TECHNISCHE UNIVERSITÄT DARMSTADT

A SYSTEM FOR PRIVACY-PRESERVING MOBILE HEALTH AND FITNESS DATA SHARING: DESIGN, IMPLEMENTATION AND EVALUATION

MAX JAKOB MAASS

Master Thesis

April 15, 2016

Secure Mobile Networking Lab
Department of Computer Science

SECURE MOBILE NETWORKING

A System for Privacy-Preserving Mobile Health and Fitness Data Sharing: Design, Implementation and Evaluation
Master Thesis
SEEMOO-MSC-0076

Submitted by Max Jakob Maaß
Date of submission: April 15, 2016

Advisor: Prof. Dr.-Ing. Matthias Hollick
Supervisor: Prof. Dr.-Ing. Matthias Hollick

Technische Universität Darmstadt
Department of Computer Science
Secure Mobile Networking Lab

ABSTRACT

The growing spread of smartphones and other mobile devices has given rise to a number of health and fitness applications. Users can track their calorie intake, get reminders to take their medication, and track their fitness workouts. Many of these services have social components, allowing users to find like-minded peers, compete with their friends, or participate in open challenges. However, the prevalent service model forces users to disclose all of their data to the service provider. This may include sensitive information, like their current position or medical conditions. In this thesis, we will design, implement and evaluate a privacy-preserving fitness data sharing system. The system provides privacy not only towards other users, but also against the service provider, does not require any Trusted Third Parties (TTPs), and is backed by strong cryptography. Additionally, it hides the communication metadata (i.e. who is sharing data with whom). We evaluate the security of the system with empirical and formal methods, including formal proofs for parts of the system. We also investigate the performance with empirical data and a simulation of a large-scale deployment. Our results show that the system can provide strong privacy guarantees. However, it incurs a significant networking overhead for large deployments.

ZUSAMMENFASSUNG

Die wachsende Popularität von Smartphones und anderen mobilen Geräten hat eine Reihe an Gesundheits- und Fitness-Anwendungen hervorgebracht. NutzerInnen können ihren Kalorien-Haushalt verfolgen, sich an ihre Medizin erinnern lassen, und ihre Leistung beim Fitness-Training verfolgen. Viele dieser Dienste haben einen sozialen Anteil, der es NutzerInnen erlaubt, sich mit ihren FreundInnen zu vergleichen oder an offenen Wettbewerben teilzunehmen. Allerdings zwingt das aktuelle Geschäftsmodell dieser Dienste die NutzerInnen, alle ihre Daten an den Anbieter zu übertragen. Dies könnte private Daten wie den Aufenthaltsort oder Gesundheitsprobleme beinhalten. In dieser Masterarbeit werden wir ein System zum privatheitserhaltenden Teilen von Fitnessdaten entwerfen, implementieren und evaluieren. Das System soll die Privatheit nicht nur gegenüber anderen NutzerInnen, sondern auch gegenüber den Systembetreibern sicherstellen. Es erfordert keine Trusted Third Parties (TTPs) und garantiert seine Sicherheitsziele durch Kryptographie. Desweiteren versteckt es die Metadaten der Kommunikation (wer kommuniziert mit wem).

Wir evaluieren die Sicherheit des Systems mit empirischen und formellen Methoden und bieten einen formellen Sicherheitsbeweis für einen Teil des Systems. Außerdem untersuchen wir die Performanz des Systems mit empirischen Daten und der Simulation eines großen Systems mit hunderttausenden von NutzerInnen. Unsere Resultate zeigen, dass das System starke Privatheitsgarantien bietet, dabei allerdings bei größeren Nutzerzahlen einen signifikanten zusätzlichen Netzwerkverkehr verursacht.

ACKNOWLEDGMENTS

I would like to express my gratitude to my family for supporting me in the years of my studies, and giving me the time I needed to find out that computer science is indeed the right match for me. In the same way, I'd like to thank Prof. Dr. Hannes Federrath and Dr. Dominik Herrmann at the University of Hamburg for awakening and supporting my interest in IT Security and academic research, and Prof. Dr.-Ing. Matthias Hollick of the TU Darmstadt for helping me take the next steps in that area.

I would also like to thank Daniel Wegemer, Lars Almon, Tom Schons, and Jacqueline Brendel for proofreading (parts of) the thesis. Any remaining mistakes are my own. Additionally, I'd like to thank the Center for Advanced Security Research Darmstadt (CASED), Uwe Müller, and Sven Oliver Pagel / the IVV Geo of the WWU Münster for providing computational resources for the simulations.

Finally, I'd like to thank all of the open source software and open knowledge projects I used over the course of this thesis: Android, Android-Maps-Utils, ArXiv.org, Bash, BibTEX, Bitstring, Bouncycastle, the Cryptography Stack Exchange, Debian, Dia, DNSJava, ejabberd, EventBus, Evince, Firefox, Git, Gitolite, GnuPG, GnuPlot, the IACR ePrint Archive, Inkscape, JodaTime, LATEX, LibreOffice, Linux, Linux Mint, Nosetests, Numpy, Okular, OpenSSH, OwnCloud, Pidgin, Protocol Buffers, PyCrypto, PyProgressbar, PyPy, Python, SciPy, Screen, Signal, SimPy, SpongyCastle, SQLCipher, SQLite, SQLite-JDBC, StackOverflow.com, TEXmaker, Ubuntu, Vim, the Wikipedia, ZXing, and many others. In the spirit of *coding it forward*, all code and data created for this thesis have been released under open licenses.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.

— Alan M. Turing

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ABE	Attribute-Based Encryption
ADB	Android Debugging Bridge
AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
API	Application Programming Interface
BF	Bloom Filter
CA	Certification Authority
CBC	Cipher-Block Chaining
CBF	Counting Bloom Filter
CFNG	Collision-Free Number Generator
CTR	Counter
DH	Diffie-Hellman
DHT	Distributed Hashtable
DNS	Domain Name System
DRBG	Deterministic Random Bit Generator
DRM	Digital Rights Management
ECC	Elliptic Curve Cryptography
ECDH	Elliptic Curve Diffie-Hellman
FPR	False-Positive-Rate
GCM	Galois/Counter Mode
HKDF	HMAC-Based Extract-and-Expand Key Derivation Function
HMAC	Hash-Based Message Authentication Code
IKEv2	Internet Key Exchange, Version 2
IV	Initialization Vector

JCA	Java Cryptography Architecture
KBKDF	Key-Based Key Derivation Function
KDF	Key Derivation Function
MAC	Message Authentication Code
MitM	Man-in-the-Middle
NFC	Near-Field Communication
NIST	National Institute of Standards and Technology
OCB	Offset Codebook
OOB	out-of-band
OSN	Online Social Network
P2P	Peer-to-Peer
PBKDF2	Password-Based Key Derivation Function 2
PCS	Post-Compromise Security
PIR	Private Information Retrieval
PPT	Probabilistic Polynomial Time
PRG	Pseudo-Random Generator
PRNG	Pseudo-Random Number Generator
PSI	Private Set Intersection
QR	Quick Response
RSA	Rivest, Shamir, Adleman
RTT	Round-Trip Time
SHA	Secure Hash Algorithm
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTP	Trusted Third Party
UI	User Interface
VI-CBF	Variable-Increment Counting Bloom Filter
VoIP	Voice over IP
XOR	eXclusive OR

Part I

INTRODUCTION

The first chapter of this part gives an introduction and a motivation to this thesis, followed by a chapter presenting related work and important concepts from the area of private information sharing.

1 INTRODUCTION

With the ongoing rise in mobile computing performance and the decreasing cost of hardware, wearable sensors have passed from being professional medical equipment into the realm of mainstream consumer products. While early wearable devices like the Pulsar Calculator Watch[1] (1970) cost over 1 800 USD adjusted for inflation,[2] a high-end wearable with GPS tracking, heart rate monitoring, and calory and step counters costs less than 300 USD today.[3]

This proliferation of affordable wearable sensors has led to the so-called *quantified self* movement, whose adherents attempt to quantify their bodies and fitness over time, using smartwatches, fitness wristbands, smart scales, and even air quality sensors and genome sequencing services. The devices and services necessary to collect and interpret this wealth of data are offered by commercial entities, which earn money from device sales, service subscriptions, and/or advertising. These services usually have a strong social component, with users competing with their friends and checking each others progress. Proponents argue that this leads to a better understanding of their bodies and a healthier lifestyle.

Over the last years, the *quantified self* has been studied by different scientific disciplines. Computer scientists investigated it as a tool for disease prevention [5], crowdsourced health research [107, 109] and personalized medicine [58, 106, 108], while the social sciences examined the inherent gamification and surveillance elements [116]. However, the collection of health data raises questions of privacy, for both participants and, in the case of genome sequencing, their relatives [62, 99].

In her PhD thesis [95], Raynes-Goldie distinguishes two forms of privacy for online social networks: *social privacy*, which encompasses privacy towards *people* (friends, neighbors, employers, stalkers, ...), and *institutional privacy*, which is privacy towards the *service provider*.

While most services offer their users controls for *social privacy*, *institutional privacy* is rarely adressed by existing *quantified self* solutions. As Raynes-Goldie puts it (using the example of Facebook), „Facebook's privacy settings are entirely focused on the control of infor-

1 See `http://www.vintagecalculators.com/html/pulsar_calculator_watch1.html` accessed 18th of march, 2016

2 300 USD in 1970, inflation adjustment calculated using `http://data.bls.gov/cgi-bin/cpicalc.pl`, accessed 18th of march, 2016

3 Example: Fitbit Surge for 250 USD (`https://www.fitbit.com/surge`), or Garmin Forerunner 230 for 250 USD (`https://buy.garmin.com/en-US/US/into-sports/running/forerunner-230/prod523893.html`). Accessed 18th of march, 2016

mation with respect to other people (social privacy), while providing users no means to prevent information from being shared with Facebook Inc. (institutional privacy)" [95, pp. 105].

The same problem applies to existing health tracking services, which we will discuss next.

1.1 PRIVACY ISSUES IN HEALTH TRACKING SERVICES

A large number of applications currently offer fitness or health tracking features. For example, *Runkeeper*[4] allows runners to track their runs and create personalized fitness plans, while *Fitbit*[5] sells specialized peripherals that allow users to track their step count, calories, and even sleep quality.

These systems usually consist of a combination of a smartphone application (an *app*) and a web service, which is used to synchronize data between devices and provide social features like sharing data with friends or participating in challenges. The data is linked with an account, which is in turn linked to an email address of the user.

Some of the uploaded data is sensitive information. For example, knowing the sleeping habits of people may be interesting to burglars, and knowing the routes a person is regularly running may be valuable information for a stalker. Additionally, advertising companies could use the information to target users with ads they consider relevant: If a user has created a weight-loss training plan, she may be interested in diet products.

In the current systems, all uploaded data is available to the service provider (no *institutional privacy* is provided). Thus, the (mis-)use of uploaded data cannot be prevented by the user, she has to trust the service provider not to misuse the data and to ensure sufficient protection against unauthorized access.

The privacy policy of these services can also shed light on their use of the provided information. For example, at the time of writing, the privacy policy of Runkeeper[6] states:

> This Privacy Policy shall not apply to any unsolicited information you provide to us. This includes [...] all User Content, as that term is defined in our Terms of Use, that by its nature is accessible or intended to be accessible to other users of the Services. All such Unsolicited Information shall be deemed to be non-confidential and we shall be free to reproduce, use, disclose, and distribute such Unsolicited Information to others without limitation or attribution.

4 See `https://runkeeper.com/`, accessed 18th of march, 2016
5 See `https://fitbit.com/`, accessed 18th of march, 2016
6 See `https://runkeeper.com/privacypolicy`, accessed 18th of march, 2016

The company reserves the right to do almost anything with the data that is uploaded to its servers, as long as the data is "intended to be accessible to other users". This is the case if the data is being shared with friends. Similar clauses exist in the privacy policies of other companies. The user has no way of preventing this use, short of not using the service. As is the case for many other internet companies offering free services, the user pays with her data.

However, a study by TNS Emnid [45] recently found that 51% of German users would be willing to pay a monthly fee in return for better data protection. 87% of these 51% would be willing to pay 5 € or more per month. This indicates an untapped market for privacy-focussed applications, as long as they offer the same features and ease of use as existing solutions. This would require a design that guarantees data privacy not only through company policy, but through strong cryptography, in order to be trustworthy.

Additionally, research has shown that metadata can be as critical as content. For example, De Montoye *et al.* [35] were able to uniquely identify 90% of credit card users using 4 or less known data points in a pseudonymized dataset of credit card transactions. While these results are not immediately applicable to a health data scenario, they show the importance of considering metadata in a system design. Consequently, any privacy-preserving application should also strive to minimize the metadata available to all parties.

1.2 CONTRIBUTIONS

The main goal of this thesis is the design, implementation and evaluation of a privacy-preserving mobile health and fitness data sharing system. The contributions of this thesis are:

- Design of a privacy-preserving identifier generation scheme
- Specification of a secure data sharing system that does not require any Trusted Third Parties (TTPs)
- Implementation of a proof-of-concept Android application using the designed system
- Formal and informal security evaluations of the system
- Performance evaluation in a simulated large-scale deployment
- Discussion of the advantages and issues of the proposed solution

Our results show that the system can ensure the confidentiality of the shared data and hide most of the metadata normally associated with data sharing. However, this comes at the price of a potentially significant overhead in the amount of transmitted data for large deployments ($> 100\,000$ users). Further work is required to bring the transmission overhead into manageable regions for large deployments.

1.3 OUTLINE

Margin notes like this will give additional information

This document is structured in four major parts: An INTRODUCTION, followed by the CONTRIBUTION and finally the DISCUSSION AND CONCLUSION, with the APPENDIX containing additional information and the bibliography.

The first part states and motivates the problem in Chapter 1, followed by an overview of related work and important concepts in Chapter 2. The second part introduces the design of the system in Chapter 3 and describes the implementation in Chapter 4. Finally, it contains the evaluation in Chapter 5. The third part contains a discussion of the system and an overview of possible future work in Chapter 6, followed by the conclusion of the thesis in Chapter 7. Finally, the fourth part contains the Appendix and Bibliography.

2 RELATED WORK

In this section, we give an overview of related work. We will first discuss a number of existing proposals for privacy-preserving Online Social Networks (OSNs). Afterwards, we will discuss a number of building blocks that can be used to build privacy-preserving OSNs: Cryptographic hash functions, authenticated encryption to ensure confidentiality and authenticity of data, and techniques for privacy information retrieval.

2.1 PRIVACY-PRESERVING OSNS

The past ten years have seen a number of works on privacy-preserving OSNs. Generally, they can be divided into *centralized* and *decentralized* approaches, where the centralized proposals often attempt to add privacy features to existing social networks. We will give a brief overview over some of the proposals.

Guha *et al.* propose *NOYB* [56], which adds privacy to existing OSNs like Facebook by substituting private information in the user profile with the information of others, a process that can only be reversed by friends of the user who are in possession of a specific key. This approach was chosen over regular cryptography because it is hard to detect and therefore hard to prevent for the OSN provider. However, it does not protect metadata (*who is friends with whom?*) or communication content.

