score.rank);
17
18         capacityChanged.put(score.id, changeSet);
19     }
20
21     return capacityChanged;
22 }
```

The capacity computation part of the new incremental Score computation algorithm is not worth much attention: As we have seen on page 15, computing capacities is merely a mapping of a single independent scalar to a gauge unit.
As the new rank computation code has produced a flat *Collection* of changed ranks, and capacities are are mapping of ranks, their computation is a simple iteration over the Collection.

3.2.3 Updating Score values

After changed ranks and capacities are calculated, the new incremental Score computation algorithm will finish by updating Score values. Score values are a sum of Trust values weighted by Score capacities. So the input to the function for updating Score values is which Trust value changed (represented by passing the Identity whose Trust changed to distrust) and which capacities have been changed.

Listing 11: New optimized incremental Score computation algorithm: updateValuesAfterDistrust...() (simplified) [43]

```
1  Set<ScoreID> updateValuesAfterDistrustWithoutCommit(Identity distrusted,
2          Map<ScoreID, ChangeSet<Score>> capacityChanged) {
3
4      Set<ScoreID> valueChanged = new HashSet<>();
5
6      // Normally, we might have to check whether a new Score has to be created
7      // due to the changed trust value
8      // - but updateRanksAfterDistrustWithoutCommit() did this already.
9      for(Score score : getScores(distrusted)) {
10         score.value = computeValueFromExistingData(score.truster, distrusted);
11         valueChanged.add(score.getID());
12     }
13
14     for(ChangeSet<Score> changeSet : capacityChanged.values())
15         Score scoreWithUpdatedCapacity = changeSet.afterChange != null ?
16             changeSet.afterChange : changeSet.beforeChange;
17         OwnIdentity treeOwner = scoreWithUpdatedCapacity.truster;
18         Identity trustGiver = scoreWithUpdatedCapacity.trustee;
19
20         for(Trust givenTrust : getGivenTrusts(trustGiver)) {
21             Identity trustee = givenTrust.trustee;
22             ScoreID scoreID = new ScoreID(treeOwner, trustee);
23
24             if(!valueChanged.add(scoreID))
25                 continue; // The Score was processed already.
26
27             Score score = getScore(treeOwner, trustee);
28             if(score == null) {
29                 // No need to create it: updateRanksAfterDistrustWithoutCommit()
30                 // has already created all scores which could be created.
31                 continue;
32             }
33
34             // Same as on page 20
35             score.value = computeValueFromExistingData(treeOwner, trustee);
36         }
37     }
38 }
```

A Score value of an Identity from the perspective of an OwnIdentity is the sum of all Trust values the Identity has received, multiplied by the capacity weight factor each Trust giver has received from the perspective of the OwnIdentity:

$$value(o,i) = \sum_{t.trustee=i} t.value * capacity(o, t.truster)/100$$

This sum can change if any of the addends changes. Thus, to determine the cases in which the Score value has changed, we use two loops to walk each of the two possibly changed factors in the addends:

1. The t.value factor changes for Scores for which an included Trust value has changed. In our case this is Scores which the distrusted Identity has has received since the only changed Trust value is the one of that Identity.

   Remember: This is because *updateScoresAfterDistrustWithoutCommit()* is to be triggered by *updateScoresWithoutCommit(Trust oldTrust, Trust newTrust)*, which is called when a single Trust value has changed, and the distrusted Identity is the receiver of that value.

   The loop at line 9 deals with this case.

2. The capacity weight factor changes for Scores in which a Trust value is included for which the capacity of the giver of the Trust value has changed.

   Thus, if the capacity of an Identity changed, we need to check for each of its given Trusts whether a Score exists which includes it. As the function receives a set of changed capacities, we must walk the given Trusts of the Identities in the set.

   This is what the loop at line 14 does.

PERFORMANCE There are two aspects to guarantee the performance of this algorithm:

- Similar to the advantages of the new rank computation over the old one which we had described on page 29, instead of blindly recomputing all Score values, this algorithm only recomputes Score values which may have changed. This is done by walking the variable addends of the Score value sums as we had just described.

  This yields:

  Lemma 2 *Let* $N = |\mathtt{Identities}|$ *and let* $|\mathtt{OwnIdentities}| = \mathtt{const}$.

  In the case where the old algorithm had to compute $\Omega(N)$ *Score values, the new algorithm will only compute* $\Omega(1)$.

- By keeping a Set which tracks which Score values were recomputed already at line 4, the algorithm ensures that each possibly changed Score value is not computed more than once.

  This is necessary because the Trust values of multiple Identities are used to determine the Score values of which Identities to update; and the Trusts of multiple distinct Identities might point to the same trustee Identities.

3.2.4 Choice and optimization of SPSP algorithm to fit structural properties of WoT graph

As we have noticed during the discussion of the new incremental Score computation algorithm *updateScoresAfterDistrustWithoutCommit(Identity distrusted)*, its entry-point is the recomputation of ranks using the function *computeRankFromScratch(OwnIdentity source, Identity target)*. Using this function, the Score computation algorithm will recompute the rank of the distrusted Identity. If its rank changed, then it will proceed to recompute the ranks which that Identity may have given to other Identities, and continue with recomputing their given ranks, etc.
We shall now have a look at how *computeRankFromScratch()* works.

3.2.4.1 First development iteration: computeRankFromScratch() using the "uniform-cost search" algorithm

For ease of understanding, let us begin with considering the first *unoptimized* version of *computeRankFromScratch()*.

Listing 12: New optimized incremental Score computation algorithm: computeRankFromScratch() initial version (simplified) [44]

```
1  int computeRankFromScratch(OwnIdentity source, Identity target) {
2      // OwnIdentity rank decision must override all other paths.
3      Trust directSourceTrust = getTrust(source, target);
4      if(directSourceTrust != null)
5          return directSourceTrust.value > 0 ? 1 : ∞;
6
7      class Vertex implements Comparable<Vertex> {
8          Identity identity;
9          Integer  rank;
10
11         int compareTo(Vertex o) { return rank.compareTo(o.rank); }
12     }
13
14     PriorityQueue<Vertex> queue = new PriorityQueue<Vertex>();
15     Set<Identity> seen = new HashSet<>();
16
17     seen.add(source);
18     for(Trust sourceTrust : getGivenTrusts(source)) {
19         int rank = sourceTrust.value > 0 ? 1 : ∞;
20
21         // Initialize queue
22         queue.add(new Vertex(sourceTrust.trustee, rank));
23
24         // OwnIdentity rank decision must override all other paths.
25         // We prevent the Identity from being able to receive a rank
26         // from others by marking it as seen even though it wasn't yet.
27         seen.add(trustee);
28     }
29
30     while(!queue.isEmpty()) {
31         Vertex vertex = queue.removeMinimum();
32
33         if(vertex.identity == target)
```

```
34              return vertex.rank;
35
36          // Identity has reached maximum rank and thus may not give a
37          // rank to its trustees -> No need to look at them.
38          if(vertex.rank == ∞)
39              continue;
40
41          seen.add(vertex.identity);
42
43          for(Trust trust : getGivenTrusts(vertex.identity)) {
44              Identity neighbour = trust.trustee;
45
46              if(seen.contains(neighbour))
47                  continue;
48
49              int neighbourRank = trust.value > 0 ? vertex.rank + 1 : ∞;
50              Vertex neighourVertex = new Vertex(neighbour, neighbourRank);
51
52              queue.decreaseKey(neighbourVertex)
53          }
54      }
55
56      return -1; // Not path found -> No rank.
57  }
```

As we remember that the purpose of *computeRankFromScratch()* is to find a solution to the "single-pair shortest-path problem" (SPSP) [10], we notice that the algorithm is very similar to the familiar Dijkstra's algorithm [31] which solves the "single-source shortest-paths problem" (SSSP) [11]:
Same as Dijkstra, it inserts the vertices (= Identities) in a PriorityQueue where the priority is the current known shortest path length to a vertex. Also like Dijkstra, it walks the PriorityQueue by extracting the vertex with the minimal distance, and then explores its neighbors by decreasing the priority of them to the possibly shorter path induced by the edges to the neighbors.
Other than Dijkstra, it aborts once the target vertex is found, which can without proof be imagined as the natural way of transforming a SSSP algorithm to a SPSP one [10].
However, there is one outstandingly significant difference to the Dijkstra algorithm which is the reason why the algorithm which inspired *computeRankFromScratch()* has its own name of "uniform-cost search algorithm" [12] [13]:
Instead of initializing the PriorityQueue with the set of *all* vertices, only a few [3] vertices are added first. The majority of the vertices are then added *on-demand as they are discovered* by walking edges.

PERFORMANCE The *gradual* enqueuing of vertices, in our case Identities, can be believed to be a very suitable choice for the use case of WoT: As we learned on

3 We hereby notice the most prominent difference to the UCS-algorithm: At line 18, instead of initializing the queue by just adding the source OwnIdentity, the first iteration of the queue processing loop was extracted into a separate loop. That loop processes the trustees of the source OwnIdentity in a different fashion than the main loop would do: It enforces the policy of the OwnIdentity's rank decision overriding remote rank decision - by marking the trustees of the OwnIdentity as seen before even having actually processed them, their ranks can be made read-only in the PriorityQueue.

page 8, one of the main jobs of WoT is discovering Identities by accumulating yet unknown Identities from Trust lists of already known ones. And by having a closer look, we may notice that it will discover *almost all worldwide existing* Identities: If we assume human interaction to be mostly peaceful, then most Trust values are positive, and hence are eligible for discovery of their target Identities. If we further assume the "social graph" of Identities to be well-connected by Trust values, as for example the "six degrees of separation theory" [34] may indicate, then the graph will not be partitioned an in fact most worldwide users will be reachable by Trust steps.
As a consequence, the WoT database of all Identities will be very large. If they were not enqueued gradually in the UCS-fashion, but all at once like with Dijkstra, then this would cause exhaustive memory usage. Further, as a PriorityQueue is a sorted queue, the runtime of the operations which yield the sorting would be non-negligible for large contents.
We thereby may recognize the UCS-algorithm with its on-demand exploration as a natural fit for the *large-graph* SPSP problem of WoT.

PERFORMANCE PITFALL The algorithm was in fact first implemented as just described. Profiling of the whole of incremental Score computation using this implementation of *computeRankFromScratch()* unfortunately showed execution times which were so exorbitantly large that doing benchmarks which significant sample count would have taken days. The reason for this can be identified as follows:
If there *is no* path existing from source to target, the algorithm will have to walk *all* reachable vertices and edges to be able to conclude that no path exists. This is also described in literature [10] as the worst case of SPSP being as slow as the fastest way of solving the whole of SSSP, i.e. finding shortest paths to all reachable targets.
This problem is exacerbated by our usage pattern of *computeRankFromScratch()*:
As shown on pages 23 to 27, we use it in *updateScoresAfterDistrust...()* to find a new rank for Identities which have been *distrusted* by removal of a single Trust value. If an Identity is not only distrusted by removal of the single Trust value which caused this function call, but also by the fact that it has received no other Trusts, then there will *not be a reachable rank path*.
And one might assume that having received one distrust indicates a high probability of not receiving any other positive Trust: Distrust is intended to be used only for combating spam (page 9), and it seems probably that someone is either a spammer or not, and thus having received one distrust is a good indicator for not receiving any non-distrust at all.
Overall, then the probable situation at our usage of *computeRankFromScratch()* is the case where the target Identity is fully distrusted, and thus no rank path exists. So we will usually run into the worst case of walking the whole graph of all Identities and Trusts!

To alleviate the impact of this, an optimization of the way UCS is used was invented and shall be discussed in the following section [4]

4 In a personal note, it shall be stated that the following optimization might be considered the most interesting idea of this thesis, and thus reading it is strongly recommended.

3.2.4.2 Second development iteration: computeRankFromScratch() using optimized UCS algorithm

With a slight modification compared to the code we just discussed, the following algorithm shows improved behavior with regards to fully distrusted Identities.

Listing 13: New optimized incremental Score computation algorithm: computeRankFromScratch() optimized version (simplified) [45]