Starin *et al.* designed *Persona* [2], using Attribute-Based Encryption (ABE) [16] to protect the data of users while still allowing users to encrypt data for friends of their friends. This is especially useful for comment features, where a comment should be readable to all friends of the user who posted the original entry. However, ABE is computationally expensive, and the system does not specifically attempt to hide metadata, which is one of our goals.

ABE allows efficient encryption to all users with a specific attribute, like „friends with Alice"

Cutillo *et al.* propose *Safebook* [28, 29], a distributed (peer-to-peer) social network. Data is mirrored on the machines of close friends, and is encrypted unless the user chooses to make it public. However, the design requires a *trusted identification service* as a Trusted Third Party (TTP).

Finally, Sun *et al.* propose a system [105] for privacy-preserving OSNs that offers an efficient method to revoke access for certain users once they are no longer trusted. They use Identity-Based Cryptography (IBC) [19], broadcast encryption [50], and searchable public-key encryption [20]. In order to achieve this, it uses a trusted credential authority as a TTP, which we would like to avoid, as this presents a single point of failure.

This concludes our brief overview on privacy-preserving OSN proposals. None of the proposed systems offer an efficient method for privacy-preserving data sharing with minimal metadata.

2.2 CRYPTOGRAPHIC HASH FUNCTIONS

In this thesis, the term hash function always refers to ***cryptographic*** *hash functions*

Cryptographic hash functions (usually denoted with the letter h) are widely used in many cryptographic applications, from message authentication [8, 68] to key derivation functions [67] and Pseudo-Random Generators (PRGs) [4].

They take an input of an arbitrary length and map it to a fixed-length output:

$$h(\{0,1\}^*) \to \{0,1\}^n$$

More properties of hash functions are discussed by Rogaway et al. in [97]

We are interested in four major properties of hash functions:

1. *Efficiency*: They are fast to compute
2. *Preimage resistance* (cf. [97, § 3.1]): They are non-invertible (i.e. given only a hash $h(x)$, it is infeasible to find x)
3. *Collision-resistance* (cf. [97, § 3.3]): It is infeasible to generate two different messages with the same hash value (i.e. $m_1 \neq m_2$, but $h(m_1) = h(m_2)$)
4. *Avalanche effect*: Changing one bit of the input value will change, on average, half of the bits of the output

This is also known as diffusion

In our case, we are interested in one additional property: Their output *looks random* if the input is not known. While the formal definition of (cryptographic) hash functions does not include this property, real-world hash functions like the Secure Hash Algorithm (SHA) family [53] fulfill this property (with some limitations).

Notably, the avalanche effect combined with the pseudorandomness of the output ensures that hashing two related values (e.g. the integers 1 and 2) produces output that *looks* unrelated if the input values are not known. This will be relevant later in our design.

2.3 AUTHENTICATED ENCRYPTION

Traditional encryption algorithms like the Advanced Encryption Standard (AES) [30] in a normal mode of operation like Cipher-Block Chaining (CBC) or the Rivest, Shamir, Adleman (RSA) system [96] offer encryption, but no authentication of the data. This allows an adversary to modify the ciphertext without being detected, unless the encrypted data is protected using a message authentication algorithm like the Hash-Based Message Authentication Code (HMAC) system [8].

In recent years, a new paradigm called Authenticated Encryption (AE) (or sometimes Authenticated Encryption with Associated Data (AEAD)) has gained popularity. AE algorithms combine the encryption and authentication of the data, thereby both increasing the efficiency and decreasing the complexity of using cryptography as a developer.

Lower complexity generally results in fewer bugs

There are a number of AE algorithms [9, 69, 115], but the most popular one is the Galois/Counter Mode (GCM) [81]. GCM (usually used with AES) internally uses a regular Counter (CTR) mode to perform the encryption, but also calculates an authentication *tag* at the same time. While it isn't as fast as some of its competitors (most notably Offset Codebook (OCB) [69]), it is free from any known patents and still provides good performance and security.

GCM uses Galois fields for authentication, hence the name

2.4 INFORMATION RETRIEVAL

Encrypting data solves only half of the problem: We also need to be able to retrieve the data from a server in a privacy-preserving way. In this section, we will first discuss private information retrieval and private set intersection and motivate why we won't be using them, before moving on to bloom filters, which will play an important role in our design.

2.4.1 *Private Information Retrieval*

Traditional data retrieval methods work by directly or indirectly requesting a certain piece of data from a server. This necessarily implies that the server receives the information *which* piece of data we are interested in. This raises privacy concerns, ranging from personal issues (user has requested information about a certain illness) to business decisions (a high number of queries from one company for certain stock prices may indicate preparations for a buyout or investment).

Private Information Retrieval (PIR) tries to solve this problem by allowing users to privately retrieve data from a server, without the server knowing which data the user requested. PIR can be divided into two classes:

1. *Information-theoretic PIR*, proposed by Chor *et al.* [25], provides privacy through the use of multiple, non-colluding servers.

Hybrid approaches exist, for example Devet et al. [37]

2. *Computational PIR* (cPIR), proposed by Kushilevitz *et al.* [71], achieves privacy through cryptographic guarantees and does not require multiple servers.

PIR is an active field of research, and is connected to a number of other fields, like Oblivious Transfer [47, 22] and Collision-Resistant Hashing [64]. A survey from 2007 listed a number of different cPIR schemes, based on homomorphic encryption, the ϕ-hiding assumption and one-way trapdoor permutations (cf. Ostrovsky *et al.* [87]).

Efficient cPIR was long thought infeasible (cf. Sion *et al.* [104]), but new cryptographic systems have enabled more efficient approaches. At the time of writing, the most efficient published cPIR scheme is called XPIRe. Proposed by Barrier *et al.* [82], it uses lattice-based cryptography and achieves a throughput of up to 10 Gbps, depending on the size of the database (cf. [82, Figure 6]).

However, the user-perceived delay until the data starts arriving scales with the number of database entries, with a worst-case delay of 100 000 seconds for 100 000 database entries (cf. [82, Figure 7]). While the best-case delay is only around 4 seconds for the same number of database entries, the necessary parameters reduce the throughput considerably.

Generally, PIR scales with the number of database entries, making it less and less practical the more entries a database contains.

2.4.2 *Private Set Intersection*

Given two users with two sets of items S_1, S_2, Private Set Intersection (PSI) is used to privately compute the intersection $S_1 \cap S_2$ without one party having to disclose their set to the other party. This could, for example, be used after a data breach at a company resulted in a large number of unencrypted passwords being published to the web. Users could determine if their password is in the set of compromised passwords without disclosing their passwords to any third party, and without having access to the full password database, by privately computing the intersection of the set containing their credentials and the set of compromised credentials. If the resulting intersection is not empty, their credentials are compromised.

PSI protocols, first proposed by Huberman *et al.* [61], are an active field of research. Existing proposals use RSA [34], oblivious transfer [92], garbled or oblivious bloom filters [43, 46], or garbled circuits [60], among other technologies.

Like PIR, all of these protocols introduce a significant overhead in terms of computation and data transmission. The performance characteristics vary between the different techniques and often depend on the size of the sets that are to be intersected. A performance evaluation by Pinkas *et al.* showed the best evaluated protocol to require the transmission of 78.3 MB over the network, with a computation time of 13.8 seconds to compute the intersection between two sets of 2^{18} items (cf. [92, Table 5 and 6]). The large overhead makes PSI unsuitable for a large-scale deployment with large sets, especially on

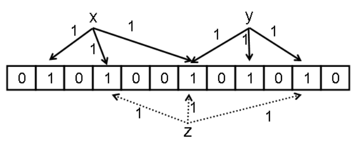


Figure 1: Bloom Filter with two inserted items (x and y) and one false positive query (z). Source: [98, Fig. 1(a)]

devices with limited capabilities and/or slow networks, like smartphones.

2.4.3 *Bloom Filters*

Bloom Filters (BFs) were originally proposed by Burton H. Bloom [18]. They are a space-efficient data structure that can be used for set membership queries (i.e. „is this item in a specific set of items?"). They may give false positives (i.e. claim that an item is in a set while it is not), but will never return a false negative.

Bloom filters work by allocating a fixed number m of bits. When an item is inserted into a bloom filter, it is hashed using k different hash functions. The resulting hashes are used to derive k positions within the m-bit-array, and the bits at these positions are set to 1 (see x and y in Figure 1).

To test if a certain item was inserted into the bloom filter, the same hashes are computed. If at least one of the k bits indicated by the hashes is not set to 1, the item cannot have been inserted into the bloom filter. If all k bits are set to one, the item *may* have been inserted.

Note that bloom filters cannot give a definite statement that an item *has* been inserted, as false positives are possible: See the query for z in Figure 1 for an example: z hashes to three slots that have been set to 1 when x and y were inserted. This results in the bloom filter falsely claiming that z has been inserted. The False-Positive-Rate (FPR) depends on the number of bits, the number of hash functions, and the number of items that have been inserted into the bloom filter.

Another problem with bloom filters is that they do not allow entries to be deleted once they have been inserted. Thus, the number of set bits will never decrease, leading to an ever increasing false positive rate as more and more bits are set.

2.4.4 *Counting Bloom Filters*

Counting Bloom Filters (CBFs) improve on regular bloom filters by also allowing the deletion of items. They were originally proposed by Fan *et al.* [49] and work by using m counters instead of m single bits. When an item is inserted, the k counters determined by the

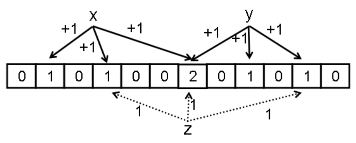


Figure 2: Counting Bloom Filter with two inserted items (x and y) and one false positive query (z). Source: [98, Fig. 1(b)]

hash functions are incremented by one (see Figure 2). Membership tests check if the respective counters are larger than zero, instead of checking for a binary 1. Deletion of items from the filter is achieved by decrementing the adressed counters by one. As illustrated in Figure 2, false positives are still possible.

As counters require multiple bits, they also require more space than a regular bloom filter to achieve the same FPR. Additionally, counters may overflow if too many items are inserted: If 4 bits are used per counter, the maximum number that can be represented is $1111_2 = 15_{10}$. If enough inserts increment the same counter, it could overflow. In order to prevent that, counters are never incremented past their maximum value. However, they are also no longer decremented on deletions if they have reached their maximum value. Otherwise, false negatives would become possible.

False negatives are, however, still possible when working with dynamic datasets: If a false-positive item is removed from the CBF, a false negative could later occur. These problems are described in detail by Guo *et al.* [57].

2.4.5 *Variable-Increment Counting Bloom Filters*

In the later parts of the thesis, the term „bloom filter“ will always refer to a VI-CBF

A Variable-Increment Counting Bloom Filter (VI-CBF) further improves on CBFs by incrementing the counters by different values, depending on the value to be inserted into the BF. Originally proposed by Rottenstreich *et al.* [98], VI-CBFs use two sets of k different hash functions. One family of hash functions determines the counter that is incremented, the other selects the value by which the counter is incremented. This value is taken from a list D of values.

Example: $D_2 = \{2,3\}$, $D_4 = \{4,5,6,7\}, \ldots$

For D, they propose to use a D_L sequence, where $L = 2^i$ (i.e. L is a power of two). D_L is defined as

$$D_L = [L, 2L) = \{L, L+1, \ldots, 2L-1\}$$

The value at position $0 \leqslant i < |D_L|$ of the D_L sequence can be calculated as $L + i$. Using these values allows us to attempt to reconstruct which numbers have been added onto a counter, further reducing the chance of false positives: For example, if we are using $D_4 = \{4,5,6,7\}$ and the counter has the value 9, we know that two values have been inserted into it: a 4 and a 5. If we query for a value that would have

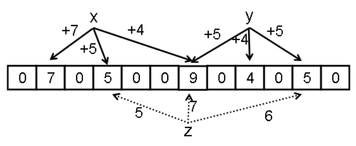


Figure 3: Variable-Increment Counting Bloom Filter with two inserted items (x and y) and one true negative query (z). Source: [98, Fig. 1(c)]

incremented this slot by 7, we can thus be sure that it was not inserted before (see Figure 3).

This strategy decreases the FPR of the VI-CBF, making it more efficient than a regular counting bloom filter. Rottenstreich *et al.* show that the VI-CBF will achieve a lower FPR with the same number of bits as a regular counting bloom filter.

2.5 SUMMARY

In this chapter, we discussed a number of related systems and important concepts. We discussed existing privacy-preserving OSNs and briefly described the concept of cryptographic hash functions, authenticated encryption, and several variants of bloom filters. These will be relevant for our design, which we will motivate, describe and evaluate in the next part.

Part II

CONTRIBUTION

The contribution starts with a design chapter, where we describe the design of the privacy-preserving health data sharing system. After the design follows the implementation of a proof of concept on an Android device. The last chapter concentrates on evaluating the performance of our system using the proof of concept, simulations, and theoretical analysis.

3 DESIGN

In this chapter, we will motivate and introduce the design of our system for privacy-preserving sharing of health and fitness data. We will start with a high-level overview of the goals and architecture, followed by defining our adversary model. We will briefly discuss existing work on securing local data storage, before going into more detail on the server and the different steps involved in our data sharing protocol.

3.1 SYSTEM OVERVIEW

Before we go into more detail, we will give a high-level overview of the system we are designing. We will motivate and discuss the design goals and architecture we have chosen to use.

3.1.1 *Design Goals*

The term „privacy-preserving“ can have many different meanings, depending on the context. We need to more precisely define what we want to achieve. For this purpose, we will be defining a number of *security goals* that our system should achieve.

We will be using the definitions introduced by Pfitzmann *et al.* [90] and RFC 4949 [102]. Our system should achieve the following goals:

- *Confidentiality*
  „The property that data is not disclosed to system entities unless they have been authorized to know the data“ [102, pp. 94]
- *Integrity*
  „The property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner“ [102, pp. 95]
- *Authenticity*
  „The property of being genuine and able to be verified and be trusted“ [102, pp. 28]
- *Availability*
  „[...] a system is available if it provides services according to the system design whenever users request them“ [102, pp. 30]
- *Anonymity* against the Server
  „Anonymity of a subject from an attackers perspective means that the attacker cannot sufficiently identify the subject within a set of subjects, the *anonymity set*“ [90, § 3]

- *Unlinkability* of individual records
  „Unlinkability of two or more [records] from an attacker's perspective means that within the system (comprising these and possibly other [records]), the attacker cannot sufficiently distinguish whether these [records] are related or not" [90, § 4]

Finding a business model for server operators and developers is out of scope for this thesis

We decided to aim for an approach that maximizes privacy and minimizes the knowledge any system participant has about others. Notably, we not only aim for *data* privacy (i.e. keeping the content of the shared data private), but also *metadata* privacy (i.e. keeping private who communicates with whom - anonymity and unlinkability).

Experience with paradigms like Onion Routing [41] or Mix Networks [24, 32] has shown that stronger privacy guarantees usually imply a higher communication and computation overhead. Part of the purpose of this thesis is to evaluate what overhead we have to expect when building a strongly privacy-preserving system.

Note that not all of our security goals can be achieved through protocol specifications alone. For example, the availability may be attacked by performing a *denial of service* attack on the server, which cannot be ruled out by protocol design.