```
int computeRankFromScratch(OwnIdentity source, Identity target) {
    class Vertex implements Comparable<Vertex> {
        Identity identity;
        Integer  rank;

        int compareTo(Vertex o) { return rank.compareTo(o.rank); }
    }

    PriorityQueue<Vertex> queue = new PriorityQueue<>();
    Set<Identity> seen = new HashSet<>();

    seen.add(target);
    for(Trust targetTrust : getReceivedTrusts(target)) {
        int rank = targetTrust.value > 0 ? 1 : ∞;

        // If a Trust exists from the OwnIdentity source to the target,
        // then it must always override any other remote rank paths.
        if(targetTrust.truster == source)
            return rank;

        queue.add(new Vertex(targetTrust.truster, rank));
    }

    while(!queue.isEmpty()) {
        Vertex vertex = queue.removeMinimum();

        if(vertex.identity == source)
            return vertex.rank;

        seen.add(vertex.identity);

        Trust trustFromSource = getTrust(source, vertex.identity);
        if(trustFromSource != null) {
            // Again OwnIdentity rank decision always wins

            if(trustFromSource.value > 0) {
                int rank = vertex.rank + 1;
                queue.decreaseKey(new Vertex(source, rank));
            } else {
                // An Identity with a rank of ∞ may not give its
                // rank to its trustees. So the only case where the rank of an
                // Identity can be ∞ is when it is the last in the
                // chain of Trust steps.
                // By adding the received Trusts of the search target to the
                // queue before starting to process the queue, we already
                // consumed the last links of the chain. Here we can only be
                // at last + 1, last + 2, etc. So at this point, a rank of
                // ∞ cannot be given because it would be in
                // the middle of the chain, not at the end.