In order to allow sharing of data, the system needs to support the following operations in a secure manner:

- Setting up relationships between users
- Selecting which data to share with which friend
- Storing that data in the system
- Retrieving the data from the system
- Removing information from the system
- Revoking friendships and associated access rights to data

A number of proposals for making use of this data exist, e.g. [5, 58, 106, 107, 108, 109]

Additionally, we would like to support making the data available to researchers. This way, the data can still be used for scientific purposes instead of being locked away completely, assuming the user consents to this use of their data (*opt-in*). This decision is made by the user, and on a case-by-case basis, giving the users certainty that they always know by whom and for which purpose their data is used (*informed consent*). This research functionality requires four additional features:

- Registering research studies
- Joining studies
- Submitting data to studies
- Stopping participation or terminating studies

All of these operations should fulfill the security goals defined above.

Now that we have defined the design goals of our system, we will discuss the architecture of the system.

3.1.2 *Architecture*

We decided to use a centralized (client-server) architecture instead of a Peer-to-Peer (P2P) system, as the limitations of mobile devices (e.g. battery life, processing power, frequent short connectivity outages, limited internet speeds and traffic quotas) can cause problems for P2P systems.

We chose to use a key-value-store paradigm for the server. Clients can store data under a specific key (i.e. an n-bit identifier) using the STORE(KEY, VALUE) function, and retrieve data using the GET(KEY) function. They can also delete data by providing a revocation token using the DELETE(REV) command. If a client attempts to store data under a key that is already in use, requests data for a key under which no data has been stored yet, or attempts a deletion with an invalid revocation token, an error will be returned.

When a new key-value-pair is stored on the server, the key is also inserted into a Variable-Increment Counting Bloom Filter (VI-CBF) (cf. Section 2.4.5). Upon request, the VI-CBF is transmitted to the client. This allows clients to determine if a key is stored at the server (limited by the False-Positive-Rate (FPR) of the Bloom Filter). The server also offers some additional functionality to support research studies, which will be discussed in Section 3.7.

The key-value-store paradigm was chosen because it does not require user accounts (which would violate unlinkability). It also allows us to put most of the application logic into the clients, requiring only a simple server application, which reduces the potential for security issues on the server.

Connections to the server are secured against eavesdroppers and active attackers using a standard transport security protocol like Transport Layer Security (TLS) [39]. This provides confidentiality, integrity and authenticity during transport to and from the server, and also serves to authenticate the server to the clients.

Techniques like Certificate Pinning can be used to increase the security of the connection.

Figure 4 shows a high-level overview of the planned protocol flow. Initially, Alice and Bob communicate directly to perform security-critical actions like establishing and authenticating shared secrets. This step will be specified in Section 3.5. They will then collect data, derive shared identifiers, encrypt the data and upload it. The recipient will also derive the expected identifier, retrieve the ciphertext, and decrypt the data. These steps will be described in Section 3.6.

3.1.3 *Registration and Authentication*

Almost all web services use some form of authentication to prevent unauthorized access to data. However, authentication almost always implies linkability and at least pseudonymity, as all actions of an authenticated user can be linked to that user's identity.

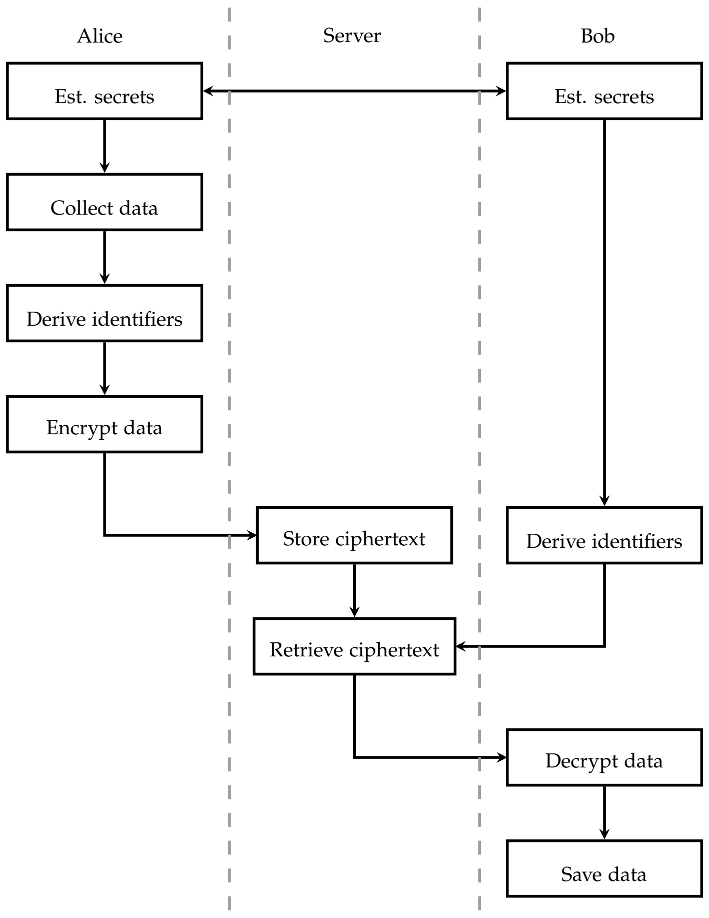


Figure 4: High-level overview of the system

There are some systems which allow for anonymous authentication. For example, Au *et al.* proposed *PERM* [1], a scheme for anonymous authentication which does not require any trusted third party, and which still allows for the blacklisting of misbehaving users. However, the solution introduces an overhead in transmitted data and computation which limits its practicality in a resource-constrained system like a mobile device. The same is true for other anonymous authentication systems [75, 101], making them unsuitable for use in our system until further improvements have reduced the overhead to a manageable level.

As regular authentication protocols violate the goal of unlinkability and anonymous authentication is currently impractical for mobile devices, we chose to forego authentication entirely and rely on the encryption of records on the server. As all records are encrypted using keys known only to authorized devices, the damage from unauthorized read-only access to the data is minimal.

The potential damage will be evaluated in detail in Section 5.3

3.2 ADVERSARY MODEL

To define a secure system, we first need to define the capabilities of the strongest adversary the system is still secure against, and our assumptions about the technologies and algorithms our system uses. In our case, we will distinguish three types of adversaries: An *honest-but-curious* server, a malicious user, and a malicious researcher.

HONEST-BUT-CURIOUS SERVER An honest-but-curious server adversary will obey the protocol (i.e. it will not deviate from the protocol to gain additional information), but will try to find out as much information as possible from the data it stores, sends and receives. It can read any data it has access to (as per the protocol), but is computationally bounded (i.e. it cannot break cryptographic algorithms better than anyone else).

Dishonest servers will be considered in the discussion

This is equivalent to a malicious server administrator that wants to invade the privacy of the users without being found out, and thus limits itself to reading all the data stored on and transmitted to and from the server.

MALICIOUS USER A malicious user is a user that tries to attack other users and/or the server. She can interact with the server in any way, including deviating from the protocol if necessary. She may also interact with other (adversarial or non-adversarial) users, and is also computationally bounded.

MALICIOUS RESEARCHER A malicious researcher tries to attack users of the system. She may, for example, be running a legitimate study with participating users, but try to deanonymize the users. She, too, can interact with the server in any way, including deviating from the protocol, may interact with others, and is computationally bounded.

OTHER ASSUMPTIONS We assume the user device to be trusted. This implies that the device will not sabotage our calculations or disclose any information to others unless the user chooses to allow it.

Otherwise, trustworthy cryptography is impossible

Additionally we assume the network connection used by clients to be anonymous, unlinkable, and confidentiality- and integrity-protected. The server is authenticated. This can be achieved by using an existing anonymization technique like Onion Routing [41] combined with TLS [39] to provide confidentiality, integrity-protection, and authentication of the server. This also implies security against Dolev-Yao-Attackers [42], as Onion Routing and TLS are secure against them.

The widely-deployed Tor network could provide onion routing for this purpose

We will discuss the impact of deviating from these assumptions and models in Chapter 6.

3.3 SECURE LOCAL STORAGE

The problem of encrypted data storage has been solved in a number of ways. Matt Blaze proposed the encrypted file system *CFS* [17], which performs all encryption on the file system layer, requiring no explicit support from the applications generating the data. Manual file encryption is possible using software like *PGP/GnuPG* [119] or the recently discontinued *TrueCrypt*. There are also encrypted database solutions like *SQLCipher* that can transparently encrypt relational databases.

TrueCrypt is continued in projects like TCNext or VeraCrypt

Designing our own system for local data encryption is error-prone and outside the scope of this thesis. We will assume that a system for secure storage of files on the local device exists. This system will be used to store the data collected by the system, and all sensitive key material.

3.4 SECURE UNLINKABLE SHARED IDENTIFIERS

The use of a key-value-store implies that the sender (*Alice*) has a way to let the recipient (*Bob*) know which identifier the data is stored under. Otherwise, Bob would be unable to receive any messages from Alice. It follows that Bob needs to be able to predict the identifier Alice is going to use. However, predictable identifiers seem to conflict with the security goal of unlinkability, which requires the identifiers to seem unrelated to observers.

The solution lies in using secrets shared between Alice and Bob. The identifier generation system we need should make it possible for Bob to predict the identifiers Alice is going to use. At the same time, the resulting identifiers must be unlinkable to anyone not in possession of the shared secret(s). The system should also support a method to generate authentication tokens that can be used to authenticate requests to delete data from the server in a privacy-preserving manner.

Data deletion should be authenticated to prevent deletion of other users' data

The following sections describe how our solution achieves these goals. First, we are going to describe the assumptions under which the system operates. Afterwards, we will discuss the identifier generation procedure. We continue with the revocation authentication strategy, and conclude with a brief comparison between our solution and related systems.

3.4.1 *Assumptions*

Our system operates under the assumption that Alice and Bob share four secrets in total: Two symmetric cryptographic keys, k_{AB} and k_{BA}, and two counters, ctr_{AB} and ctr_{BA}, initialized to a random value. These secrets are only known to Alice and Bob.

We also assume that h is a cryptographic (one-way) hash function whose output is *pseudorandom* (i.e. the function is deterministic, but the output is indistinguishable from a *random oracle* [11]). This function h is known to everyone using the system, including any adversaries.

Before we continue with the specification of the system, we will give a brief overview about the Random Oracle model, as specified by Bellare *et al.* [11]. We will later use this model to prove the security of our identifier scheme.

THE RANDOM ORACLE MODEL Informally, given an input $x \in \{0,1\}^*$ of arbitrary length, a *random oracle* O maps x to a uniformly random output $O(x) \in \{0,1\}^n$ of length n. If the oracle later receives the same input again, it will give the same output. This makes it similar to a (cryptographic) hash function, with the added guarantee that the output is indistinguishable from random data of the same length if the input is not known.

$\{0,1\}^n$ denotes the set of all n-bit strings

A system is secure in the random oracle model if no efficient successful attacker with access to the same oracle O (but not the secret inputs and outputs of the system users) exists. An attacker A is *efficient* if it runs in Probabilistic Polynomial Time (PPT) (i.e. trying out every possible input does not qualify as efficient), and *successful* if it breaks the system with non-negligible *advantage* Adv_A (i.e. the probability that it breaks the system is non-negligibly higher than the probability that it does not). What constitutes a *break* of the system depends on the security goals we are trying to achieve.

Note that a random oracle is a *theoretical* construct, and non-trivial to achieve in practice. For example, hash functions using the Merkle-Damgård construction [31, 83] cannot be used as random oracles, as the structure they introduce to the output can be detected using a length extension attack (cf. Tsudik [113]).

However, it is useful to prove the security of a system *under the assumption that a random oracle exists*. Once this has been shown, we have a baseline of security, and our problem has been reduced to finding a way to implement a random oracle in the real world. For example, the Sponge construction (used by the designated SHA-3 algorithm Keccak [15]) was shown to be indistinguishable from a random oracle if it uses a random transformation or permutation as its round function (cf. Bertoni *et al.* [14]).

Note that this does not mean that SHA-3 itself is a random oracle

In the following evaluations, unless otherwise stated, all adversaries must be efficient (run in PPT), and successful with a non-negligible probability. Using the shared secrets and the random oracle h, we can now specify our system.

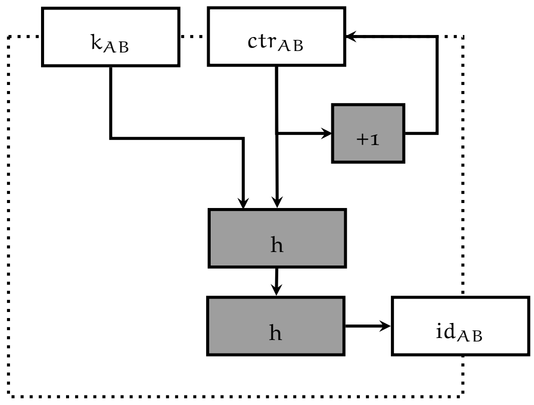


Figure 5: Identifier generation using random oracle h and two shared secrets k_{AB} and ctr_{AB}

3.4.2 *Identifier Generation*

The reason for applying h twice will be explained in Section 3.4.3

The identifier generation process is shown in Figure 5. To generate an identifier id_{AB} Bob can predict, Alice will concatenate the shared key k_{AB} and the counter ctr_{AB}, applying the random oracle h to the result twice, as shown in Equation 1.

$$id_{AB} = h(h(k_{AB} \parallel ctr_{AB})) \tag{1}$$

Afterwards, she will increment the counter ctr_{AB} and use the incremented value for the next identifier.

The system can also generate random identifiers that are not predictable to anyone except the generating party. For this, a random seed value r of n bits is chosen. The identifier id_r is then derived by calculating

$$id_r = h(r) \tag{2}$$

We will now analyze this system and show a number of desireable properties. A more detailed analysis will be given in Section 5.2.

For the purpose of this analysis, we will assume h to be a random oracle. The effects of replacing the random oracle with a real hash function for the implementation will be discussed in Section 5.2.

PREDICTABILITY FOR RECIPIENT One important property of the system must be that Bob can predict which identifier Alice is going to use next, using the secrets he shares with her.

Theorem 1. *Given the shared secrets k_{AB} and ctr_{AB}, Bob can predict id_{AB}.*

Proof (sketch). Bob is in posession of k_{AB} and ctr_{AB}. To predict the identifier, Bob calculates id_{AB} according to Equation 1. As h will always return the same output for the same input, Bob will receive the same identifier as Alice. □

UNLINKABILITY At the same time, we want the generated identifiers to be unlinkable. This means that an adversary who does not know the shared secrets is unable to determine if two identifiers are related (e.g. were generated by the same person, are adressed to the same person, ...). If the identifiers do not have this property, we could not fulfill the unlinkability goal specified in Section 3.1.1.

Theorem 2. *If* h *is a random oracle, the results* $\mathtt{id}_{AB}$ *and* $\mathtt{id}'_{AB}$ *of two iterations of Equation 1 with the same* k_{AB} *and different* $\mathtt{ctr}_{AB}$ *are unlinkable.*

Proof (sketch). Assume Theorem 2 does not hold. This implies the existence of a PPT algorithm A capable of linking $\mathtt{id}_{AB}$ and $\mathtt{id}'_{AB}$ with a non-negligible advantage Adv_A. This algorithm can be used to build a PPT distinguisher B that distinguishes the output of h from truly random data (which is, by definition, unlinkable) in the following way:

Let O be an oracle that either returns the output of the random oracle h or truly random data, depending on a hidden variable $b \in \{0, 1\}$, where $b = 1$ means that it uses h, and $b = 0$ indicates the use of true random data. The goal of algorithm B is to determine the value of b (and thereby determine if O contains true random data or the random oracle h).

Given O, B selects a k_{AB} and a $\mathtt{ctr}_{AB}$ and computes

$$x_1 = O(O(k_{AB} \parallel \mathtt{ctr}_{AB}))$$

Afterwards, it sets $\mathtt{ctr}'_{AB} = \mathtt{ctr}_{AB} + 1$ and computes

$$x_2 = O(O(k_{AB} \parallel \mathtt{ctr}'_{AB}))$$

It now passes x_1 and x_2 to A. If A claims that x_1 and x_2 are linked ($A(x_1, x_2) = 1$), B claims that h is a random oracle ($B(O) = 1$). Otherwise ($A(x_1, x_2) = 0$), it claims that h produces truly random data ($B(O) = 0$).

The advantage of B depends on the advantage of A: If $b = 1$, O uses h. This means that $O(O(k_{AB} \parallel \mathtt{ctr}_{AB}))$ is equivalent to Equation 1 and is therefor a valid input for attacker A. The advantage of A, Adv_A, thus remains unchanged. B will be correct *iff* A is correct. This leads to

iff stands for „if and only if"

$$\Pr[B(O) = 1 \mid b = 1] = 0.5 + Adv_A/2$$

It follows that

$$\Pr[B(O) = 0 \mid b = 1] = 0.5 - Adv_A/2$$

If $b = 0$, A will return 1 („data is linked") only with a negligible probability $negl(n)$, as true random data is unlinkable. This will lead to B returning 0 („true random data") and thereby being right in all but a negligible number of cases.

$$\Pr[B(O) = 1 \mid b = 0] = negl(n)$$
$$\Pr[B(O) = 0 \mid b = 0] = 1 - negl(n)$$

This results in the following advantage Adv_B for B:

$$\begin{aligned} Adv_B &= |\Pr[B(O) = 1 \mid b = 1] - \Pr[B(0) = 1 \mid b = 0]| \\ &= |0.5 + Adv_A/2 - negl(n)| \end{aligned}$$

Subtracting a negligible amount from a non-negligible amount results in a non-negligible amount. Since Adv_A is not negligible, this implies that Adv_B isn't negligible, either. Since A is efficient and a constant number of uses of O are also efficient, B is efficient.

This means that B can efficiently distinguish h from true random data with a non-negligible probability. This contradicts our assumption that h is a random oracle (which is, by definition, indistinguishable from true random data). ↯

The lightning symbol ↯ indicates a contradiction to our assumptions

It follows that the results of Equation 1 must be unlinkable for different values of ctr_{AB}. □

UNPREDICTABILITY FOR OTHERS Also related to unlinkability is the requirement that no adversary may be able to predict which identifier we are going to use next. This is covered in the following two theorems.

Theorem 3. *No adversary not in possession of* k_{AB} *and* ctr_{AB} *can predict* id_{AB}. *No adversary not in possession of* r *can predict* id_r.

Proof (sketch). The output of h is indistinguishable from random data if the input values are not known. This implies that no adversary can do better than randomly guess id_{AB}. Assuming a fixed length of n bits for id_{AB}, this gives the adversary a chance of $1/2^n$ to guess the identifier correctly. Anything else would contradict the assumption that h is a random oracle.

The proof for id_r is analogous. □

This follows from Theorem 2, but is given explicitly for clarity

Theorem 4. *No adversary knowing the used identifiers* id_{AB_0}, ..., id_{AB_n} *(generated with* ctr_{AB}, ..., $ctr_{AB} + n$*), but not knowing* k_{AB} *and* ctr_{AB}*, can predict the next identifier,* $id_{AB_{n+1}} = h(h(k_{AB} \parallel ctr_{AB} + n + 1))$.

Proof (sketch). In the literature, Goyal *et al.* discuss a more general version of this concept under the name *correlated-input security* [55]. They propose a number of security definitions based on correlated input, one of which is the *selective correlated-input unpredictability* [55, Def. 6], based on the *selective Correlated-Input Predicting* (sCI-pred) experiment.

In this experiment, the adversary is given the outputs of the tested function h for n correlated inputs. It then has to predict the output of h for the $n + 1$st correlated input. h is *selective correlated-input unpredictable* if no (efficient) adversary can achieve this (with a non-negligible probability).

This is a simplified explanation. For the full definition, check [55, Def. 6]

According to Goyal *et al.*, it can be shown that random oracles are *selective correlated-input unpredictable* using the techniques used by Bellare *et al.* in [12, Theorem 5.1]. This proves Theorem 3. □

CONFIDENTIALITY Finally, the system involves secret values which have to be kept secret to ensure the security guarantees. This means that we need to make sure our generators do not disclose any information about the cryptographic keys used in the system.

Theorem 5. *$\mathtt{id}_{AB}$ does not disclose $\mathtt{k}_{AB}$ and/or $\mathtt{ctr}_{AB}$ to any adversary. Analogously, $\mathtt{id}_r$ does not disclose r.*

Proof (sketch). Two approaches can be used to prove this theorem. We will briefly outline the first before referring to the second, stronger proof.

Assume Theorem 5 does not hold. This would imply an efficient attacker A with non-negligible advantage that could infer information about $\mathtt{k}_{AB}$ and/or $\mathtt{ctr}_{AB}$ from $\mathtt{id}_{AB}$. From this, we could construct an efficient attacker B on the unlinkability of identifiers that uses the information about $\mathtt{k}_{AB}/\mathtt{ctr}_{AB}$ that A provides to to link the identifiers with non-negligible advantage, violating Theorem 2.

Impagliazzo *et al.* [63] have proven random oracles to be one-way for all but polynomially many cases (defined by $poly(n)$). This leads to a chance of $poly(n)/2^n$ to invert a random input of length n, which becomes negligible for sufficiently large n. This result implies that no PPT attacker can invert h to calculate information about $\mathtt{k}_{AB}$ or $\mathtt{ctr}_{AB}$ from $\mathtt{id}_{AB}$ with non-negligible probability, confirming Theorem 5. The proof for $\mathtt{id}_r$ is analogous. □

This matches our intuition, as random oracles are, informally, improved cryptographic hash functions (adding the guarantee of random-looking output), and cryptographic hash functions have the *one-way* property, meaning that they cannot be efficiently inverted. Additionally, random oracles generate their output randomly, independent of the input they were given. The randomly generated output is in no other way related to the input data, meaning that it does not contain any information about the input data that it could leak.

The general concept is described by Goyal *el al.* as *selective correlated-input one-wayness* [55, Def. 5], and can be proven for the random oracle using the same method used for the *selective correlated-input unpredictability* discussed before.

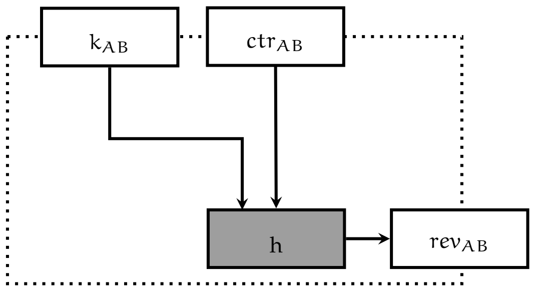


Figure 6: Revocation authenticator generation, using random oracle h and shared secrets

Now that we have shown our identifier system to be unlinkable and secure against information leakage (in the random oracle model), we will proceed with the revocation system.

3.4.3 *Revocation*

Now that we have a privacy-preserving way to store data, we also need a method to privately *delete* data from the server. On the other hand, we don't want *others* to be able to remove our data, so we require some form of authentication.

In a system using some form of persistent identifiers for users (e.g. user accounts, public keys, ...), ensuring that only authorized users can delete a certain record can be achieved by binding the data to their identities and validating their user credentials. However, our system does not have any user accounts, as this would violate the unlinkability goal. Thus, we need to find a way to securely authenticate deletion requests on a per-object basis.

We solve this problem by using the *one-way* property of the random oracle: Since it is impossible to efficiently invert h, Alice can provide a *preimage* of id_{AB} as proof that she was the one to generate that identifier. The *preimage resistance* (cf. Rogaway [97]) of the oracle prevents other from efficiently calculating the preimage, even knowing the identifier.

Recall that identifiers are generated as $h(h(k_{AB} \parallel ctr_{AB}))$. It follows that Alice can calculate the preimage, her *revocation token* rev_{AB}, as

$$rev_{AB} = h(k_{AB} \parallel ctr_{AB}) \qquad (3)$$

(using the value of ctr_{AB} used to generate the identifier). The process is also illustrated in Figure 6. The token can be authenticated by checking if

$$h(rev_{AB}) \stackrel{?}{=} id_{AB}$$

This also means that both Alice and Bob have the knowledge needed to create a revocation token for data shared between them.

For a random identifier id_r, only the person generating them knows the proper preimage: As these identifiers are generated as $\mathrm{id}_r = h(r)$, the value r is required to authenticate their deletion. Disclosing the value r directly (instead of a hash of the value) is not an issue, as it is random and only used for this one identifier, so its disclosure does not impact the security or privacy of the system.

We're now going to briefly state the most important characteristics of this revocation system.

PREDICTABILITY FOR SENDER AND RECIPIENT Alice should be able to predict the revocation token for data she sends - otherwise, she could not use the revocation system. This is true for both id_{AB} and id_r.

Bob should also be able to revoke data sent directly to him under id_{AB}. As the identifier is specific to Alice sending data to Bob, any deletion will only affect Alice and Bob.

Theorem 6. *Given the shared secret* k_{AB} *and the value of* ctr_{AB} *used to generate the identifier* id_{AB}*, both Alice and Bob are able to create a valid revocation token* rev_{AB} *for it.*

Proof (sketch). Analogous to Theorem 1, Alice and Bob are in possession of all required secrets to derive rev_{AB}. □

UNLINKABILITY Adding the revocation system should not harm the unlinkability of the scheme. A limited linkability is trivially shown: rev_{AB} can be linked to its corresponding id_{AB} by computing $h(\mathrm{rev}_{AB})$. This is required, as it constitutes the authentication we need. However, two different revocation tokens, rev_{AB} and rev'_{AB}, generated from the same k_{AB} but different values of ctr_{AB}, should not be linkable.

Theorem 7. *If* h *is a random oracle, the results* rev_{AB} *and* rev'_{AB} *of two iterations of Equation 3 with the same* k_{AB} *and different* ctr_{AB} *are unlinkable.*

Proof. Analogous to proof for Theorem 2. □

UNPREDICTABILITY FOR OTHERS While Alice and Bob should be able to derive the necessary revocation tokens for their communication, no one else should, even if they for some reason know the used identifiers. In the same way, if Alice has generated a random identifier id_r, no one except her should be able to revoke it.

Theorem 8. *No adversary not in possession of* k_{AB} *and* ctr_{AB} *can derive a valid revocation token for* id_{AB}*. No adversary not in possession of* r *can derive a valid revocation token for* id_r*.*

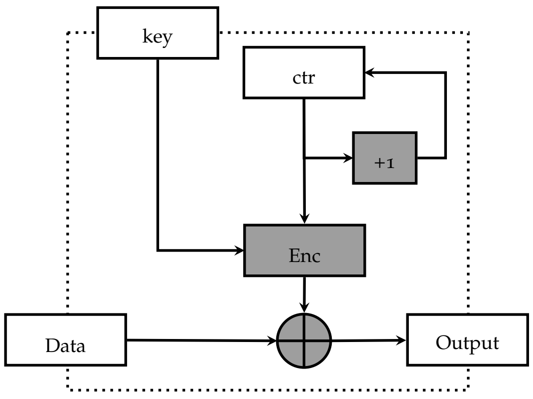


Figure 7: CTR mode illustration (simplified), using block cipher Enc

Proof (sketch). Deriving rev_{AB} from id_{AB} is equivalent to inverting h, as

$$\begin{aligned} id_{AB} &= h(h(k_{AB} \parallel ctr_{AB})) \\ &= h(rev_{AB}) \\ \Rightarrow rev_{AB} &= h^{-1}(id_{AB}) \end{aligned}$$

However, inverting h is not possible, as shown in the proof for Theorem 5. Thus, the adversary is reduced to random guessing without knowing k_{AB} and ctr_{AB}, which does not succeed in PPT with non-negligible probability.

The proof for id_r is analogous. □

CONFIDENTIALITY Finally, disclosing rev_{AB} should not disclose k_{AB} and/or ctr_{AB}, as they are still being used for other identifiers.

Theorem 9. rev_{AB} *does not disclose* k_{AB} *and/or* ctr_{AB}.

Proof (sketch). Analogous to Theorem 5. □

With this, we have shown a number of important properties of our scheme. Note that this is not a full security analysis. Further analysis will be provided in Section 5.2. Next, we are going to take a brief look at the similarities and differences to other, similar systems.

3.4.4 *Comparison with Related Work*

The identifier generation scheme has a few similarities with other algorithms, some of which we will briefly mention here. We will also motivate why we deviated from their design instead of using these algorithms.

BLOCK CIPHER CTR-MODE The most obvious similarity is with the Counter (CTR) mode of operation of block ciphers. Originally proposed by Diffie and Hellman [40], it turns a block cipher (e.g. AES)

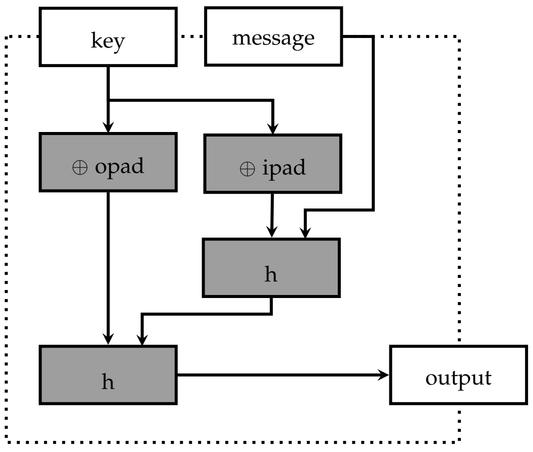


Figure 8: HMAC illustration, using hash function h

into a stream cipher by encrypting different values of a counter and using the resulting ciphertext as a key stream to encrypt data using an eXclusive OR (XOR) operation (denoted by $\oplus$, cf. Figure 7).

The CTR mode can also be used as a Pseudo-Random Number Generator (PRNG) by skipping the XOR step and returning the keystream generated by encrypting the counter. As long as the key and counter values remain secret and we are using a secure cipher, the data should be indistinguishable from true random data.

However, we cannot use this to generate our identifiers: While the values would be predictable to Bob, it does not offer an elegant way to create revocation tokens without disclosing key and counter. This is a fundamental limitation of using such a PRNG. Additionally, symmetric ciphers like AES are slightly slower than hash functions,[1] although the impact would be negligible in a real implementation.