                /* queue.decreaseKey(new Vertex(source, ∞)); */
```

```
52              }
53
54              continue;
55          }
56
57          for(Trust trust : getReceivedTrusts(vertex.identity)) {
58              Identity neighbourVertex = trust.getTruster();
59
60              if(seen.contains(neighbourVertex))
61                  continue;
62
63              if(trust.getValue() > 0) {
64                  int rank = vertex.rank + 1;
65                  queue.decreaseKey(new Vertex(neighbourVertex, rank));
66              } else {
67                  // Same as above
68                  /* queue.decreaseKey(new Vertex(neighbourVertex, ∞)); */
69              }
70          }
71      }
72
73      return -1; // Not path found -> No rank.
74  }
```

We compare the initialization of the PriorityQueue at line 13 with the similar code of the previous implementation of the algorithm (page 33 line 18): Where the old implementation initialized the queue with the given Trusts of the source OwnIdentity, the new one will begin with the received Trusts of the target Identity.
Hereby we notice that the primary change of the algorithm is a mere *reversal of the search direction*: It searches from target to source instead of from source to target.

PERFORMANCE One might wonder why reversing the search direction is claimed to improve the performance of the algorithm - SPSP sounds like a symmetrical problem. But it is not with regards to the dataset of WoT: Distrust of an Identity "blocks" edges related to it. A distrusted Identity, which is one with no rank or a rank of ∞ (see pages starting at 13), either

- has not received any Trust, and thus may not receive a rank. The edges of received Trusts to it are "blocked" by not even existing.
- has only received negative Trusts, thereby has a rank of ∞, and may not give a rank to its trustees as the rank of ∞ is not inheritable. The edges of given trusts going outside from it are "blocked".

This behavior of blocked edges causes less work for the algorithm since it does not have to walk along them. And with regards to a *fully* distrusted Identity, the graph will decompose into two disconnected sub-graphs where the distrusted Identity and its trustees are one of them, and the trusted Identities constitute the other sub-graph.
Imagine this as society splitting into two social networks where one is "good" and the other is "evil". They are disconnected as "good" does not trust "evil".
Let us remember again that the problem we are trying to optimize is the case where no path between source and target exists - the old algorithm would then have to

exhaustively search all possible edges in the graph, which took a long time. This is the problem of a graph split into "good" / "evil" which we just imagined. The central reason for a performance improvement by walking backwards from "evil" to "good" now can be discovered in the following assertion:

Lemma 3 *There will be a lot less "evil", distrusted Identities in a typical WoT database than "good", trusted ones.*

Thus, if the algorithm starts searching at the "evil" side, it will have to walk a much smaller sub-graph until it runs into a wall of blocked edges and can realize that no path exists.
Luckily, we may acknowledge that the assertion of "there are more good than evil Identities" is not axiomatic, i.e. not an optimist's good faith of "good always wins over evil", but a provable logical conclusion if we remember the core goal of WoT Score computation as explained on pages starting at 10: To prevent client applications *and* WoT from downloading content published by spammers. So Identities with a bad Score will not be downloaded. Hence, when WoT does not download the content of a distrusted Identity as it may be spam, this *includes* the Trust values given by the Identity - they will also not be downloaded. Identities only being reachable through Trust values of distrusted Identities will thus not be discovered by WoT, they stay in the "darkness" of the "evil" subgraph. So the subgraph of the "evil" Identities is sparsely populated - it only includes as many Trusts and Identities as were downloaded before the Trusts of other Identities stigmatized the "evil" ones as such. The subgraph of the "good" Identities instead will be downloaded in its complete vastness.
Overall, we can thus assume that in the worst case of no Trust path existing from source to target, the new algorithm which walks backwards does have a much smaller subgraph to explore before it can conclude that no path exists. This improves the runtime of the worst case as desired.

ADDITIONAL PERFORMANCE BENEFIT As the comment on line 40 explains, there is another minor performance benefit from walking the Trust graph backwards:
The rank value of ∞ is not allowed to be inherited, and so when walking backwards we may not walk across it. I.e. we must stop walking rank chains upwards if we encounter it. In other words, ∞ can only happen as a *leaf* rank.
This observation is used to reduce the size of the PriorityQueue: The leaf vertices of the queue were already processed outside of the main loop, in a separate setup loop at line 13. Thus, the main loop cannot encounter leafs and does not need to add ∞ to the queue.
This was not the case when walking forwards (page 33): There, the valid ranks of ∞ could not be guessed at the initialization of the queue since the initialization did not start at the leafs, and so all ranks of ∞ had to be added during the main loop.

3.2.5 Synopsis of new incremental Score computation

Our discussion of the new optimized incremental Score computation algorithm is now finished. Before we proceed to measuring its performance in practice, we shall prepare to decide how to measure it by recollecting what we have learned in this chapter so far:

- When a single Trust value changes or is deleted, WoT will trigger an incremental Score computation algorithm to update *Score.rank*, *Score.capacity* and *Score.value* of Scores which are affected by the Trust (page 19).
- Before the thesis, there already was an incremental Score computation algorithm (page 19). It was unable to handle the case where a Trust changed to "distrust", i.e. when the Trust was removed or changed to a negative value. It did fall back to recomputing *all* Scores from scratch (page 23).
- The thesis fulfilled the goal of inventing a fill-in incremental Score computation algorithm which handles the aforementioned case of Trust changing to distrust: *updateScoresAfterDistrustWithoutCommit(Identity distrusted)* (page 25).
- The algorithm for handling distrust incrementally is powered by recomputing Score attributes which could be touched by the distrust:
  - The rank of the distrusted Identity (page 27), as the rank may have been received through the Trust edge which was removed. The rank is recomputed by a standard "single-pair shortest path" graph algorithm which was specially adapted to work well in consideration of structural properties of the WoT data ("less evil than good Identities").
  - If the rank of the distrusted Identity changed, the ranks of Identities which have received Trusts from it may also change and because of that are recomputed (page 27). If their ranks do also change, the ranks of Identities trusted by those will be recomputed next, and so forth. Ranks are inherited, so changed ranks have to be tricked down.
  - The capacities of all Identities for which the rank changed (page 30), as capacity is computed from rank.
  - The Score values of all Scores for which the Trust value of an included Trust changed; or for which the capacity which weights a Trust changed (page 31). The amount of changed Trust values is only 1 here - the changed Trust which triggered incremental Score computation. The amount of changed capacities is those which were induced by changed ranks as said above.

From the overview we have just had, we can realize that the amount of iterations which the new distrust computation algorithm has to execute is dominated by the amount of changed ranks: The count of changed ranks defines count of changed capacities, and the count of changed capacities dominates the number of Score values it has to recompute.

Thus, for creating a conclusive measurement, we will aim for triggering a "stressful" situation for the new algorithm by doing something which may induce rank changes.

3.3 Benchmark

3.3.1 Choice of benchmark

We will now investigate multiple different approaches of measuring the performance improvement of the new algorithm, and eventually shall decide to use one of them.

3.3.1.1 Mathematical analysis

Without any doubt, it can be agreed that a mathematical proof of the asymptotic runtime in Landau notation is the most solid evidence of success which could be given to the new algorithm.
Unfortunately, with regards to the specific situation of Freenet, and the time constraints of a bachelor's thesis which only allow a single type of benchmark, it does not seem to be the optimal strategy to use a mathematical proof as the only result. This is because the *software engineering* aspect of ensuring *long-term maintainability* of the software would be violated:
Freenet is a mostly volunteer-driven project. During the thesis period, when for example having a look at the team chat, there were usually about 80 people in the chat room - of which only 2 are employed by the Freenet foundation. Furthermore, the foundation currently only has 2 months of donation funding left [19] to pay those employees.
Years of personal experience with volunteer-driven projects show that volunteers prefer to do "fun stuff", where "fun" usually means writing new code. Tedious maintenance tasks such as writing documentation are very often ignored - there is no boss to force volunteers to commit to such aspects. And the complexity of mathematical proofs is far beyond writing standard documentation.
Because of that, there is a high probability that a mathematical proof would only be valid for as long as no volunteer decided to continue improving the algorithm without adapting the proof.
In opposite to that, a benchmark *software* will continue to yield comparable results as long as it treats the Score computation code as a black box. By merely measuring the execution time of high-level public API [5] functions, the benchmark still can compile against evolved code if only the internal algorithms which power the API changed. With this approach, volunteers may recycle the benchmark provided by the thesis.
Another benefit of providing a benchmark framework is fostering volunteer work: Even without mathematical proofs, performance optimizations as the one conducted by this thesis can be considered as non-"fun stuff": The software does still work without them, it is just slower. Doing such work does not satisfy the volunteer's desire of *creating* something new, and seeing results *quickly* - volunteers may likely defer such work work. Providing a way for volunteers to immediately see

5 Application Programming Interface

the results of their work through an easy-to-use benchmark may hence motivate them to do tackle complex performance work.
While considering these benefits, and since the thesis timespan only was sufficient for one performance metric, it was decided to implement a future-proof software benchmark instead of a mathematical proof for both the old and new algorithm.
However, for considerations of possible future work, the conclusions at page 51 will at least have a brief look at the worst case runtime of the new algorithm.
Several types of software benchmarks were implemented, and we may now study them to be able to decide which one will be the best to fully conduct.

3.3.1.2 Real world benchmark

As promised in the agreement of the thesis' goals, a benchmark was implemented to measure the total time of *Identity file import* of all Identity files which were processed in an execution of WoT [46]. Please first consult page 8 if you need to refresh your knowledge of what an Identity file is.
As suggested by F. Daignière for performance reasons explained at page 63, the code which parses Identity files was changed from direct parsing in memory to serialize them to a disk-based queue [47]. In an own decision, it was made able to *archive* them after processing [48]. The archive will both contain the files themselves as well as the *order* in which they were processed. This can be considered as a *full dump of the real WoT network*.
It was made possible to repeat a sequence of Identity file imports in a deterministic way [49]; and to measure the time it takes for all involved computations (by the aforementioned total import time). This would allow measuring the total time it takes to bootstrap a WoT database with an import of the full current dataset of the real network's Identities, Trusts and Scores. It would also allow to repeat the measurement with both the old and the new Score computation algorithm for comparison.
There are further advantages to Identity file queuing, including a performance improvement, which will be discussed on pages 63 and 64 .
Both unfortunately and fortunately, these measurements of total Identity file computation time did not yield statistical significance with regards to quantifying performance of the new *updateScoresAfterDistrust...()*: The WoT community is too "peaceful" currently, there were only a few hundred distrust operations happening when importing a total dataset of over 200 000 Trusts. Not only is less than a thousand samples not of statistical significance - the speed difference between the old and new algorithm would also drown in the much greater total execution time of more than 200 000 non-distrust operations whose execution time did not change.
Thus the plan to use total import time of real world Identity files for benchmark had to be scrapped in favor of a synthetic benchmark which provokes the situations which we want to investigate.

VALIDITY OF THESIS GOAL IN THE LIGHT OF UNAFFECTED TOTAL COMPUTATION TIME The careful reader might question the importance of the thesis' overall goal of optimizing distrust computation now that it has become apparent that its poor performance did not affect total computation time a lot. It can be believed that the importance is nevertheless given:
The analyzed Identity file network dumps were a snapshot of the network at a finitely short timespan of less than a day. They thus do not represent the temporal development of Trust values over weeks or months. Identities might actually change them from positive to negative and vice versa a lot.
Further, an attacker might intentionally constantly flip the signum of Trust values to trigger the expensive distrust computations.
And most importantly, the execution times of the old distrust computation code of close to a *minute* (which the following measurements will show) would then block the UI of WoT for the whole of that time due to a difficult to fix issue in the way the database is used (see [4]). It is a great usability issue if a user interface becomes unresponsive for such a long time; especially if attackers may trigger this as was just described: To distrust attackers to prevent them from conducting this attack, it is necessary to be able to access the UI.

3.3.1.3 Synthetic benchmark

Since the real world benchmark was unable to provide a sufficient frequency of *updateScoresAfterDistrust...()* events, the next logical step in determining a way to benchmark this function was an artificial benchmark which is purposefully crafted to:

- cause the function to be called a lot.
- produce a random Identity and Trust graph of proper dimensions to yield an input size both sufficiently large for statistical significance, and sufficiently small to not cause excessive runtime of the benchmark.
- allow free choice of the size of the generated graph for tweaking the runtime to be appropriate for what is measured.

The first implementation of this [50] was trivial: All which had to be done is using the pre-existing unit test framework of WoT. It already provided functions for producing completely random graphs of Identities and Trusts. This code then only had to be amended to remove randomly chosen Trust values.
But after looking at the way the random graphs are generated, the scientific validity of this benchmark turned out to be questionable: The topology of the real WoT network's social graph is not studied yet, and thus was not simulated in the existing random Trust graph generation code. The Trust edges were chosen to hold random Trust values of an arbitrary variation of Gaussian-distribution; and the random pairs of Identity vertices which constituted them were uniformly distributed. It is highly doubtful whether such a random graph is an equitable model of the

complexity of social interaction between real users. It does seem necessary to simulate a real topology as the question whether an algorithm is adequate for a certain type of graph highly depends on the graph's structural properties.
An attempt was made to bend the random graph model to a more natural one:

1. A tool was written to produce a histogram of the number of occurrences of each Trust value in the allowed range of $[-100, 100]$ [51]. Such a histogram was generated for the existing full network dump from the previous attempt of doing a real world benchmark. It included 200 000 Trust values, and thus could be guessed to be significant enough. The code which generates the random Trust graph was amended to chose Trust values to fit the distribution shown by the histogram.

2. Inspired by a suggestion of A. Babenhauserheide in the Freenet team chat, and similar to the previous histogram, code was written to generate a histogram of the outgoing Trust *degree* values of Identity vertexes [51]. For example an Identity has an outgoing Trust degree of 5 if it gave five Trust values to other Identities. The histogram counted the number of occurrences of each degree value. Again, the random Trust graph generation code was amended to obey the distribution which this histogram had measured.

3. After the outgoing Trust degree histogram was available, it became apparent that A. Babenhauserheide was right: The distribution of incoming Trust degree would also have to be measured and respected in random Trust graph setup. It was decided to abort the implementation here for reasons explained below.

At the point of having followed the two random distribution histograms, and having decided that the code had to be amended to model a third, the random graph setup code filled several screens already [52].
It would have taken a non-negligible amount of time to complete the code. And then it would not yet be scientifically justified to be a valid, complete model of the real network.
As the time for completing the thesis was running out already, it was decided to continue with a less synthetic approach.
Nevertheless, the framework for random graph generation was preserved and merged into the official repository. It may serve as a useful foundation for future work to implement fully synthetic benchmarks. These will have a huge benefit over real world benchmarks: In difference to real network dumps, the size of their graphs can be chosen freely by redefining constants. By this, developers can chose between:

- Short runtime of benchmarks, which can be a useful tool during development to quickly get feedback upon whether changes are an improvement.

- Long runtime of benchmarks, which is important for getting accurate results to evaluate performance with significant confidence.

3.3.1.4 Semi-synthetic benchmark

While the real world benchmark described on page 43 did not cause a sufficient number of distrust operations to yield a satisfactory measurement, it did still yield a large dump of the real network:

- Discovered and downloaded Identities: 11 985.

  Notice: Number of total Freenet users is ~10 000, see [7].

- Discovered but not downloaded[6]: 183, which is ~1.5 % of the discovered ones.

- Trusts: 222 122

- Own Identities: 1 [7]

Remember: We want to measure the performance of the improved incremental Score computation algorithm. The primary input to that algorithm is the graph where Identities are vertexes and Trusts are edges.
For that purpose, over 10 000 vertexes and 220 000 edges may in our evaluations be considered as a strongly significant dataset. Additionally, the number of Identities being in the order of magnitude of the total amount of Freenet users and the only 1.5% of not downloadable Identities both indicate that the dump is highly complete. Consequently, it was decided to use this data set as a source data set for a semi-synthetic benchmark: The data set would be real, the operations whose execution time is to be measured would be chosen in a synthetic way.
We will now continue with choosing which the benchmarked operations will be. Before we do so, it may be recommended to re-read the pages starting at 40. They give a summary of the new algorithm and hint at which aspects of it should be measured.
We will measure the performance of the new algorithm by measuring frequent execution of an operation which was replaced by the new algorithm - once with the new algorithm, once with the old. We now need to find such a function.
The function needs to meet several requirements:

1. It has to exist both in the old and in the new code base so the benchmark can measure a performance improvement factor.

2. It always has to always cause new incremental Score computation to happen as that is what we want to measure.

6 As described on page 1, Freenet will naturally drop data if it is not downloaded by anyone for some time. Some unpopular Identities will thus not be downloadable.

7 This had to be manually chosen as own Identities are user-created. The value of 1 can be considered as a natural choice: Freenet is a peer-to-peer network, and thus each user should install it on their own instead of using a central service which runs Freenet. Thus, a single instance of WoT will only have one actual user. The amount of own Identities will only grow due to the user's privacy concerns. In laboratory conditions such as ours, a natural choice for such a small value is 1. One might argue that a value of 2 at least should be tried to provoke errors which do not happen upon boundaries such as 1, but the primary purpose of this dataset is benchmarks. Correctness was tested using a different one which included more than 1 own Identity.

3. As we learned from the real world benchmark, execution time needs to be dominated by the execution time of the old / new algorithm, i.e. not include any other expensive calculations.

4. The Synopsis of new incremental Score computation in the previous section (page 40) has indicated that a benchmark for the new algorithm shall have the goal of possibly causing rank changes, as rank changes are what increases the iterations of the algorithm.

To fulfill these requirements, it was decided to use the *deletion of an existing Trust by removeTrust()*. The requirements are met as:

1. Removal of a Trust is a core operation of WoT and has always been implemented.

2. As explained on pages starting at 23, *removeTrust()* will always cause the new incremental Score computation algorithm to run.

3. By studying the pseudocodes (also at the said pages) which determine whether the old / new algorithm will run, one may also learn that it takes $\mathcal{O}(1)$ in both the old and the new code to decide this. It only has to look at the deleted Trust, not query any further Identities / Trusts / Scores. Ergo, as constant offsets can be ignored in terms of Landau notation, the execution time of *removeTrust()* can be seen to boil down to be dominated by the execution time of the old / new algorithm which we want to benchmark. Thus *removeTrust()* can be the function whose execution time we measure.

4. Rank values are inherited through Trust values (pages starting at 12), which means that removing a Trust value causes the trustee to not be able to inherit a rank value through the removed Trust anymore. This satisfies our requirement of possibly causing changed ranks in the benchmarked operation.

3.3.2 Benchmark results

The benchmark operated as shown by the following simplified pseudocode:

Listing 14: Benchmark to compare old and new algorithm (simplified) [53] [54]

```
Queue<Trust> trusts = randomPermutation(getAllTrusts());
int trustCount = trusts.size();

while(!trusts.isEmpty()) {
    Trust trust = trusts.removeFirst();