HASH-BASED MESSAGE AUTHENTICATION CODES Hash-Based Message Authentication Codes (HMACs), proposed by Bellare *et al.* [8] and standardized in RFC 2104 [68], are used to authenticate messages in cryptographic protocols. They use a shared secret key and a hash function to create an authentication token for some arbitrary message.

The basic construction is using two nested hash functions (cf. Figure 8): The authentication token t for a message m under key k is computed as

$$t = h(k \oplus opad \parallel h(k \oplus ipad \parallel m))$$

where $opad$ and $ipad$ are two fixed values used to ensure a high *hamming distance* between the two keys.

This construction is more complex than the intuitive $h(k \parallel m)$, as it prevents a number of attacks based on details of the underlying hash

1 See, for example, the benchmarks by Crypto++: `https://www.cryptopp.com/benchmarks.html`, last visited March 11th, 2016.

functions. The most important is the *length extension attack*, which would work on all hash functions using the Merkle-Damgård construction [31, 83] (e.g. MD5 and the SHA1 and SHA2 family, but not SHA3). This attack would allow attackers to append information to a message and compute a valid authentication token for the new message without knowing the secret key (cf. Tsudik [113]). It is prevented by the two nested hash functions HMAC uses.

However, we do not require resistance against this attack, as we do not try to authenticate arbitrary messages, but create identifiers from a fixed-length string of data. An attacker would gain nothing from a length-extension attack, as all identifiers are computed on the clients, without using any external inputs. As the inputs to the hash function have a constant length, the length extension attack also cannot be used to predict future identifiers. Additionally, computing the identifiers as a nested hash prevents a trivial length extension attack on them, as the adversary could only perform a length extension on the result of the inner hash and not the original input values.

Additionally, we again encounter the problem of keeping the identifiers predictable while at the same time being able to generate a revocation token that the server can verify without knowing the secret key. We could compute the identifier as $\mathrm{id}_{AB} = h(\mathrm{HMAC}(k_{AB}, \mathrm{ctr}_{AB}))$ and use $\mathrm{HMAC}(k_{AB}, \mathrm{ctr}_{AB})$ as the revocation token. This would allow the server to verify the revocation token without knowing the key and still keep everything predictable for Bob, but the added security benefit compared to using a simple nested hash as in Equation 1 would be doubtful.

COLLISION-FREE NUMBER GENERATORS Using Collision-Free Number Generators (CFNGs), system-wide unique values can be generated. In [100], Schartner proposes three different types of CFNGs with varying degrees of privacy protection. They have a similar construction to our system, but use an encryption function with a random key instead of or in addition to a hash function. This means that all three types of CFNGs have a random component that cannot be predicted by Bob, making them unsuitable for our use case. Additionally, the same revocation authentication problem as before applies.

3.5 FRIEND DISCOVERY

Any social application faces the problem of letting users add each other as friends, which is a prerequisite for sharing data. In our case, we also need to establish the shared secrets required by the identifier system. In this section, we will discuss how friendships are established and deleted in our system.

3.5.1 *Friendship establishment*

Current systems often establish relationships between users by either matching their address book against a list of registered users, or by having the user enter an identifier of their friend (eMail, username, phone number, ...) and matching it against the database of registered users. These approaches have the advantage of being simple and fast, but they come at the price of having to disclose both your own and your friends contact information to the server.

PRIVATE CONTACT DISCOVERY Some schemes for private contact discovery have been proposed. These schemes are based on a number of different primitives like Diffie-Hellman [61], Garbled Circuits [60], Bloom Filters [43, 46] and Oblivious Transfer [92]. However, the best of these protocols introduces an overhead of multiple seconds of calculations and multiple megabytes of transmission [92], making them unsuited for practical use in our scenario.

Private contact discovery is an instance of the Private Set Intersection problem, cf. Section 2.4.2.

FRIENDSHIP SETUP Instead, we use a direct method for setting up friendships in the system: Two users establish a direct connection between their devices. This connection can use almost any technology - Bluetooth, wireless networks, Near-Field Communication (NFC), even optical channels (scanning QR codes) would be possible.

Once a connection has been established, Alice and Bob perform a key exchange algorithm (e.g. Diffie-Hellman (DH) or Elliptic Curve Diffie-Hellman (ECDH)) to generate a shared secret secret k (cf. Figure 9). They then use a Key Derivation Function (KDF) to derive the two shared secret keys, k_{AB} and k_{BA}, which are securely stored in their respective local databases. They also derive and store the starting values of the two counters, ctr_{AB} and ctr_{BA}, in the same way.

VERIFICATION The resulting counters and keys are verified in a secure, out-of-band (OOB) channel. This channel could be created by a number of technologies, including QR Codes, NFC, or even the manual comparison of fingerprints, as long as the channel is not susceptible to a Man-in-the-Middle (MitM) attack that could compromise the data exchange.

The verification itself works by calculating a hash over all derived values ($k_{AB}, k_{BA}, ctr_{AB}, ctr_{BA}$) and comparing the hash with that computed by the partner. If an active MitM attack has taken place, the key agreement protocol will have resulted in different secrets and the hashes will not match. If the hashes *do* match, an attack can be practically ruled out. In that case, the generated values are saved to the database for later use in the protocol.

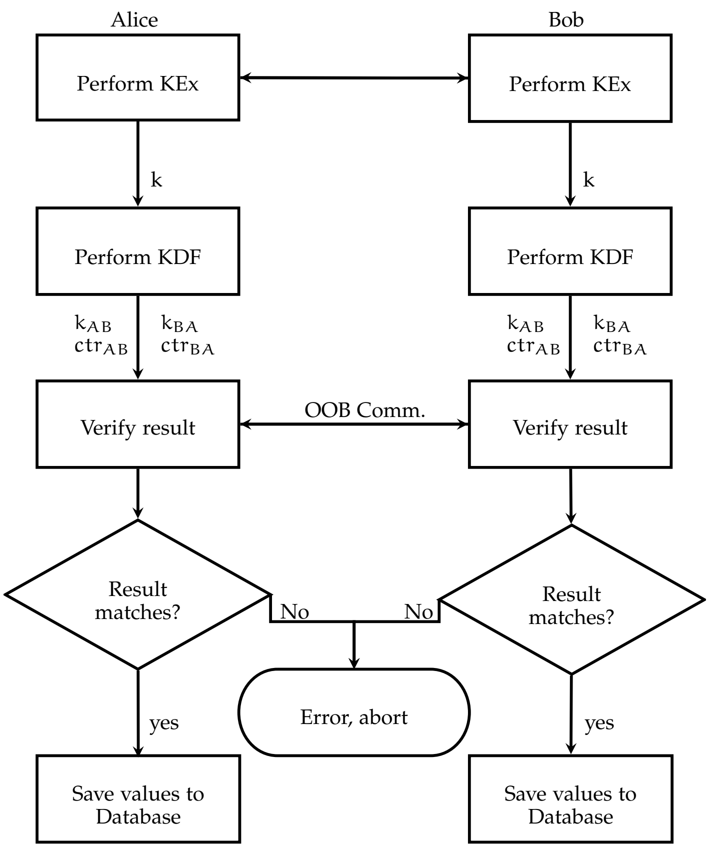


Figure 9: Relationship and key establishment

DISCUSSION This system has the additional advantage that it does not involve the server at all, so there is no need to trust the server with any information for this step. This also means that the server does not get any explicit information about new friendships being established in the system.

The disadvantage is that physical proximity is required to establish a connection between the devices. This increases the difficulty of using the system. A long-range friendship establishment mechanism based on, for example, eMail would be possible, but in that case, the resulting keys could not be trusted until they have been explicitly authenticated - a classical problem in cryptography.

Another classical cryptographic problem is the problem of revoking access to data, which is what we are going to discuss next.

3.5.2 *Revocation*

Being able to share data with friends is only half of the problem - the other half is being able to *stop* sharing data.

INTUITIVE APPROACH The intuitive solution is to re-encrypt everything that the revoked user had access to with a new key and distribute that key to all authorized parties. This involves a lot of cryptographic operations and scales linearily with the amount of data to be re-encrypted. It also does not solve the problem that the user could have made copies of the decrypted data while she still had access - solving this problem would be equivalent to solving the Digital Rights Management (DRM) problem.

In our scheme, the problem is compounded by the fact that the system relies on downloading and decrypting the data once and then storing it in Bob's local database, thereby making it effectively impossible for Alice to regulate access to it. Any attempt to delete the data relies on the cooperation of Bob.

On the other hand, the scheme makes it very easy for Alice to prevent Bob from gaining any further information *after* the revocation: She simply stops producing encrypted key blocks for Bob.

REVOKING PENDING SHARES As there may still be data Bob has not retrieved from the server yet, Alice will check the VI-CBF for all encrypted key blocks she has uploaded for Bob, and delete all that are still on the server (i.e. have not yet been downloaded by Bob). She will perform this process in *reverse* order, i.e. start with the newest $\mathtt{id}_{AB}$ and work her way backwards. As soon as she finds the first identifier that is no longer present on the server, she stops, as this indicates that Bob has downloaded all previous key blocks.

DELETING RETRIEVED DATA In that process, she can derive the current value of Bobs $\mathtt{ctr}_{AB}$ and thereby the next identifier Bob will look for on the server. She can then leave a special message under that identifier, encrypted with k_{AB} using an authenticated encryption mode, instructing Bob's client to delete all history associated with Alice.

Assuming Bob's client is honest, it will see the message and perform the deletion as requested. However, deletion cannot be guaranteed, as Bob may be using a modified client, may have made backups of the database, or even screenshots of Alice's shared data. Thus, privacy for previously shared data cannot be guaranteed.

3.6 SHARING PROCESS

Now that we have established the shared secrets required for our protocol, we can start sharing data between users. This section specifies the process for sharing, retrieving and deleting data on the server.

3.6.1 *Data Selection*

Data can be shared individually, and with a granularity decided by the user. For example, a user may decide to only share their step

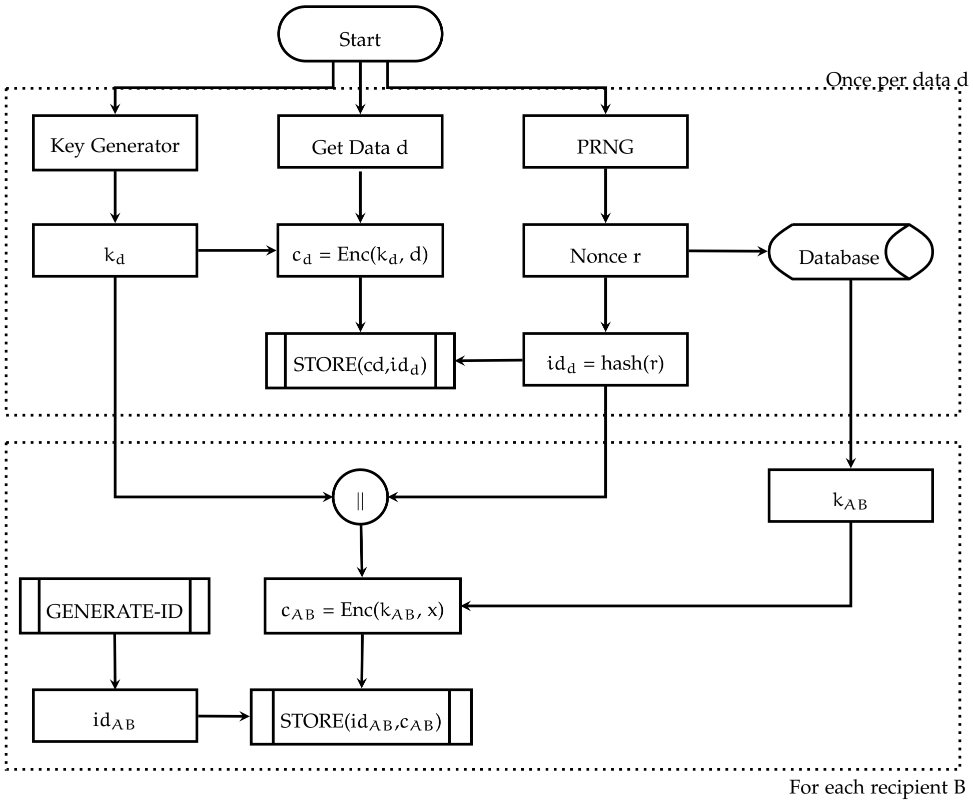


Figure 10: Data encryption and storage on server

count rounded to the closest multiple of 1000, or only share the distance and time of a run (but not the GPS coordinates). The available granularities are decided by the implementation and may vary across different data types.

3.6.2 *Data Storage*

Once Alice decides to share a piece of data with Bob, it needs to be securely stored on the server. We will first discuss the case of sharing data with one specific user. Later, we will expand the process to allow for sharing with multiple users. The process is illustrated in Figure 10.

At this point, Alice already has two shared secrets k_{AB}, k_{BA} with Bob, established during setup of the sharing relationship. She also has a piece of data, d, which she wants to share. d can be anything, from a simple integer value (e.g. a step count) to a complex serialized data structure (e.g. a full GPS track of a training run). d also contains a header which identifies the type of data it describes, and other metadata about the values enclosed in it.

DATA BLOCK GENERATION Alice will now generate a new, random symmetric key k_d and use it to generate an encrypted *data block*

c_d containing the data d we want to share. The data is encrypted using a symmetric cipher in an Authenticated Encryption (AE) mode (cf. Section 2.3), for example the Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) [81]. She receives the data block c_d:

AES in an AE mode like GCM provides Confidentiality, Integrity and Authenticity of the data using one key

$$k_d = \mathrm{KGen}(1^n)$$
$$c_d = \mathrm{Enc}(k_d, d)$$

She now derives a random identifier id_d for the data, as defined in Section 3.4.2 and stores the associated revocation token in her database.

$\mathrm{KGen}(1^n)$ *indicates the generation of a new cryptographic key with security parameter* n

KEY BLOCK GENERATION Next, Alice needs to transmit the identifier and key for the data block to Bob. In order to do that, she creates a *key block* m_B for Bob, stating the identifier of the data and the symmetric key that can be used to decrypt it:

$$m_B = (id_d \parallel k_d)$$

This key block is now symmetrically encrypted with the key k_{AB} in an AE mode like GCM, using the current value of ctr_{AB} plus a random nonce as Initialization Vector (IV). This ensures that the message can only be decrypted if the recipient is using the same counter value. This is an additional security feature against replay attacks, which will be discussed further in Section 6.1.1. The encryption results in c_{AB}.

GCM supports IVs of arbitrary length

$$r \leftarrow \{0,1\}^n$$
$$c_{AB} = \mathrm{Enc}(k_{AB}, m_B, IV = (ctr_{AB} \parallel r))$$

SERVER STORAGE She now uses the shared secrets k_{AB} and ctr_{AB} to derive the next identifier id_{AB} Bob expects her to use, as per Section 3.4.2.

Now, the two pieces of data can be stored on the server with their respective identifiers (including the random IV component r, as it is required to decrypt the key block):

$$\textsc{Store}(id_d, c_d)$$
$$\textsc{Store}(id_{AB}, r \parallel c_{AB})$$

Alice then stores the old ctr_{AB} in her local database, associated with the piece of data she shared. This allows her to derive the revocation token at a later point, if necessary. She also stores the incremented ctr_{AB} for use in the next sharing operation.

MULTIPLE RECIPIENTS This scheme can be trivially extended to facilitate sharing with multiple friends: Alice simply repeats the steps she performed for Bob for all other friends, using their respective keys and counters, and re-using the existing c_d and id_d. In the end, she will upload $1 + n$ key-value-pairs: the encrypted data, and one encrypted set of key and identifier for each of the n friends the data is shared with.

If Bob wants to share data with Alice, he performs the same protocol, using k_{BA} and ctr_{BA} instead of k_{AB} and ctr_{AB}.

3.6.3 *Data Retrieval*

Now that Alice has uploaded data to the server, Bob needs to retrieve it. This poses a series of problems: Because there are no user accounts, the server does not know which user a certain shared piece of data is destined for. This is, in fact, a requirement for the system, because otherwise we would not achieve the goal of unlinkability. However, this means that the server cannot actively notify the recipient about the new piece of data (*push*-principle). Instead, the recipient needs to periodically poll the server to find out if new data is available (*pull*-principle).

That leaves us with another problem: With overwhelming probability, Bob will be the only person interested in the identifier id_{AB}. But Bob does not know if Alice has uploaded anything under that identifier yet, which means that he needs to repeatedly query the server for that identifier until Alice, at some point, uploads some data under it. However, id_{AB} does not change in the meantime, which means that Bob would repeatedly send the same identifier to the server - an identifier unique to him. This violates the unlinkability property, as it would allow the server to link his sessions.

PRIVATE DATA RETRIEVAL Intuitively, this leaves us with the problem of privately retrieving data from a server without the server knowing which data we are interested in. This problem is called the Private Information Retrieval (PIR) problem, and some approaches have been discussed in Section 2.4.1. They all have one thing in common: The required computations and/or data transmission is impractical for a mobile device.

However, we do not need perfect PIR for our application. The server is allowed to receive id_{AB}, but he may only see it once: When we successfully retrieve the data stored under that key. Afterwards, we will never use the same key again, and since the keys themselves are unlinkable, it does not give the server any significant information. This detail intuitively changes our problem from PIR to Private Set Intersection (PSI), as we only need to know if id_{AB} is in the set of uploaded keys.

The server gains the knowledge that a certain value has been retrieved, and nothing else

PRIVATE SET INTERSECTION The general idea of PSI is described in Section 2.4.2. It can be achieved with much lower overhead than

PIR, but the existing solutions still have a high overhead. To the best of our knowledge, the best published PSI protocol was proposed by Pinkas *et al.* [92] and achieves a communication complexity of 78.3 MB and a runtime of 4.9 seconds to compute the intersection between two sets of 2^{18} elements (cf. [92, Table 6 and 8]), with the runtime increasing to 77.5 seconds for 2^{16} elements over a HSDPA mobile internet connection (cf. [92, Table 7]).

Sending the sets in the clear (without any privacy) would require 2 MB

In our case, the sets would be asymmetric: The set of the server may contain many thousand or even millions of entries, while the set of the client depends on the number of friends and will most likely rarely exceed 100 entries. Additionally, the security requirements of PSI are stricter than necessary for our use case: We only require that the server does not learn about the set of the client, while PSI requires that the client does not learn anything about the set of the server, aside from the intersection with its own set. In our case, no harm would come from the client knowing the identifiers stored on the server - all data is encrypted and the identifiers themselves are unlinkable. Accordingly, the overhead of using a PSI scheme is not justified for our use case.

PRIVATE PRESENCE CHECKS Instead, we require a method for the server to provide the client with the list of uploaded identifiers. The intuitive solution would be for the server to send a list of all identifiers to the client. However, this requires a lot of network communication, as transmitting t n-bit keys requires $t * n$ bits of network traffic.

Instead, we use a data structure that allows us to represent the set of uploaded identifiers in a more compact manner: A bloom filter. Bloom filters are described in Section 2.4. They provide a more compact representation of the list of keys on the server, at the price of having a small FPR. We use a VI-CBF (cf. Section 2.4.5) instead of a regular bloom filter, as it offers two improvements: A lower FPR for the same size, and the ability to remove entries without re-generating the whole bloom filter.

As described in Section 3.1.2, the server maintains a VI-CBF of all keys that have been uploaded to it. Upon request, it will provide this bloom filter to clients, which can then use it to privately check for the presence of a certain key on the server.

KEY BLOCK RETRIEVAL The full process of retrieving new data from the server is illustrated in Figure 11. When Bob wants to check if a new piece of data is available, he requests the current VI-CBF from the server. He can then compute the expected identifier id_{AB} Alice would use from the shared secrets, as defined in Section 3.4.2.

Afterwards, he queries the VI-CBF to find out if the server has any data stored under that key: If the key is not in the VI-CBF, he can be

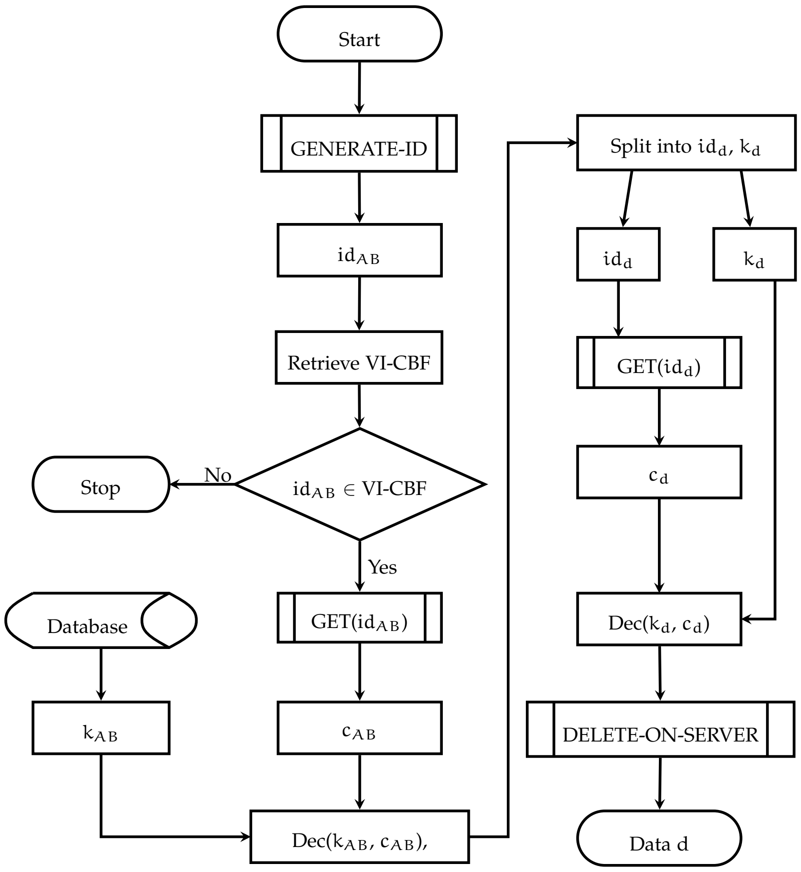


Figure 11: Data retrieval and decryption

certain that Alice has not uploaded any data under that key. If the VI-CBF contains id_{AB}, Bob can send a request to the server to retrieve the associated data.

$$(r \parallel c_{AB}) = \textsc{Get}(\mathrm{id}_{AB})$$

DATA BLOCK RETRIEVAL Assuming the VI-CBF did not return a false positive on the query, Bob will have received c_{AB} from the server. He can now decrypt it using k_{AB}, reconstructing the IV from the current value of ctr_{AB} and the random component r.

$$(\mathrm{id}_d \parallel k_d) = \mathrm{Dec}(k_{AB}, c_{AB}, IV = (\mathrm{ctr}_{AB} \parallel r))$$

Since both id_d and k_d have a fixed length that is known to Bob, he can separate identifier and key and use the identifier to retrieve the encrypted data:

$$c_d = \textsc{Get}(\mathrm{id}_d)$$

The retrieved data is then decrypted using k_d:

$$d = \mathrm{Dec}(k_d, c_d)$$

The decrypted data can be inserted into Bob's local database and displayed to him. After the data has been successfully retrieved and decrypted, c_{AB} is deleted from the server (see Section 3.6.4) and ctr_{AB} incremented.

DECRYPTION FAILURE HANDLING A decryption failure can have a number of different reasons. The data could have been damaged on the server, in which case the authentication check of the AE algorithm would fail. It could also have been maliciously tampered with, which would also result in an authentication failure. Finally, although unlikely, a collision may have occured, and the data may be destined for someone else, in which case we cannot decrypt it as we do not have the key. These three cases are indistinguishable for the client.

There is no straightforward solution to this problem. In case of an identifier collision, our friend may upload data under that identifier at a later date, or she may have already tried and failed to upload data and used the next identifier. If the data was damaged or tampered with, it is unlikely that we will receive an undamaged copy by querying the server again.

We solve this problem by ignoring the block and moving on to the next identifier. If the problem was indeed a collision, this means that we may miss one share by our friend. If the data was damaged or tampered with, we will also have missed one message. However, the alternative (attempting to retrieve the data again at a later point) would allow an adversary to directly influence our behaviour, making it easier to re-identify us later or perform other attacks. Additionally, there is no guarantee that we would ever receive a non-broken copy of the data. Thus, simply ignoring the identifier and moving on is the best decision in the majority of cases.

FALSE POSITIVE HANDLING With a certain probability (depending on the number of entries and the parameters of the VI-CBF), Bob will have received a false positive from the VI-CBF. In this case, the server returns an error. In order to avoid sending further false positive queries, Bob will now locally remember the state of the relevant counters and, when checking a new VI-CBF, only send a new query if at least one of the counters has changed and the key has still plausibly been inserted into the CBF (i.e. querying the bloom filter for the key still returns *true*).

There remains a very small chance that this will result in a situation where a newly inserted key would not be detected: If a query for the key has returned a false positive once, another key has been removed from the bloom filter which decremented *all* relevant counters by the amount the correct key would increment them, and the correct key was inserted afterwards.

This would result in a bloom filter with the same counter values, thereby triggering the false positive cache. While this situation is extremely unlikely, it could be worked around by periodically (e.g. once every day or every week) resending GET-queries which are assumed to be a false positive, compromising unlinkability for availability.

3.6.4 *Data Deletion*

In order to reduce the number of entries in the VI-CBF and the storage load on the server, entries should be deleted from the server when they are no longer needed. Deletion requests are authenticated by providing a *preimage* of the identifier that should be deleted, as described in Section 3.4.3.

TOKEN AUTHENTICATION When the server receives a DELETE(x) command, it computes $i = h(x)$ and checks its local database if it has stored a key-value-pair with the identifier i. If the identifier is found, it removes the key-value-pair and also removes i from the VI-CBF. Finally, it returns a success message. If no identifier i is found, an error is returned.

By verifying if the identifier is actually present in the database, we avoid potentially removing keys from the VI-CBF that have never been inserted. Thereby, we also avoid the problem of potential false negatives described by Guo *et al.* [57] and ensure that our VI-CBF will never give false negatives.

DELETION VS. REVOCATION Simply deleting the key-value-pair may lead to problems if the deletion happens before the recipient retrieved the data. In this case, the recipient would not have seen that the identifier $\mathtt{id}_{AB}$ had been used, and would not increment $\mathtt{ctr}_{AB}$, while Alice would use the incremented counter for her next share. This would lead to a desynchronization, with Bob waiting for an identifier that Alice will never use again.

AES-GCM in particular becomes insecure when re-using key and IV

On the other hand, re-using the same identifier for the next piece of shared data could also lead to problems. Depending on the encryption algorithm, re-using the same key and counter could open up the client to certain cryptoanalytic attacks, and should be avoided. Even if these problems could be avoided, it would introduce complexity to the protocol, which should be avoided if possible.

To solve this problem, Alice can upload a placeholder value in place of the deleted key-value-pair. This indicates to Bob that the identifier has been used, but the data deleted, thereby avoiding the desynchronization of their counters. The placeholder can then be retrieved and deleted by Bob when he next retrieves data from the server.

3.7 RESEARCH FUNCTIONALITY

Although we are discussing research studies here, the same techniques can be used for any similar scenario in other use cases.

A special case of sharing is the case of sharing data with researchers. While we are concerned with the privacy of our users, there may be legitimate interests of researchers, who could use the data to aid their work. Restricting them from accessing the data could impede progress in scientific research.

CURRENT SITUATION In the current commercial centralized server model, research on the data is trivial: The data can be retrieved from the server and given to a researcher. However, while not always legally required, study participation *should* be a voluntary and informed action by the subjects (*informed consent*), and not something decided for all users by the executives of a company. Additionally, there may be privacy concerns when providing data to researchers, meaning that we will have to anonymize or at least pseudonymize the data.

Data protection legislation may add additional requirements.

In our system, the server is unable to provide researchers with the unencrypted health data of the users. In order to preserve the ability of researchers to use that data, a special component is added to the system. This component manages research study requests and allows clients to *opt-in* to (i.e. voluntarily join) studies.

3.7.1 *Creation and Registration*

In order to register a study, a researcher first has to create a study proposal. This proposal serves to inform interested users about the goals of the study, the data that is required, how the data will be used, and other information, as required. It also contains a link to a website with further information on the study, which *must* be protected using TLS with a valid certificate. As university websites almost always offer an HTTPS-protected version, this should not be a problem.

In practice, the study proposal will be generated using a special tool built for this purpose

Alternatively, the open Let's Encrypt CA can provide certificates

The researcher will also generate a unique keypair for a supported asymmetric cipher (e.g. the Rivest, Shamir, Adleman (RSA) cryptosystem or a secure Elliptic Curve Cryptography (ECC) system) and the public data of a key exchange algorithm (e.g. DH or ECDH, with secure parameters) and appends them to the study request. The private key and private parameters to the key exchanged are securely stored by the researcher on their own device.

The study request will also include a random identifier from a different identifier namespace (i.e. if regular identifier generation uses SHA-256 (32 byte), the studies could use a 128-bit identifier). This special identifer will later be used by clients to submit their study registration.

REQUEST AUTHENTICATION Finally, the researcher is required to put a cryptographic hash of the public key into a well-known location relative to the TLS-protected study URL. If this proves impossible (e.g. because the university website does not offer that option), other systems (e.g. a `<meta>` HTML tag) can be used instead. The used method is indicated in the final study request. The final study proposal is signed using the private key and sent to the server in a *study-register* message.

The server, upon receiving the message, will check for the presence of the public key hash at the expected location and verify that it matches the hash of the public key contained in the request. It will also verify that the URL is TLS-protected with a trusted certificate, and check if the chosen random identifier is available. Finally, it will verify the signature on the request using the public key. If every verification succeeds, the study requests is added to the list of available studies and the random identifier is registered to that study. Otherwise, an error is returned to the researcher.

3.7.2 *Retrieval and Participation*

Clients will occasionally check for the presence of new study proposals, if configured to do so. Once a new study proposal is detected and has been retrieved from the server, the client will check if the requested data is available (if the study required blood pressure data but the client is not connected to a blood pressure sensor, it cannot participate in the study). If the requested data is available, the study can be presented to the user.