    System.gc();    // Try to exclude garbage-collection peaks

    StopWatch result = new StopWatch();
    removeTrust(trust);
    result.stop();

    int  x = trustCount;
    Time y = result.seconds();
    graph.plot(x, y);

    trustCount--;
}
```

Random Trust edges are removed, the time for reach removal is measured, and plotted as y-value with the x-value being the total number of Trusts in the database. The benchmark was executed once with the old Score computation code, once with the new one. Both times, the same fresh copy of the initial data set was used as input.

No user was present at the machine [8] during the time of the benchmark, and it was disconnected from the Internet to prevent disturbance due to periodical jobs such as automated software upgrades.

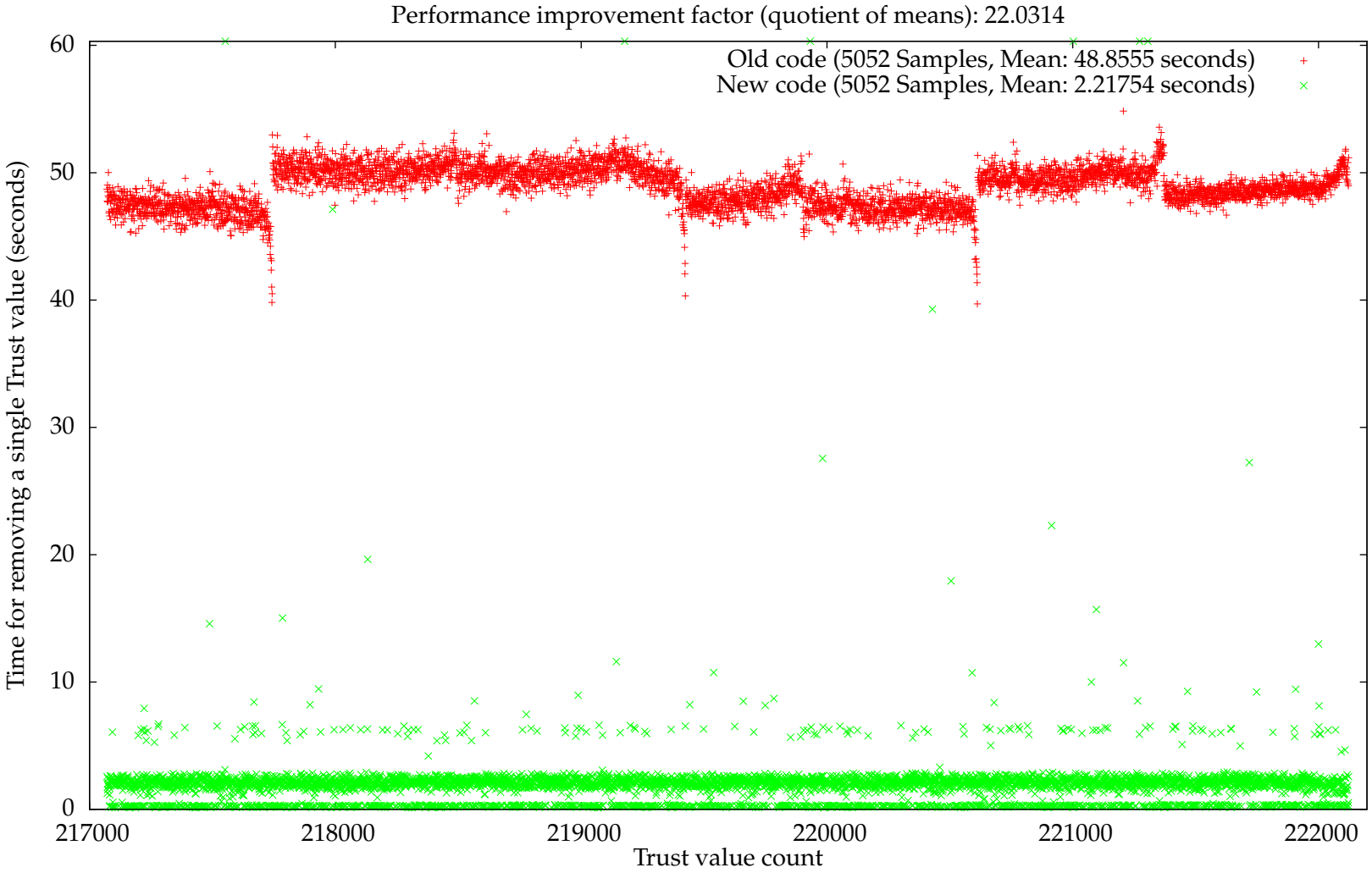
Due to time constraints of the thesis, the benchmark was aborted at an execution time of 68 hours with the old code. While the resulting amount of ~5000 iterations of *removeTrust()* could be argued to only be slightly above the lower boundary of statistical significance, it has to be accredited that blocking the author's workstation for 9 hours a day for a week might be enough of effort for the time constraints of a bachelor's thesis.

The same number of ~5000 iterations was then repeated with the new code, which took it a total of 3 hours only. It was chosen to not use the same number of hours as with the old code as a similar iteration count is of more interest: It allows a plot which shows the same amount of measurements for the old and new code. Furthermore, the numbers at 5000 samples were already steady enough to allow the conclusion that further samples would not reveal much more information.

For readability of the graph, a cutoff was set above ~60 seconds. Values above this cutoff are rendered on the top frame of the graph.

8 ThinkPad T61p, 2.4 GHz Core 2 Duo processor, 8 GiB of memory, 1 GiB limit for WoT. The specific machine is not of much relevance as our primary interest is relative speed improvement; but it may help to understand how bad the performance was for regular users.

Figure 3.1: Benchmark results of optimized part of incremental Score computation algorithm [53] [54].
Uses a network dump of the real network - Identities: 11985 (Not fetched: 183), Trusts: 222122, Own Identities: 1.



4 Conclusion and Outlook

4.1 Analysis of benchmark results

As the aesthetics of the idea of reversing the UCS algorithm to gain speed already caused a certain elation, the performance improvement factor of 22 which we just measured may now induce euphoria.
But let us not be fooled: It shall be admitted that the huge performance improvement is probably not due to incredible smartness of the new algorithm, but rather due to the naïveté of the old algorithm. Remember: Upon removal of a Trust, the old algorithm would just "bail out" from having to incrementally recompute Scores by throwing away all Scores and recomputing all of them from scratch. Colloquially, this was more of a hack than a real algorithm. Therefore, even a bad fully incremental algorithm had an easy battle.
And when looking really closely, one might even ascertain that the new algorithm actually does the same mistake as the old algorithm, only on a smaller scale: Upon removal of Trust edges, its core function *computeRankFromScratch()* (cRFS) searches new shortest paths while completely ignoring the possibility of using old information about preexisting paths to accelerate the search.
This has wide reaching consequences as we will learn now.

4.1.1 Deficiencies of the new algorithm

CRFS DOMINATES THE RUNTIME OF THE NEW ALGORITHM During development, various manual measurements did in fact show that *computeRankFromScratch()* consumes the majority of the execution time of the new algorithm.
This is slightly visible in the benchmark plot as well: There are discrete "levels" of execution time, which are probably equal to whether cRFS is executed 1 time, 2 times, etc.

CRFS IS USED VERY FREQUENTLY If we reread the function which decides whether it needs to run the new incremental on algorithm on page 23, we may realize that the cases where it does so are precisely those with a high probably of a rank path having died. As ranks are inherited, batch searches of new ranks for neighbor Identities by calling cRFS multiple times are then very likely.

WORST-CASE RUNTIME OF CRFS USAGE IS QUADRATIC The worst case runtime of cRFS is the runtime of the SSSP problem as explained on page 35.
SSSP for WoT can be rather large: As we learned on page 34, WoT will try to discover all world-widely existing Identities.
And even worse than that, a single iteration of the the new incremental Score computation algorithm may use cRFS *multiple* times (page 27), and so may run into its

worst case multiple times. The worst case here is calling cRFS for all V Identities, so the resulting runtime is $\mathcal{O}(V * t(cRFS)...) = \mathcal{O}(V * t(SSSP)...)$.
The worst-case runtime of cRFS being convergent to SSSP means in our case being convergent to Dijkstra: cRFS is based on UCS, and UCS is based on Dijkstra.
This yields $t(SSSP) = \mathcal{O}(VlgV + E)$ [31].
Overall, we are thus at a worst-case runtime for the new incremental Score computation algorithm of $\mathcal{O}(V * t(SSSP)...) = \mathcal{O}(V^2...)$ which is unacceptable.
This is already visible in the benchmark results on page 50: While the average execution time for the new algorithm is ~2 seconds, there is a small amount of samples well above 60 seconds.

4.1.2 Probability of occurrence of deficiencies

While a quadratic worst-case runtime of an algorithm usually is a testimony of a design failure, it must still be investigated whether the worst case will actually happen in practical use of the algorithm.
And in fact, there are signs that in our case, the worst case has a low probability.
For runtime to become $\mathcal{O}(V^2)$, the two factors V need to be contributed by:

- For *every* passed pair of a source OwnIdentity and target Identity, the SPSP algorithm of cRFS needs to be unable to find a path between the source and the target; and the walked subgraph must contain all V Identities. Hence the "evil", distrusted part of the social network must be *all* Identities.

- For *every* Identity reachable by the single Trust edge which changed to distrust, the new algorithm needs to determine that its rank was invalidated since it had been inherited only through the removed Trust edge (page 27). This can be imagined to be the removal of a Trust edge towards a "king" identity, which was the only source of ranks in a hierarchy of slave-Identities below it. Not a single rank-route "around" the king must have existed.

The coherence between these two requirements is the situation where a huge "monarchy" subgraph is removed.
For this to happen in practice, a single Identity would have to be the single source of Trust for a large web of "sybils": Identities which do only receive Trust from it.
This should typically only happen in the situation of the "sybil attack" [55], where an attacker creates many Identities of his own and makes them give Trust to each other. Other than that, there is no reason for a majority of the WoT users to avoid giving Trust to a huge subset of the community, and instead only delegate it to a leader of the subset.
And luckily, it was a design goal of WoT to avoid the sybil attack: To create an Identity by receiving a Trust value from a remote Identity, WoT requires new users to solve a "Completely Automated Public Turing test to tell Computers and Humans Apart" (CAPTCHA) [9]. Thus the amount of sybil Identities has an upper bound

by the amount of human CAPTCHA-solving workforce an attacker is capable of investing.
The happening of the worst case in the benchmark can be explained by the fact that the Trust values which the benchmark removes are chosen at random; and that it keeps removing Trust values from the same graph instead of resetting it to contain all initial Trust values. Over time, this will cause well-established Identities to become a "king" as was described. Once the last edge to the king is removed, a subgraph breaks off from the main WoT. This is again believable to be unlikely in practice.

4.2 Conclusion

As we just saw, the worst case of the new algorithm running in $\mathcal{O}(V^2)$ was overall concluded to be improbable.
In addition, our benchmark (page 50) had showed an average execution time of ~2 seconds on an elder laptop, which from a user's point of view is a lot more bearable than the average execution time of ~49 seconds of the old algorithm.
From a developer's point of view, the average performance improvement factor of 22 is definitely beyond micro-optimization, and so may be considered as sufficient for a thesis project.
The thesis' work has therefore been merged into the main Git tree of WoT [20–25], and thus will definitely be included in the next WoT testing release. An extended testing period will be recommended, so testers can provide statistics on how frequently the worst case maybe does happen in practice. Logging code has been added to allow them to determine this.
It shall be admitted, that for security considerations, it is still desirable to prevent the worst case altogether by further algorithmic improvements.
At this thought, it can be hoped that the thesis will be completed to success by now proceeding to provide a variety of ideas to prevent the worst case. The author is eager to provide their implementation as future work.

4.3 Ideas for future work

We shall now have a look at some of the most interesting ideas for future work which the thesis yielded. Please notice that the following list is incomplete: There was also a fair amount of low-level "FIXME" and "TODO" comments added to the code, as well as bugtracker entries filed, which would be beyond scope to discuss here.

4.3.1 Opportunistic rank computation

A fundamental property of a shortest path in a graph is that sub-paths of it are also shortest paths.
Hence, when cRFS has computed a rank for Identity Z by discovering a shortest path from OwnIdentity O across identities A, B, C, ... Y, the ranks of A to Y have also been computed since the chosen paths across those Identities must also be shortest paths. Those shortest paths might be stored in a cache, which is kept for the duration of the incremental Score computation algorithm. It will use cRFS multiple times, and so the opportunistically computed paths might help.
On a grand scale, this can help to prevent the $\mathcal{O}(V^2)$ worst case of the new incremental Score computation algorithm which we just discussed (pages starting at 51): We again notice: The worst case of incremental Score computation arises from the worst case of cRFS not finding a path happening *each* of the worst-case V-times the outer algorithm calls cRFS. And the worst case of cRFS solves the SSSP problem when it happens the first time already. So as soon as that, we have discovered a shortest-path (of "no path exists") for *all* Identities! Thus, in the subsequent $V-1$ calls to cRFS which the worst-case of incremental computation will continue with, cRFS can determine the rank from its cache in $\mathcal{O}(1)$.

4.3.2 Backtracking

While we just saw that there are indeed viable options to speeding up cRFS, we could also instead just avoid its whole problems by trying to not call it whenever possible.
Without knowledge of the new algorithm, M. Toseland had suggested the approach of backtracking [38] in a broader sense. Albeit taken out of context here, this can be judged to be a relevant technique nevertheless: In a space-for-time trade, we might amend class Score to not only store the rank of an Identity, but also the Trust edge it has inherited the rank from. In the SPSP terminology, this would be the last edge of the shortest path to the target. When a Trust edge is removed, it could then be decided whether the rank of the target Identity really has to be recomputed using cRFS: If the removed Trust edge was not the provider of the rank, then the rank does not have to be recomputed.

This concept could be extended to storing not only the last edge in a rank-path, but all edges in the path. While the approach of caching only the last edge would likely only allow checking whether the first encountered rank has to be changed, this would likely even allow checking the validity of all visited ranks when looking at more than the first Identity which *updateRanksAfterDistrust...()* encounters (see page 27).

4.3.3 Divide and conquer

The primary user-facing problem with large Score computation times currently is not really the usage of system resources such as CPU, but rather the UI becoming unresponsive for times more than a second.
This is due to issues with the the way the database is used and likely cannot be fixed quickly, see [4] and the linked issues there.
The atomic unit of blocking the UI in this context is a single database transaction. To reduce the blocking, database transactions could be split up.
And in fact, our new incremental Score computation algorithm is suitable for that: The procedure of updating ranks (page 27), which defines the worst case runtime (page 51), is suitable for being split up in a divide and conquer fashion.
It processes a queue of Scores for which the rank needs to be recomputed, and currently consumes all of it in a single transaction. The queue could be kept in the database instead of in memory. A single database transaction would then consist of processing the single head element of the queue - instead of all of it.
Since the worst case $\mathcal{O}(V^2)$ of new incremental Score computation is a compound of two factors V, where one of them is a result of the Queue possibly containing V elements, this could cut down the transaction size from $\mathcal{O}(V^2)$ to $\mathcal{O}(V)$. This would be traded off against having $\mathcal{O}(V)$ transactions instead of 1.
A further benefit with the divide and conquer approach may be that what was just described is not the only place where it could be applied. There is more than one queue processing loop in the new Score computation algorithm, and all of them could be considered for being split into multiple transactions.

4.3.4 New class of shortest path algorithms?

If we forget about the technical details of the new algorithm for a while, and remember which problem it tries to solve at rank computation (page 13), we notice that WoT has a quite compact goal there:
It wants a data structure which contains the solution to the SSSP problem upon the graph of Identities and Trusts, and this database shall stay up to date upon variations of the graph in steps of one changed Trust edge at once.
It is tempting to name this problem to denominate a whole new class of graph algorithms, perhaps “Single-Source Variable Shortest-Paths” problem (SSVSP).

Due to the concise nature of this problem's definition, one may wonder whether WoT development is really the first manifestation of this goal in the history of algorithmics. It is quite imaginable that this is a fundamentally important problem in the domain of graph algorithms, and thus a whole class of algorithms might exist to solve it.
Hereby it shall be stated that the utmost interest of the author with regards to feedback from peer review is in this aspect. Information upon whether the computer science community has already dedicated a name and a catalog of solutions to this problem would be highly appreciated!

4.3.5 Different Score computation algorithm

As agreed when deciding the goals for this thesis, one goal was to *not* modify the concept behind Score computation, i.e. keep the specification of which output it should produce as is. Only the performance should be improved, the saneness of the ideas which constitute Score computation was not to be questioned.
It is technically possible that the desired resulting output of Score computation as specified by the reference implementation cannot be computed in an efficient way. If all efforts to find a fast algorithm to compute Scores turn out to be in vain, then a new Score computation scheme may need to be invented. For the purpose of gathering different ideas about Score assignment, we will now continue with looking at related works.

4.4 Related works

The concept of a "web of trust" predates the Freenet WoT. Therefore, we have a wide variety of trust calculation systems to chose from when considering related works.
Let us now have a look at only some of them.

4.4.1 Freenet Message System (FMS)

FMS [56] is a forum system built on top of Freenet. It existed before WoT, and may be considered as a predecessor which inspired the creation of WoT.
Architecturally, there is an unfortunate disadvantage of FMS: The trust system of its own which it includes is *not* exposed with an API which would allow it to be the foundation for other client applications. It will only serve FMS for its own purposes.
The specification of the trust metric of FMS [57] is rather thin. Recent discussion on its own forums [58] seems to clarify the way it works:
It also has trust values which are averaged. However, they are not weighted by a graph distance metric as in WoT. Instead the user assigns a special secondary trust value called the "trust list trust" which is the weight for the trust list of remote peers it is assigned to.