JOINING A STUDY The user will read the information about the study and decide if she is willing to join it. If she decides to join, the client will retrieve the researcher's public key hash using the method indicated in the study request, verify that the connection to the URL was secured with a valid TLS certificate, verify that the retrieved hash matches the hash of the public key included in the request, and finally verify the signature over the request. If the verifications succeed, the request data is assumed to be authentic (i.e. has not been modified by the server or other adversaries).

The authenticity of the data is derived from the authenticity of the TLS certificate.

The client will also perform the second half of the key exchange protocol specified in the request and receive a shared secret s, which can then be expanded into two keys and two counters, as described in Section 3.5.1. Keys, counters, and the researcher's public key are saved into the secure local database of the client. The public data of the key exchange is added to the *study-join* message.

The *study-join* message is encrypted with the public key of the researcher using a hybrid encryption scheme and uploaded to the server using the identifier specified in the study. The special study identifiers work differently from the regular key-value-pair storage: Instead of storing only one value, they can store a list of values, to which new *study-join* requests are appended.

JOIN HANDLING The researcher can retrieve the *study-join* message from the server, authenticating the request using their private key. After retrieving the message, she can perform her half of the key exchange using the public data included in the request, derive keys and counters, and securely save them to her local database.

study-join messages can only be retrieved by the authenticated researcher.

Note that the *study-join* message is not explicitly authenticated. This means that an adversary could theoretically replace it with her own. However, this would only mean that the researcher would never receive data from the participant who sent the message, as she is not in possession of the keys and counters it derived. Thus, modifying the message would be equivalent to dropping it from the perspective of the researcher. The attacker also gains nothing from providing her own data, as this would be equivalent to joining the study herself, which she could have done either way.

UPLOADING AND RETRIEVING STUDY DATA Study data is uploaded and retrieved in the same way regular data is uploaded. Refer to Section 3.6.2 and Section 3.6.3, respectively, for the details. The client may be configured to automatically upload study data without user interaction as soon as it becomes available.

As the derived symmetric keys are (with overwhelming probability) unique, they serve as a pseudonym for study participants. Thus, multiple pieces of data from the same participant can be linked, while not providing any clue as to the real identity of the participant. While this seems to violate the *unlinkability* property, this is desired in the case of medical studies: Different samples from the same subject often need to be linkable to provide useful information for the researcher.

SENDING MESSAGES TO STUDY PARTICIPANTS Occasionally, a researcher may want to contact one or all study participants to provide updates on the study or ask for further information. These messages can be treated like normal data in the system and can be sent and received using the normal mechanisms described in the previous section. Replies to these messages by the clients can also be treated the same way.

3.7.3 *Termination*

In order to end a study and remove it from the list of available studies, a special *study-remove* message is sent to the server. The message is signed with the same private key that was used to sign the original study request, thus authenticating the request. Once a study has been ended, it is no longer possible to join it. The server may choose to retain it in an archive of ended studies.

If a study is removed from the server, clients will no longer display it in the list of avaialble studies. If the client had already joined the study, its status changes from *active* to *archived*, and no new data will be submitted.

This concludes the research functionality. Before we begin implementing and evaluating the system, we will take a brief look at a possible alternate, simplified protocol.

3.8 PROTOCOL VARIANTS

The protocol, as specified in this chapter, introduces some complexity in order to be able to efficiently share arbitrary data. In particular, for each share operation, the *indirect* sharing and retrieving process (described in Section 3.6.2 and Section 3.6.3, respectively) creates one key-value-pair on the server that will never be removed - the *data block* containing the actual encrypted data, uploaded under the identifier $\mathtt{id}_r$.

By not encrypting the shared data *directly* for each recipient, the system is capable of efficiently sharing larger pieces of data with a large number of recipients. The full data only has to be encrypted once, with all other encryption operations only encrypting the key and identifier, which have a constant and small size. However, fitness data like GPS tracks, heart rates, and step counts, usually has a fairly small size. Thus, the additional computational overhead incurred by encrypting the same data multiple times is in many cases negligible.

On the other hand, directly encrypting the data for all recipients and uploading it under $\mathtt{id}_{AB}$ does not result in permanent load on the server, as the recipient will delete the data from the server as soon as it was retrieved. This significantly reduces the number of key-value-pairs stored on the server, thus decreasing the size of the VI-CBF and thereby reducing the overhead incurred by transferring it to clients.

In this case, the share procedure would work like this: Alice would take her piece of data d and encrypt it for Bob, using the same IV construct as in Section 3.6.2

$$c_{AB} = \mathrm{Enc}(k_{AB}, d)$$

She would then derive the identifier and upload the data:

$$\begin{aligned} \mathtt{id}_{AB} &= h(h(k_{AB} \parallel \mathtt{ctr}_{AB})) \\ &\textsc{Store}(\mathtt{id}_{AB}, c_{AB}) \end{aligned}$$

When retrieving data, Bob will directly access the data saved under $\mathtt{id}_{AB}$, without needing to send a second GET for the second identifier. However, if Alice wants to share larger pieces of data with a larger number of people, this will significantly increase the upload time and storage load on the server.

We specify two variants of the protocol:

1. **Protocol 1 (P1)** is the protocol as previously described, using indirect sharing.
2. **Protocol 2 (P2)** is an alternative protocol, encrypting the data directly for each recipient.

It would also be possible to specify a third protocol that dynamically chooses between the two options, depending on the size of the data and the number of recipients. However, for the purpose of this thesis, we will limit ourselves to these two options.

3.9 SUMMARY

In this chapter, we have specified and motivated the design goals of our system. We have introduced an unlinkable identifier scheme, and proven a number of its important properties. Finally, we specified the processes for sharing and receiving data, and for creating and participating in research studies in the system. This concludes the design chapter. Next, we will take a closer look at how the system is implemented in our proof of concept.

4 IMPLEMENTATION

In order to evaluate the feasibility of the system, we implemented a proof of concept. It implements only a subset of the system - due to time constraints, the advanced features of the friendship revocation (cf. Section 3.5.2), and messages from researchers to study participants (cf. Section 3.7) have not been implemented. For the same reason, the connections to the server are not run through the Tor network.

In this chapter, we will discuss the proof of concept implementation. We start off with a brief overview over the Android operating system, followed by a discussion of our strategy for securely storing the shared secrets. Afterwards, we will briefly discuss how sensor data is collected. We will then take a deeper look at the implementation of the different parts of the sharing system: The server, the friend discovery, the sharing process, and the research functionality. Finally, we will discuss a simulator we developed for the evaluation, which will be the topic of the next chapter.

4.1 THE ANDROID OPERATING SYSTEM

Our proof of concept application was developed for the Android platform. In this section, we will give a brief overview about the security model used by the Android operating system. We will also dicuss the available cryptographic libraries, communication channels and sensors.

4.1.1 *Security Model*

The security model of Android consists of two parts: The permissions, and the sandboxing system.[1] Both are integrated into and enforced by the operating system.

The permission system forces developers to declare which critical functions of the device they want to use. They include access to the internet, the built-in camera, microphone, and storage, and many other functions. Up to the release of Android 6, the user was presented with a list of permissions requested by the app, and had the option to cancel the installation. However, the process was *all or nothing*, it was not possible to reject specific permissions.

1 See `https://developer.android.com/guide/topics/security/permissions.html`, last visited March 23rd, 2016

Our proof of concept still uses the old permission system

With Android 6, the permission model was changed to *runtime permissions*: When the app wants to use a protected function for the first time, the user is asked if this is acceptable, and has the option to reject the request. Apps should be engineered to handle rejections of non-critical functions and work with a reduced set of functions in that case.

The sandboxing system provides another layer of security: Each application is run in its own sandbox, without access to the data of other applications. This is enforced by both the operating system and the underlying kernel.

4.1.2 *Cryptographic Libraries*

Android has built-in support for cryptographic functions. The implementation is based on the Open Source *BouncyCastle* library,[2] using the standard Java Cryptography Architecture (JCA) Application Programming Interfaces (APIs).

However, the version of BouncyCastle shipped with Android does not offer the *lightweight API* offered by the standard BouncyCastle, which provides a more convenient and compact syntax for cryptographic operations. Additionally, the library is only updated with system updates, so many devices are running an outdated version of the library, thus lacking the latest features.[3]

To ensure that the latest version of the library is available, we need to include the *SpongyCastle* library[4] in the application. SpongyCastle is a fork of BouncyCastle which is made compatible with Android and has been renamed to prevent namespace conflicts with the preinstalled version. As the library is compiled into the Android app itself, the developer has full control over the used version and available features.

The drawback of this solution is that each application needs to ship its own copy of SpongyCastle, leading to multiple redundant versions of the library being saved on the user's device.

4.1.3 *Communication Channels*

Android devices typically have at least three dedicated communication channels built-in. They can connect to wireless networks using their WiFi chips, they have a mobile network connection for telephony and internet access, and they also have a Bluetooth chip for medium- to short-range connections. Many also have a Near-Field Communication (NFC) chip that allows communication over very short distances.

2 See `https://www.bouncycastle.org`, last visited January 17th, 2016
3 See `https://code.google.com/p/android/issues/detail?id=3280`, last visited January 17th, 2016
4 See `https://rtyley.github.io/spongycastle`, last visited January 17th, 2016

There are also other, less obvious communication channels. Optical communication could be facilitated using the screen and built-in camera, and the speakers and microphone can also be used to communicate between devices. The latter is used both in both benign and malicious applications: Google's *Nearby Messages* API[5] uses it to communicate very short pieces of data between devices, but advertisers also use it to track which phones are in the vicinity of specially-crafted advertisements.[6]

4.1.4 *Sensors*

The available sensors and their quality differ on different phone types. Some sensors are built into every Android device (e.g. the accelerometer that determines how the device is moved by the user). Others may only be available on some specific phones (e.g. fingerprint or pulse sensors).

These sensors can be supplemented with specialized devices (e.g. smartwatches, fitness armbands, ...) that connect to the phone using a wireless communication channel, usually Bluetooth. The additional sensors may be made available to every application, or require a proprietary companion app.

Android offers APIs to determine which sensors are available, and to gain access to their readings. These values can then be used and saved by the application. Google also offers a special set of functions for fitness data under the name *Google Fit*.[7]

While most of its features require integration with the proprietary Google Fit service, the Google Fit Sensors API can be used without submitting data to Google Fit. However, it still requires the proprietary Google Play Services to be installed, which keeps the API from running on devices without the Google Apps package. While almost all phone vendors ship their phones with this package, some alternative Android systems like *CyanogenMod* ship without it due to licensing constraints.[8]

4.2 SECURE LOCAL STORAGE

Android offers four different kinds of local storage:

- The *preferences* store, where user preferences can be stored in a key-value-store (stored as an XML file).

5 See `https://developers.google.com/nearby/messages/overview`, last visited March 23rd, 2016

6 See `http://arstechnica.com/tech-policy/2015/11/beware-of-ads-that-use-inaudible-sound-to-link-your-phone-tv-tablet-and-pc/`, last visited March 23rd, 2016

7 See `https://developers.google.com/fit/`, last visited March 10th, 2016

8 See `https://wiki.cyanogenmod.org/w/Google_Apps`, last visited March 23rd, 2016

- Storage in one or more file-based SQLite databases
- Private file storage (accessible only to the application)
- Public file storage (accessible to everyone on the device)

The first three options are stored in the application directory and protected using Androids Sandboxing model, which prevents other applications from accessing the files. However, if a device is *rooted* or contains a software vulnerability that allows privilege escalation, these restrictions can no longer be enforced by the system, and any application with root privileges can access these files. Additionally, if the device has developer features enabled, an attacker can read out the application directory via the Android Debugging Bridge (ADB).

As the SQLite database contains security-critical information like cryptographic keys, it follows that it should be protected against attackers gaining access to the file in which the database is stored. The same is true for any file-based caches or other forms of file-based storage containing sensitive information.

The security properties of SQLCipher are evaluated in Section 5.1

In our implementation, the SQLite database is protected using the *SQLCipher* library.[9] SQLCipher adds a transparent encryption layer to the Android database APIs, encrypting the database and relevant temporary files. All data, including collected fitness data and data shared by friends, is saved in the encrypted database.

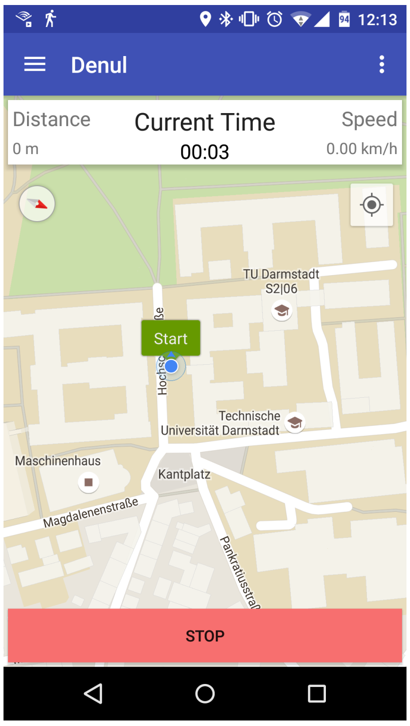
The encryption key is derived from a passphrase. In the proof of concept implementation, this passphrase is set to a constant value in the source code. This allows us to test all aspects of the application without having to enter a passphrase on every test run. However, it is unsafe for productive use and should be replaced with a passphrase provided by the user. The resulting keys can then be cached until the application is explicitly locked or the device rebooted to avoid inconveniencing the user.

4.3 COLLECTING SENSOR DATA

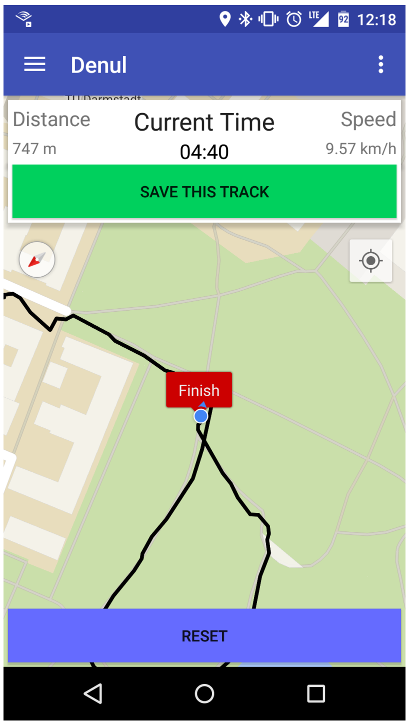
The proof of concept implementation is only capable of performing GPS tracking. This tracking is explicitly started and stopped by the user (cf. Figure 12a). While the tracking is running, the system continually monitors the position of the user and plots her path on a map, using features provided by the Google Play Services API.[10] It also calculates an average speed and the travelled distance (cf. Figure 12b).