Most notably, this has a recursion depth of 1: For example, let the local user L assign a trust list trust T1 to remote user X. If X gives a trust value T2 to an identity Y, then the trust T2 is weighted by T1. If Y continues to give a trust T3 to another peer, that trust is ignored as the local user L did not assign a direct trust to Y.
An anonymous user in the aforementioned discussion labeled this as "selective moderator approach", which seems quite fitting.
This provokes a question which the author of the thesis also asked in the discussion: If trust values are only valid up to a depth of 1, then how does FMS ensure a high visibility of remote users? A new user will only assign a handful of trusts, and the remote users which the new user trusted will probably not assign trust to the full set of all global FMS users.
The further answer of another anonymous user can be paraphrased as: The discovery of users has no depth limit, they are collected from the trust lists of all users.
This behavior does sound very vulnerable to the sybil attack of creating many fake users because the trust values which distrust a user are ignored beyond depth 1, but the ones to create them are not. An example could be given as: An attacker creates a "root" identity R, which trusts sybils $S_1 ... S_n$ of his own. With R, he solves an introduction CAPTCHA to get a trust value of a remote user. He then publishes spam with R, and R gets distrusted. However, as trust is not propagated beyond depth 1, the negative trust upon R will not propagate across R's trust edge to his sybils $S_1 ... S_N$. Overall, the sybils will thus stay visible as identity discovery did propagate across the trust edge already.

Admittedly, due to the lack of a clear specification of how FMS works, this is still very speculative information. A review of its source code should likely yield an answer to these concerns, but was not undertaken.
Nevertheless, it can be stated that a trust depth of 1 is a quite low limit: It gives the user a lot of choice to ignore remote trust wars, but also requires them to perform more maintenance.

4.4.2 Less Crappy Web of Trust (LCWoT)

The choice of the name "Less Crappy Web of Trust" [59] can be hoped to be a testimony for the necessity to write this thesis: It was created as an alternate, compatible implementation of WoT with the primary goal of fixing the severe performance issues - which was also the motivation of the thesis.
It was chosen to fix the regular WoT in the thesis anyway: Personal conversation with the author determined that it is meant as a proof of concept only, not as a full replacement. It lacks critical features such as the ability to provide CAPTCHAs.
Inspection of the current source code showed that its performance fixes are unfortunately also not a solution to the incremental Score computation issues which this thesis addressed:
There is no incremental Score computation algorithm in LCWoT [60]. It just avoids finding a solution to this algorithmic problem by not updating Scores for every Trust change. Instead, a full Score recomputation happens every 3 hours [61].
While this does impose a limit upon CPU usage of LCWoT, it is of course more of procrastination than solving the actual problem.
Still, several things can be learned from LCWoT, which we may have a look at now.

4.4.2.1 Graph databases

While WoT uses the object database library "db4o"[1], LCWoT uses the *graph* database library "Neo4j" [62].
A graph database is specially suited for storing graphs, and hence may be of benefit with the graph algorithms of WoT / LCWoT.
And in fact, LCWoT already uses it to ease its calculations: Instead of solving SSSP manually for rank computation, it uses a built-in shortest path computation API of Neo4j [63].
While this API is of course limited to the same algorithmic possibilities which a custom implementation also is limited by, it may benefit a lot from the expertise of programmers whose sole job is writing graph algorithms for a graph database.
Also, it could be imagined that graph databases are maybe able to cache solutions to the SSSP problem, and update them incrementally as edges change, just as was suggested earlier on page 55.

1 As this product has been discontinued, there is no website to add to the bibliography. db4o was open source however, and development may thus be resumed by a different team.

The fact that the company behind db4o ceased its development is an indicator to consider changing the database which backs WoT, and a graph database should then definitely be taken into consideration.

4.4.2.2 Finite rank depth

Similar to what FMS does (page 57), LCWoT seems to limit rank computation to a finite depth [63].
While this seems to violate its goal of being a re-implementation of WoT, it is an interesting approach nevertheless: We have already learned that the "six degrees of separation theory" suggests that all humans are connected by 6 "trust" edges. This indicates that WoT's infinite rank computation depth very much exceeds the necessary depth to reach all Identities.
Additionally, it is questionable whether a user would even want a depth of 6: Consider real world friendships for example. If Alice trusts Bob as a friend, Bob trusts Charlie, Charlie trusts David, David trusts Elisabeth, Elisabeth trusts Frank and Frank trusts George - would Alice put any trust in George? Rather not.
Given the good reputation LCWoT seems to have in the Freenet community, it might be indicated to try lower depths of rank computation in WoT, and evaluate how they affect performance and visibility of remote users. Overall, it may then be wise to find a tradeoff between FMS' very short depth limit of 1, and the depth of 6 which LCWoT uses. It might also be reasonable to make this a setting which can be configured by the users.

4.4.3 OpenBazaar Web of Trust proposal

The OpenBazaar project [64] aims to implement a distributed Internet market, similar to eBay for example, with Bitcoin [65] as a currency.
As money is involved, sellers and buyers then need to be able to find out whether they can consider each others as trustworthy.
For these purposes, it was planned to implement a system similar to WoT [66]. Both the proposal and personal communication with the author showed that a completely different approach is to be taken:
For purposes of anonymity of the involved users, there shall be no global, public knowledge of identities or trusts. The "is Identity X trustworthy?" question will not be answered locally by a database query, but by sending a query to the peer-to-peer network. Instead of revealing their knowledge of all identities and trusts, the peers will only answer the trust-queries individually.
This approach will for sure be less intensive with regards to local resource consumption as each peer does *not* have to download the whole identity and trust graph; and so also does not have to process it.
However, it likely has a completely different security model as more trust is put into immediate peers: They are used to download information about peers other than themselves.

Further, without local knowledge of the global network, it is a lot more difficult to debug the implementation: Where WoT can answer whether its computed Scores are correct by comparing the results of using multiple Score computation implementations, a proposed networked trust system will possibly yield different results for *every* query. This is because the nature of networks is unstable, dynamic behavior.

Besides that, it it is also noteworthy that the efforts OpenBazaar is conducting to invent a peer-to-peer market system are unfortunately a duplication of work: An isomorphism which maps a market to a forum system can easily be imagined. A sub-forum such as "Computers / Hardware / Memory" is a product category. A sell offer is a thread in such a forum, a buy offer is a reply to a thread.

Therefore, an existing Freenet-WoT-based forum system such as Freetalk [67] could be used to implement a market. But instead OpenBazaar seems to be implementing a market system from scratch, and also wants to go down the road of re-inventing WoT.

For these reasons, efforts have been made to contact OpenBazaar and provide help to implement OpenBazaar using Freenet technology. The team seems to be willing to accept related code into their repository, but lacks a programmer willing to implement it. Readers are encouraged to consider volunteering in this area!

With regards to OpenBazaar's WoT proposal of a "networked" trust system, it can be concluded that the approach is interesting, but very different. It would likely be a full rewrite of the Freenet WoT codebase, and in the end not share much in common. With this idea on mind, a suitable recommendation seems to be: Before any work is invested in implementing the OpenBazaar WoT system, it should be thoroughly scientifically studied whether it can be implemented in a secure way. This is crucial because all users in Freenet are anonymous, and thus putting all trust into the immediate peers when downloading results to trust queries is very risky.

4.4.4 Further related works

The time constraints of the thesis did not allow investigation of the technical details of further related systems: The author did not realize soon enough that related works should be part of any bachelor's thesis.
Part of the reasons for this might have been that the provided LaTeX-template did not mention that related works are a mandatory component of a bachelor's thesis. It shall be suggested to make related works a chapter in the standard template.
Nevertheless, for the reader's own investigations, the following related works shall be provided without comment:

- Advogato Trust Metric [35]. This inspired the WoT metric.
- Credence [68]. Kindly provided by A. Babenhauserheide.
- GnuPG [69]
- LOCKSS [70]. Kindly provided by A. Brinkmann.

A Bonus work

Let it please be politely requested to acknowledge the following additional pieces of work when deciding about the grade for this thesis. While not being interesting enough to have been noted as own work in the front matter, they nevertheless constitute significantly important "backend" work.

A.1 Identity file queuing

As explained on page 43, a disk-based queue for Identity files was implemented.
It is an additional improvement of apparent performance: Previously, each Identity file was processed on a thread of its own. As Score computation is single-threaded, if more than one Identity file was downloaded by Freenet, the additional threads would stall waiting for the Score lock [71].
The consequence of that used to be that possibly hundreds or even thousands of threads stalled: Downloading Identity files from Freenet is often much faster than processing.
This excessive thread load caused a lot of memory usage. Further, the UI would become a lot slower: The UI thread had to compete against hundreds of threads when trying to acquire the Score lock.
As suggested by F. Daignière, with the new Identity file disk queue, there is only a single thread which processes Identity files [72]. This improves the apparent speed of the UI a lot, since it only has to compete against that thread.
There also is a real performance improvement with Identity file queuing: As decided in a team discussion, the disk queue was implemented to be *deduplicating* [73]: When multiple editions of the Identity file of the same Identity are queued, only the latest edition will be processed. Interestingly, this benefit will increase on slower machines, where it is also of more use: The longer the queue takes to process, the bigger its size, the higher the probability that multiple editions of an Identity's file arrive in the queue and can be deduplicated.
The correctness of the Identity file queuing code was validated in a robust way:
Not only was a disk based implementation written, but also a memory based one [74]. For simplicity and thereby less potential for bugs, the memory-based queue does not deduplicate.
A unit test was then written which feeds both queues with a random permutation of the same Identity files [75]. Several threads are created to consume the queues - the parallel execution ensures that threading issues may be triggered. The both WoT databases which result from processing the two queue implementations are then compared: If they are identical, it can be assumed that both queues produce the same results. As two different implementations are unlikely to have the same bugs, it is hoped that this tests correctness of the queue code.

A.2 Correctness test of new Score computation

On page 42, it was explained that a software benchmark was preferred over a mathematical proof for the sake of long term code maintainability. This was done because a software benchmark will continue to provide measurements even if the implementation receives further improvements.
The same idea was applied with regards to checking correctness of the new incremental Score computation code: Software-based tests can be of use for future development. Unit tests provide a way of even automating the testing as part of the compilation procedure.

A.2.1 Correctness test using Identity file queue

The original plan of the thesis contract was to use the aforementioned Identity file queuing (page 63) for a correctness test: The developer would gather a network dump using the Identity file queue's archival capability. This dump would then be fed into the queue of a test run with once the old and once the new Score computation code. The resulting WoT databases would be compared, and if they matched, the new Score computation code could be assumed to produce the same results as the old one.
This was in fact conducted for ~16 000 Identity files, albeit with a slight modification:
As we learned in the section about the old Score computation reference implementation, it is able to recompute all Scores from scratch without relying upon any pre-existing values (page 11). What was not mentioned yet is that the existing code is also able to test correctness of the existing database contents: It compares the results of its own against what was in the database.
This capability was used to write a command line tool which is able to test whether a WoT database is correct in terms of the old Score computation reference implementation [51].
The tool was used upon the database which was produced from processing the 16000 Identity file dump, and determined all Scores to be correct.
Furthermore, the database which was a result of the benchmark of doing 5000 random *removeTrust()* operations (page 42) was also tested and found to contain correct Scores.

A.2.2 Unit tests

The existing unit tests of WoT already were quite sufficient: They conduct random changes upon a WoT database, and use the Score computation reference implementation to check whether the Score database is correct [76]. This helped debugging a lot.

For being able to determine bugs in rank computation with shorter test runs, a test was written to compare the results of the now 3 available implementations of rank computation [77]:

- Full Score computation reference implementation (page 11)
- computeRankFromScratch() initial implementation (page 33)
- computeRankFromScratch() optimized implementation (page 37)

The test produces a random Identity / Trust graph using the pre-existing framework, and calls rank computation upon all Identities in the graph. It checks whether the three implementations yield the same results. It succeeded in multiple executions.

A.3 Pseudocode

All listings of pseudocode, both those which explain the pre-existing WoT code, and the ones which explain new code, are strongly refurbished compared to the original code: A lot of thought was put into how to reduce the code in size and to restructure it for better ease of understanding. The pseudocodes can thus be considered as own pieces of work.

A.4 Event propagation

For ease of understanding, the pseudocodes both of pre-existing and new Score computation code were stripped from code to update the status of the objects of WoT core classes IdentityFetcher and SubscriptionManager. It is quite a bit of effort to correctly propagate the Score changes to those classes. The new Score computation code both ships new code to do that, and validates that the new code is correct using existing and new unit tests.
Notably, the ability of the Score computation reference implementation to validate the database contents (page 64) was amended to be able to validate the state of event propagation to class IdentityFetcher [78].

B Obtaining the thesis' source code

The official WoT source code repository [79] is managed using Git [80].
Git uses cryptographic hashes to identify commits, and the hashes are built from the content of the commits, *and* the chain of hashes of the previous commits.
Due to the hash chaining, a single hash uniquely identifies the whole state of a repository. No file in the repository can be modified without changing the hash.
With this knowledge, it can be understood why the bibliography claims to provide "permanent" links when linking the code: The links contain the commit hashes, and thus the linked code *cannot* be changed by the author afterwards.
While the thesis document may ship with a CD-ROM to include the source code, it is thus nevertheless safe to obtain the source code using the hash of the last commit of the thesis:

```
1    git clone https://github.com/freenet/plugin-WebOfTrust.git
2    cd plugin-WebOfTrust
3    git checkout e187b5cad6df3bcd94a59efff0447ce5d8cdc18e
```

The commit hash of the first commit of the thesis is 232164858736dceef9103f72de4adb229c75d100 [81], the last commit is e187b5cad6df3bcd94a59efff0447ce5d8cdc18e [82].
The commits of the thesis can thus be viewed by:

```
1    # Hashes shortened for line length. Git will accept those, but please use
2    # long ones for more security.
3    gitk 23216485~1..e187b5ca
```

A full diff can be viewed by:

```
1    # The "~1" tells diff to compare against the last commit *before* the thesis
2    git diff 23216485~1..e187b5ca
```

A full diff can also be viewed online with proper syntax highlighting on Github at [83].

C Statistics about the thesis' source code

Furthermore, for your convenience, here are some numbers to measure the quantity of work which was put into this thesis' code.
The numbers of added ("+") and removed ("-") lines per file add up to:

```
$ git diff --stat=80 23216485~1..e187b5ca

 build.xml                                          |   37 +--
 src/plugins/WebOfTrust/Configuration.java          |   84 +--
 src/plugins/WebOfTrust/Identity.java               |   42 +--
 src/plugins/WebOfTrust/IdentityFetcher.java        |  156 ++--
 src/plugins/WebOfTrust/IdentityFile.java           |  136 +++
 src/plugins/WebOfTrust/IdentityFileDiskQueue.java  |  514 ++++++++
 .../WebOfTrust/IdentityFileMemoryQueue.java        |  113 ++
 src/plugins/WebOfTrust/IdentityFileProcessor.java  |  252 ++++
 src/plugins/WebOfTrust/IdentityFileQueue.java      |  206 ++++
 src/plugins/WebOfTrust/Persistent.java             |   51 +--
 src/plugins/WebOfTrust/Score.java                  |   10 +--
 src/plugins/WebOfTrust/Trust.java                  |   18 +--
 src/plugins/WebOfTrust/WebOfTrust.java             | 1244 ++++++++++++++++++--
 src/plugins/WebOfTrust/XMLTransformer.java         |   15 +--
 .../exceptions/DuplicateScoreException.java        |    4 +
 .../exceptions/NotInTrustTreeException.java        |    4 +
 src/plugins/WebOfTrust/l10n/lang_de.l10n           |   21 +
 src/plugins/WebOfTrust/l10n/lang_en.l10n           |   26 +--
 src/plugins/WebOfTrust/l10n/lang_it.l10n           |    7 +--
 src/plugins/WebOfTrust/l10n/lang_ru.l10n           |    2 -
 src/plugins/WebOfTrust/ui/terminal/WOTUtil.java    |  246 ++++
 .../WebOfTrust/ui/terminal/package-info.java       |    8 +
 src/plugins/WebOfTrust/ui/web/StatisticsPage.java  |  116 +--
 src/plugins/WebOfTrust/util/Base32.java            |  169 +++
 src/plugins/WebOfTrust/util/StopWatch.java         |   44 +
 .../plugins/WebOfTrust/AbstractJUnit4BaseTest.java |   20 +--
 test/plugins/WebOfTrust/BenchmarkTest.java         |   60 -
 test/plugins/WebOfTrust/IdentityFileQueueTest.java |  227 ++++
 test/plugins/WebOfTrust/Issue0006599.java          |   94 ++
 test/plugins/WebOfTrust/RankComputationTest.java   |  117 ++
 .../WebOfTrust/ScoreComputationBenchmark.java      |  848 ++++++++++++++
 .../WebOfTrust/ScoreRecomputationBenchmark.java    |   64 +
 .../WebOfTrust/SubscriptionManagerFCPTest.java     |    2 +
 test/plugins/WebOfTrust/WoTTest.java               |  105 +--
 wotutil.sh                                         |    3 +
 35 files changed, 4779 insertions(+), 286 deletions(-)
```

The total number of commits is:

```
    $ git rev-list --count 23216485~1..e187b5ca

    336
```

D Copyrights

Where not further noted, all figures and code in this thesis have been produced by myself, or are based upon reading existing Freenet sources and have been extended by myself by an amount significant enough to be considered as an own piece of work. Content which is quoted as pre-existing, non-own work is where not further mentioned owned by Freenet Project Inc. and licensed with the GNU General Public License (GPL) or later versions of it [84]. The GPL inherently permits usage for the purposes of the thesis.
Especially shall it be noted that where not further noted, all copyright of my work is granted to Freenet Project Inc. for the purpose of unlimited publishing with any license of their decision.

BIBLIOGRAPHY

[1] Freenet Project Inc. Official website of the Freenet Project and its foundation. https://freenetproject.org/ *Loaded on 2015-07-31.*

[2] Various contributors. Article about sub-projects of Freenet on the official Freenet Project Wiki. https://wiki.freenetproject.org/index.php?title=Projects&oldid=2817 *Loaded on 2015-07-31. Permanent link to the latest revision created on 2015-07-25. The Wiki can be considered as an official source since registration requires manually contacting the Freenet team.*

[3] Various contributors. Article about Web of Trust on the official Freenet Project Wiki. https://wiki.freenetproject.org/index.php?title=Web_of_Trust&oldid=2569 *Loaded on 2015-07-31. Permanent link to the latest revision created on 2014-05-23. The Wiki can be considered as an official source since registration requires manually contacting the Freenet team and the article has not been edited for over a year.*

[4] Previous own work. WoT bugtracker, issue 0005748: Do not synchronize reads of the web interface. https://bugs.freenetproject.org/view.php?id=5748 *Loaded on 2015-08-10.*

[5] I. Clarke. Freenet white paper. https://freenetproject.org/papers/ddisrs.pdf *Loaded on 2015-07-31*, **1999**.

[6] M. Toseland. *Personal communication with former chief Freenet developer and employee of Freenet Project Inc.*

[7] S. Dougherty. Freenet Statistics. http://localhost:8888/SSK@pxtehd-TmfJwyNUAW2Clk4pwv7Nshyg21NNfXcqzFv4,LTjcTWqvsq3ju6pMGe9Cqb3scvQgECG81hRdgj5WO4s,AQACAAE/statistics-927/ *Loaded on 2015-07-31. Permanent link to version of 2015-07-31. Requires Freenet to be viewed.*, **2015**.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*, page 594. The MIT Press, 3rd edition, **2009**.

[9] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. http://www.iacr.org/cryptodb/archive/2003/EUROCRYPT/2005/2005.pdf *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings. Loaded 2015-08-28.*, **2003**.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*, page 644. The MIT Press, 3rd edition, **2009**.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*, page 643. The MIT Press, 3rd edition, **2009**.

[12] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, pages 83–85. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, **2009**.

[13] A. Felner. Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. *http://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/view/4017/4357 Proceedings, The Fourth International Symposium on Combinatorial Search (SoCS-2011). Loaded on 2015-08-22*, **2011**.

[14] Freenet Project Inc. Understand Freenet. *https://freenetproject.org/understand.html Loaded on 2015-08-01.*

[15] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1 (Second Edition). W3C Recommendation, W3C - World Wide Web Consortium, **September 2006**.

[16] The Tor Project, Inc. Official website of the Tor project. *https://www.torproject.org/ Loaded on 2015-08-28.*

[17] Freenet Project Inc. How is Freenet different to Tor? *https://freenetproject.org/faq.html#tor Freenet frequently asked questions. Loaded on 2015-08-28.*

[18] Freenet Project Inc. and various contributors. Official Git repository of the Freenet code. *https://github.com/freenet Loaded on 2015-08-28. Lines of code were computed to include Freenet and the most important WoT based sub-projects. Individual numbers are: fred (= Freenet) 265004, WoT 35340, Freemail 23755, Freetalk 26270, FlogHelper 5770.*

[19] Freenet Project Inc. Donations. *https://freenetproject.org/donate.html Loaded on 2015-08-26*.

[20] Own work. Commit which merges part of the thesis' work into the main WoT Git tree. *https://github.com/freenet/plugin-WebOfTrust/commit/c3de0e0928c06bb0a4c0f7c95be7dd1dbb12e2ee Permanent link to commit c3de0e0928c06bb0a4c0f7c95be7dd1dbb12e2ee*.

[21] Own work. Commit which merges part of the thesis' work into the main WoT Git tree. *https://github.com/freenet/plugin-WebOfTrust/commit/9a96de68c1effa4413c7e938bf573385aeccfd7b Permanent link to commit 9a96de68c1effa4413c7e938bf573385aeccfd7b*.

[22] Own work. Commit which merges part of the thesis' work into the main WoT Git tree. *https://github.com/freenet/plugin-WebOfTrust/commit/1342cf8cc04ecaee55716bf501e4a840cbcc3702 Permanent link to commit 1342cf8cc04ecaee55716bf501e4a840cbcc3702*.

[23] Own work. Commit which merges part of the thesis' work into the main WoT Git tree. https://github.com/freenet/plugin-WebOfTrust/commit/52cfa5e27a22845557da3151fa16c2071836f6a3 *Permanent link to commit 52cfa5e27a22845557da3151fa16c2071836f6a3.*

[24] Own work. Commit which merges part of the thesis' work into the main WoT Git tree. https://github.com/freenet/plugin-WebOfTrust/commit/f25fe3c0286e4031def04514006ac6e6708825de *Permanent link to commit f25fe3c0286e4031def04514006ac6e6708825de.*

[25] Own work. Commit which merges part of the thesis' work into the main WoT Git tree. https://github.com/freenet/plugin-WebOfTrust/commit/50c0d203f7ada86341dc1bfee673e54a42ed9425 *Permanent link to commit 50c0d203f7ada86341dc1bfee673e54a42ed9425.*

[26] Freenet Project Inc., Various contributors. Source code of Freenet before thesis. https://github.com/freenet/fred/tree/build01468 *Permanent link to version "build01468". While this commit happened after the begin date of this thesis, it is the earliest official stable release which the used development branches of WoT will compile against. In other words, the development branch of WoT required code which had not been released in a Freenet version before build01468.*

[27] Freenet Project Inc., Various contributors. Source code of WoT before thesis. https://github.com/freenet/plugin-WebOfTrust/tree/7c254bd62c6940da402e7788fe01ec84d07da539 *Permanent link to commit 7c254bd62c6940da402e7788fe01ec84d07da539. While this commit happened after the begin date of this thesis, it does not include any work from it.*

[28] Freenet Project Inc., Various contributors. Core classes of WoT. https://github.com/freenet/plugin-WebOfTrust/tree/7c254bd62c6940da402e7788fe01ec84d07da539/src/plugins/WebOfTrust *Permanent link to latest commit before thesis (7c254bd62c6940da402e7788fe01ec84d07da539).*

[29] Freenet Project Inc., Various contributors. Reference implementation of Score computation algorithm. https://github.com/freenet/plugin-WebOfTrust/blob/7c254bd62c6940da402e7788fe01ec84d07da539/src/plugins/WebOfTrust/WebOfTrust.java#L1436 *Permanent link to WebOfTrust.java, line 1436, of latest commit before thesis (7c254bd62c6940da402e7788fe01ec84d07da539).*

[30] Freenet Project Inc., Various contributors. Rank computation reference implementation. https://github.com/freenet/plugin-WebOfTrust/blob/7c254bd62c6940da402e7788fe01ec84d07da539/src/plugins/WebOfTrust/WebOfTrust.java#L1475 *Permanent link to WebOfTrust.java, line 1475, of latest commit before thesis (7c254bd62c6940da402e7788fe01ec84d07da539).*

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*, page 658. The MIT Press, 3rd edition, **2009**.

[32] Various contributors. WoT bugtracker, issue 0005758: Profile and optimize computeAllScoresWithoutCommit. https://bugs.freenetproject.org/view.php?id=5758 *Loaded on 2015-08-31.*

[33] Freenet Project Inc., Various contributors. Capacity computation reference implementation. https://github.com/freenet/plugin-WebOfTrust/blob/7c254bd62c6940da402e7788fe01ec84d07da539/src/plugins/WebOfTrust/WebOfTrust.java#L1398 *Permanent link to WebOfTrust.java, line 1398, of latest commit before thesis (7c254bd62c6940da402e7788fe01ec84d07da539).*

[34] Various contributors. Article about the "Six degrees of separation theory" on Wikipedia. https://en.wikipedia.org/w/index.php?title=Six_degrees_of_separation&oldid=672713170 *Loaded on 2015-08-05. Permanent link to the latest revision created on 2015-07-25.*

[35] Advogato. Advogato's Trust Metric. http://www.advogato.org/trust-metric.html *Loaded on 2015-08-05.*

[36] Freenet Project Inc., Various contributors. Score value computation reference implementation. https://github.com/freenet/plugin-WebOfTrust/blob/7c254bd62c6940da402e7788fe01ec84d07da539/src/plugins/WebOfTrust/WebOfTrust.java#L1567 *Permanent link to WebOfTrust.java, line 1567, of latest commit before thesis (7c254bd62c6940da402e7788fe01ec84d07da539).*

[37] Freenet Project Inc., Various contributors. Old incremental Score computation algorithm. https://github.com/freenet/plugin-WebOfTrust/blob/7c254bd62c6940da402e7788fe01ec84d07da539/src/plugins/WebOfTrust/WebOfTrust.java#L2967 *Permanent link to WebOfTrust.java, line 2967, of latest commit before thesis (7c254bd62c6940da402e7788fe01ec84d07da539).*

[38] Various contributors. WoT bugtracker, issue 0005757: Get rid of using computeAllScoresWithoutCommit whereever possible. https://bugs.freenetproject.org/view.php?id=5757 *Loaded on 2015-08-10.*

[39] Own work. Refactored old incremental Score computation to call new algorithm. https://github.com/freenet/plugin-WebOfTrust/commit/44f4eba633aaef1735015f9739769d31766fa260 *Permanent link to WebOfTrust.java diff of the commit which added the relevant code (44f4eba633aaef1735015f9739769d31766fa260).*

[40] Own work. New optimized incremental Score computation algorithm core. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/WebOfTrust.java#L3822 *Permanent link to WebOfTrust.java line 3822 as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[41] Own work. New optimized incremental Score computation algorithm: updateRanksAfterDistrust...(). https://github.com/freenet/plugin-WebOfTrust/

blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/WebOfTrust.java#L4012 *Permanent link to WebOfTrust.java line 4012 as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[42] Own work. New optimized incremental Score computation algorithm: updateCapacitiesAfterDistrust...(). https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/WebOfTrust.java#L4131 *Permanent link to WebOfTrust.java line 4131 as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[43] Own work. New optimized incremental Score computation algorithm: updateValuesAfterDistrust...(). https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/WebOfTrust.java#L3845 *Permanent link to WebOfTrust.java line 3845 as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e). Notice: The linked code was not actually extracted into a function called "updateValuesAfterDistrust...()" yet. For ease of understanding, the simplified pseudocode in the thesis refers to it with that function name already.*

[44] Own work. New optimized incremental Score computation algorithm: computeRankFromScratch() initial version. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/WebOfTrust.java#L3087 *Permanent link to WebOfTrust.java line 3087 as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[45] Own work. New optimized incremental Score computation algorithm: computeRankFromScratch() optimized version. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/WebOfTrust.java#L3188 *Permanent link to WebOfTrust.java line 3188 as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[46] Own work. "Identity file queuing" UI for viewing total processing time. https://github.com/freenet/plugin-WebOfTrust/commit/afd2b7a7747bc7eb079bcc2c44a25a26fb5bd52b *Permanent link to commit afd2b7a7747bc7eb079bcc2c44a25a26fb5bd52b.*

[47] Own work. "Identity file queuing" main merge commit. https://github.com/freenet/plugin-WebOfTrust/commit/78aed8120960d76abf2a81d8419eeac1692848dd *Permanent link to commit 78aed8120960d76abf2a81d8419eeac1692848dd.*

[48] Own work. "Identity file queuing" file archiving code. https://github.com/freenet/plugin-WebOfTrust/commit/78aed8120960d76abf2a81d8419eeac1692848dd#

diff-a30abe4587257da1fa192cd8cf250aafR36 *Permanent link IdentityFileDiskQueue.java line 36 of commit 78aed8120960d76abf2a81d8419eeac1692848dd.*

[49] Own work. "Identity file queuing" deterministic repeat merge commit. https://github.com/freenet/plugin-WebOfTrust/commit/a2f33fe65ebc34a343eaaf9ac6ac306165cc889c *Permanent link to commit a2f33fe65ebc34a343eaaf9ac6ac306165cc889c.*

[50] Own work, Pre-existing library functions. Synthetic benchmark for Score computation. https://github.com/freenet/plugin-WebOfTrust/blob/4d81d13de1786c913e019038d47252962168b96e/test/plugins/WebOfTrust/ScoreComputationBenchmark.java *Permanent link to ScoreComputationBenchmark.java initial draft (commit 4d81d13de1786c913e019038d47252962168b96e).*

[51] Own work. Command-line tool for analyzing WoT databases. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/ui/terminal/WOTUtil.java *Permanent link to WOTUtil.java as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[52] Own work. Synthetic benchmark for Score computation. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/test/plugins/WebOfTrust/ScoreComputationBenchmark.java *Permanent link to ScoreComputationBenchmark.java as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[53] Own work. Benchmark for old Score computation code. https://github.com/freenet/plugin-WebOfTrust/blob/3edbad7a7061e0a03c9bf361e1e80b2a03440c79/src/plugins/WebOfTrust/util/WOTUtil.java *Permanent link to WOTUtil.java commit used for benchmarking the old, pre-thesis Score computation code (3edbad7a7061e0a03c9bf361e1e80b2a03440c79).*

[54] Own work. Benchmark for new Score computation code. https://github.com/freenet/plugin-WebOfTrust/blob/2b035587e3b7086ba8417d25770c6afe49b4d9c8/src/plugins/WebOfTrust/util/WOTUtil.java *Permanent link to WOTUtil.java commit used for benchmarking the new Score computation code (2b035587e3b7086ba8417d25770c6afe49b4d9c8).*

[55] J. R. Douceur. The sybil attack. http://research.microsoft.com/apps/pubs/default.aspx?id=74220 *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS). Loaded on 2015-08-28.*, **2002**.

[56] An anonymous user with the pseudonym "SomeDude". Freenet Message System. http://localhost:8888/SSK@0npnMrqZNKRCRoGojZV93UNHCMN-6UU3rRSAmP6jNLE,~BG-edFtdCC1cSH4O3BWdeIYa8Sw5DfyrSV-TKdO5ec,AQACAAE/fms-143/ *Loaded on 2015-08-29. Permanent link to version of 2015-08-29. Requires Freenet to be viewed.*

[57] An anonymous user with the pseudonym "SomeDude". FMS Trust System. http://localhost:8888/SSK@0npnMrqZNKRCRoGojZV93UNHCMN-6UU3rRSAmP6jNLE,~BG-edFtdCC1cSH4O3BWdeIYa8Sw5DfyrSV-TKdO5ec,AQACAAE/fms-143/trust.htm *Loaded on 2015-08-29. Permanent link to version of 2015-08-29. Requires Freenet to be viewed.*

[58] Various anonymous users. Discussion about FMS trust metric. http://localhost:8080/forumviewthread.htm?messageuuid=6EF3D2A9-9D22-42BC-A1A0-42B7A179B702@dt2mOS7KiAJt8gNInlKxU13zImWDoAOU6KaKswzNtY#6EF3D2A9-9D22-42BC-A1A0-42B7A179B702@dt2mOS7KiAJt8gNInlKxU13zImWDoAOU6KaKswzNtY *Loaded on 2015-08-30. Permanent link a forum thread reply on FMS. Requires Freenet and FMS to be viewed.*

[59] Thomas Markus. Less Crappy Web of Trust. https://github.com/tmarkus/LessCrappyWebOfTrust *Loaded on 2015-08-30.*

[60] Thomas Markus. Less Crappy Web of Trust - ScoreComputer. https://github.com/tmarkus/LessCrappyWebOfTrust/blob/e643966f11dac2afebb87afffb4ba11f724ce1c8/src/main/java/plugins/WebOfTrust/ScoreComputer.java *Permanent link to ScoreComputer.java of commit e643966f11dac2afebb87afffb4ba11f724ce1c8. Loaded on 2015-08-30.*

[61] Thomas Markus. Less Crappy Web of Trust - RequestScheduler. https://github.com/tmarkus/LessCrappyWebOfTrust/blob/e643966f11dac2afebb87afffb4ba11f724ce1c8/src/main/java/plugins/WebOfTrust/RequestScheduler.java#L309 *Permanent link to RequestScheduler.java line 309 of commit e643966f11dac2afebb87afffb4ba11f724ce1c8. Loaded on 2015-08-30.*

[62] Neo Technology, Inc. Neo4j. http://neo4j.com/ *Loaded on 2015-08-31.*

[63] Thomas Markus. Less Crappy Web of Trust - ScoreComputer. https://github.com/tmarkus/LessCrappyWebOfTrust/blob/e643966f11dac2afebb87afffb4ba11f724ce1c8/src/main/java/plugins/WebOfTrust/ScoreComputer.java#L130 *Permanent link to ScoreComputer.java line 130 of commit e643966f11dac2afebb87afffb4ba11f724ce1c8. Loaded on 2015-08-30.*

[64] Various contributors. OpenBazaar. https://openbazaar.org/ *Loaded on 2015-08-30.*

[65] Bitcoin Foundation. Bitcoin - Open source P2P money. https://bitcoin.org/ *Loaded on 2015-08-31.*

[66] Dionysis Zindros. A pseudonymous trust system for a decentralized anonymous marketplace. https://gist.github.com/dionyziz/e3b296861175e0ebea4b *Loaded on 2015-08-30.*

[67] Freenet Project Inc., Various contributors. Freetalk. https://github.com/freenet/plugin-Freetalk-staging *Loaded on 2015-08-30.*

[68] K. Walsh and E. G. Sirer. Credence: Thwarting P2P Pollution. http://credence-p2p.org/ *Loaded on 2015-08-30.*

[69] Various contributors. GnuPG. https://www.gnupg.org/ *Loaded on 2015-08-31.*

[70] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Trans. Comput. Syst.*, 23(1):2–50, **2005**.

[71] Various contributors. WoT bugtracker, issue 0006244: Queue fetched trust lists instead of processing them immediately. https://bugs.freenetproject.org/view.php?id=6244 *Loaded on 2015-08-10.*

[72] Own work. "Identity file queuing" processing thread. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/IdentityFileProcessor.java *Permanent link to IdentityFileProcessor.java as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[73] Own work. "Identity file queuing" deduplication code. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/IdentityFileDiskQueue.java#L265 *Permanent link IdentityFileDiskQueue.java line 265 of commit e187b5cad6df3bcd94a59efff0447ce5d8cdc18e.*

[74] Own work. "Identity file queuing" alternate memory-based implementation. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/src/plugins/WebOfTrust/IdentityFileMemoryQueue.java *Permanent link to IdentityFileMemoryQueue.java as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[75] Own work. "Identity file queuing" unit test. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/test/plugins/WebOfTrust/IdentityFileQueueTest.java *Permanent link to IdentityFileQueueTest.java as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[76] Freenet Project Inc., Various contributors. Part of pre-existing unit tests which implicitly tests Score computation using random WoT operations. https://github.com/freenet/plugin-WebOfTrust/blob/7c254bd62c6940da402e7788fe01ec84d07da539/test/plugins/WebOfTrust/SubscriptionManagerFCPTest.java#L250 *Permanent link to SubscriptionManagerFCPTest.java, line 250, of latest commit before thesis (7c254bd62c6940da402e7788fe01ec84d07da539).*

[77] Own work. Unit test for rank computation. https://github.com/freenet/plugin-WebOfTrust/blob/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e/test/plugins/WebOfTrust/RankComputationTest.java *Permanent link to RankComputationTest.java as of last commit of the thesis (e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[78] Own work. Improved implementation of full Score recomputation: IdentityFetcher state validation. https://github.com/freenet/plugin-WebOfTrust/commit/84277c50c3851446dff6ac89c6223014f711f459 *Permanent link to commit 84277c50c3851446dff6ac89c6223014f711f459.*

[79] Freenet Project Inc., Various contributors. Official WoT source code repository. https://github.com/freenet/plugin-WebOfTrust *Loaded on 2015-08-30.*

[80] Various contributors. Git. https://git-scm.com/ *Loaded on 2015-08-30.*

[81] Own work. First commit of this thesis' source code. https://github.com/freenet/plugin-WebOfTrust/tree/232164858736dceef9103f72de4adb229c75d100 *Permanent link to commit 232164858736dceef9103f72de4adb229c75d100.*

[82] Own work. Last commit of this thesis' source code. https://github.com/freenet/plugin-WebOfTrust/tree/e187b5cad6df3bcd94a59efff0447ce5d8cdc18e *Permanent link to commit e187b5cad6df3bcd94a59efff0447ce5d8cdc18e.*

[83] Own work, Freenet Project Inc., Various contributors. Full diff of the thesis' source code. https://github.com/freenet/plugin-WebOfTrust/compare/7c254bd62c6940da402e7788fe01ec84d07da539...e187b5cad6df3bcd94a59efff0447ce5d8cdc18e#files_bucket *Permanent link to compare the WoT code as of last commit before the thesis to the last commit of the thesis (7c254bd62c6940da402e7788fe01ec84d07da539 and e187b5cad6df3bcd94a59efff0447ce5d8cdc18e).*

[84] Free Software Foundation. GNU General Public License, version 2. http://www.gnu.org/licenses/old-licenses/gpl-2.0.html *Loaded on 2015-07-31.*, **1991**.