This feature is mostly intended to perform GPS tracking of jogging or cycling routes. After a track has been recorded, the used mode of transportation (running or cycling) and a short name can be entered

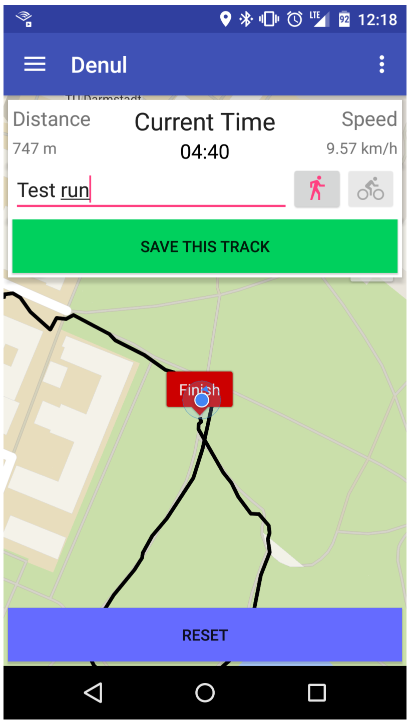
9 See `https://www.zetetic.net/sqlcipher`, last visited December 7th, 2015
10 See `https://developers.google.com/android/guides/overview`, last visited March 15th, 2016



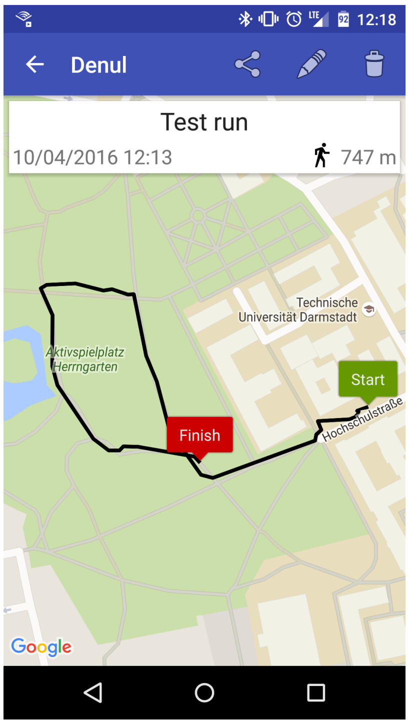
(a) Starting a run



(b) Status after the run is finished



(c) Saving the GPS track



(d) Inspecting the saved track

Figure 12: Recording and saving a GPS track

to save the track (cf. Figure 12c). Alternatively, the user can choose to discard the data. Collected routes can be inspected in the „exercise history“ window (cf. Figure 12d). From there, they can also be deleted, renamed, or shared with friends.

The application is also capable of tracking the number of steps the user takes per day using a dedicated pedometer built into the device, if available. However, due to time constraints during the implementation phase, the data is not exposed in the User Interface (UI) and cannot be shared with friends.

4.4 SERVER

The server is written in the Python[11] programming language, which offers convenient high-level functionality for server applications, thereby reducing the amount of work needed to implement the server. There are also a large number of open source libraries that provide additional functionality.

The server offers a Transmission Control Protocol (TCP) socket protected with Transport Layer Security (TLS) using a valid certificate issued by a Certification Authority (CA) for clients to connect to. The data is encoded and decoded for transmission using the Protocol Buffers library,[12] which offers consistent data encoding and serialization functionality over different programming languages with low overhead. Data is stored in an SQLite database as a key-value-pair, mapping the identifier to the encrypted data. Support for cryptographic operations is supplied by the PyCrypto library.[13]

4.4.1 *VI-CBF*

The protocol requires a Variable-Increment Counting Bloom Filter (VI-CBF) to protect the privacy of users and reduce the amount of queries that need to be sent to the server. The server maintains a VI-CBF and sends it to clients, which will use it to determine if they need to send a query to the server or not. This means that both the server and the client need a VI-CBF implementation, and these implementations need to be compatible.

As the Android client is written in Java and the Server is written in Python, two separate implementations were needed. Neither Java nor Python had an existing open source VI-CBF implementation, forcing us to write our own.

As previously discussed, the VI-CBF uses two sets of k different hash functions each. The first set is used to determine the slot in the bloom

11 See `https://python.org`, last visited January 16th, 2016
12 See `https://developers.google.com/protocol-buffers/`, last visited March 15th, 2016
13 See `https://www.dlitz.net/software/pycrypto/`, last visited March 15th, 2016

filter, and the second is used to determine the increment value. These functions were implemented using SHA1 as a base hash function h. When inserting the value x, we calculate $h(x \parallel n)$. The result is interpreted as a number and the remainder modulo the number of slots, m, indicates the slot that should be incremented in the nth step.

The nth increment is determined similarily, using $h(-n \| x)$ to derive a number. We take the remainder modulo L (as defined in Section 2.4.5), and add L to the result to obtain a value $v \in D_L$ We then increment the previously-calculated slot by v. This process is repeated for each of the k hash functions, resulting in the k pairs of slots and increments required by the VI-CBF.

As the VI-CBF is transmitted very often, achieving a space-efficient serialization format was key to improve the overall efficiency of the protocol. The serialization algorithm is described in the Appendix, Section A.1.

4.5 FRIEND DISCOVERY

In order to allow any sharing of data, cryptographic keys have to be established between the two parties. This process is embedded into the process for adding new friends. To add a new friend, both users currently have to be in physical proximity.

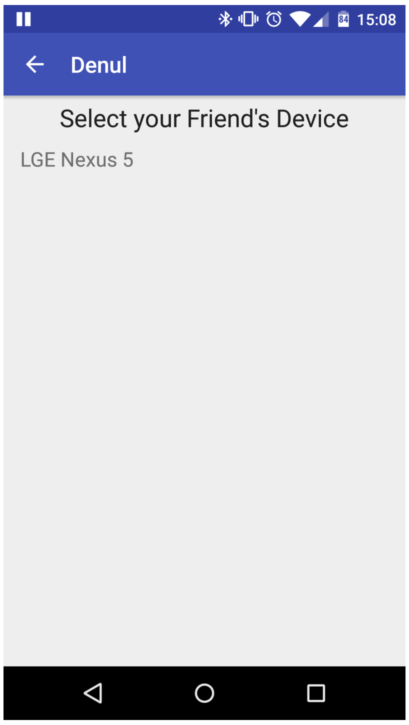
The users will establish a connection between their devices, and a key agreement protocol will be executed over this connection to generate a shared secret. This secret will be transformed into the correct format for the protocol by using a Key Derivation Function (KDF). Finally, the result will be verified by checking the fingerprints of the established key material. Each of these steps will be discussed in detail in this sections.

The process is designed to be modular, so it is possible to switch out any implementation with a functionally identical component using different algorithms without breaking the process. That way, other communication channels or protocols can be added.

4.5.1 *Connection Establishment*

The initial connection can technically be established using any technology capable of transmitting data between two devices: WiFi, Bluetooth, NFC, even optical channels like the scanning of Quick Response (QR) Codes. For the proof of concept implementation, we chose the *Google Nearby Connections API*.[14] The API offers a high level of abstraction for establishing connections between devices in the same wireless network, using multicast for device discovery.

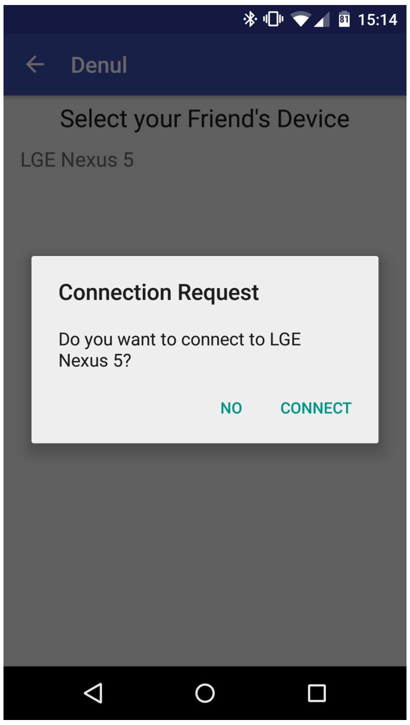
14 See `https://developers.google.com/nearby/connections/overview`, last visited December 7th, 2015



Select your Friend's Device

LGE Nexus 5

(a) List of nearby friends to connect to



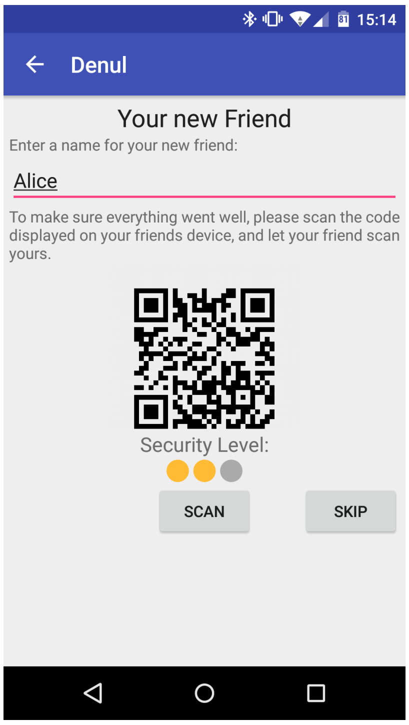
Select your Friend's Device

LGE Nexus 5

Connection Request

Do you want to connect to LGE Nexus 5?

(b) Incoming connection request from friend



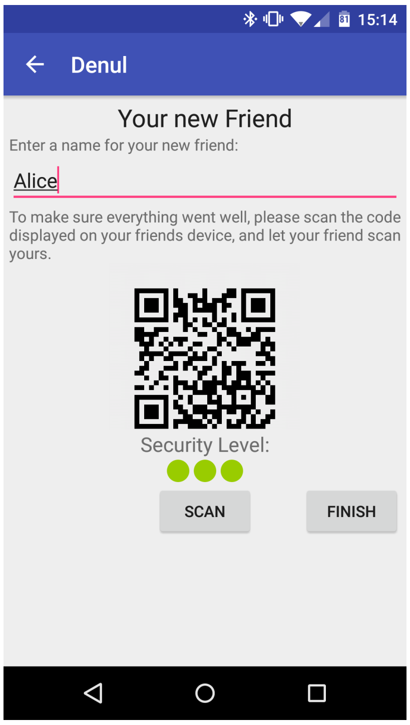
Your new Friend

Enter a name for your new friend:

Alice

To make sure everything went well, please scan the code displayed on your friends device, and let your friend scan yours.

(c) Unverified friend information



Your new Friend

Enter a name for your new friend:

Alice

To make sure everything went well, please scan the code displayed on your friends device, and let your friend scan yours.

(d) Verified friend information (after scan of QR code)

Figure 13: Friend connection establishment, key exchange and -verification in proof of concept implementation

Once two devices have discovered each other (cf. Figure 13a), they can establish a connection. The connection has to be initiated by one user (the *initiator*), and confirmed by the other (cf. Figure 13b). Once the connection has been established, messages of up to 4 KB can be transmitted. The API guarantees reliable, in-order delivery of messages.

The channel is completely open and unprotected, and the identities of the devices are not verified. Thus, a Man-in-the-Middle (MitM) attack is possible if the attacker has control over the used network. However, we will show that this does not impact the security of the scheme.

4.5.2 *Key Agreement*

Once a connection has been established, a Key Agreement protocol is executed. Due to the modular nature of the system, any key agreement protocol could be used. Performance and compatibility consideration led us to choose an Elliptic Curve Diffie-Hellman (ECDH) Key Agreement using Curve25519 [13] as the underlying elliptic curve. Curve25519 is resistant to all known attacks on other elliptic curves,[15] and not covered by any known patents.[16] It is also computationally more efficient than regular Diffie-Hellman.

Diffie-Hellman key agreement protocols result in a shared secret s between both parties. However, s does not have the correct format for our scheme, as it does not have enough bits to be split into the two keys and two counters required by the protocol while maintaining sufficient key lengths. Thus, we need to securely derive the required secrets from the shared secret s.

4.5.3 *Key Derivation*

In order to expand s into the four shared secret values k_{AB}, k_{BA}, ctr_{AB} and ctr_{BA} (see Section 3.5.1), a Key-Based Key Derivation Function (KBKDF) is used. Again, the implementation makes it easy to switch out the used KBKDF. For the proof of concept implementation, the HMAC-Based Extract-and-Expand Key Derivation Function (HKDF) [67] is used.

HKDF is widely used in protocols like the Internet Key Exchange, Version 2 (IKEv2) and considered secure. It uses a two-phase process: In the *extract*-phase, the entropy of the input value (in our case the shared secret s) is concentrated into a strong, but potentially shorter cryptographic key. The second phase, *expand*, uses this key to derive cryptographic keys of the desired length (in our case, 4 values of 256 bit each).

15 See `https://safecurves.cr.yp.to/`, last visited December 7th, 2015
16 See `https://cr.yp.to/ecdh/patents.html`, last visited December 7th, 2015

After these values have been derived, they have to be mapped to k_{AB}, k_{BA}, ctr_{AB} and ctr_{BA}. As the secrets are directional, it is important that both devices know which secrets to use when. To achieve this, the *initiator* (the device that initiated the connection) is defined to be device A, and the other device is device B. Now, the values can be consistently mapped to the keys and counters on both devices.

However, at this point, the authenticity of the values cannot be guaranteed. A MitM attacker could have intercepted the key exchange and performed her own key agreements with both parties, intending to transparently decrypt, re-encrypt and relay all future communication and thus gaining access to the data. In order to prevent this attack, the keys have to be validated.

4.5.4 *Verification*

To verify the validity of the keys, a channel that is not under the control of the attacker must be used. As previously stated, the connection using Google Nearby requires both devices to be connected to the same (possibly hostile) wireless network. This implies physical proximity of the devices, as the range of typical wireless networks is comparatively low.

As the devices themselves are assumed to be trusted, we can leverage the physical proximity to use an optical channel for key validation. Each device will compute a fingerprint over the derived keys by computing $h(k_{AB} \parallel k_{BA} \parallel ctr_{AB} \parallel ctr_{BA})$ using a cryptographic hash function like SHA-256. The resulting hash will be represented as a QR Code and displayed on the screen of both devices (cf. Figure 13c). The users can now use the built-in camera of their device to scan the QR code of their partner, at which point the fingerprints are compared.

The device will show a security level for each friend, represented as three colored dots.

- *Unverified* keys that have been established using an *untrusted channel* like the open internet are represented using one red dot. This is currently not used, as no such channel has been implemented yet.

- *Unverified* keys that have been established using a somewhat *trustworthy channel* (like a local wireless network, which is at least likely to not be compromised) are represented with two orange dots (cf. Figure 13c).

- Keys that have been successfully *verified* using a secure verification system like a QR code are shown with three green dots (cf. Figure 13d).

- Keys that have been actively validated, but whose fingerprints did *not* match (i.e. they failed validation) are displayed with three red warning signs. These keys are considered compromised. The user is strongly encouraged to repeat validation (to rule out a validation error) or delete the keys and repeat the key exchange, potentially using a different channel.

While this is strongly discouraged, the verification can be skipped during the initial key establishment. In that case, it can be performed later by selecting the unverified friend from the list of friends and scanning their QR code from their profile page.

Once the keys have been validated (or the validation explicitly skipped), the new friend and the related keys are saved to the local database.

4.6 SHARING PROCESS

Once data has been collected and at least one friend added, it becomes possible to share data. The general protocol for sharing data is described in Section 3.6. This section will describe how the protocol was implemented for the proof of concept. A high-level overview is shown in Figure 14.

4.6.1 *Data Encryption and Storage*

Data storage, as described in Section 3.6.2, is a two-stage process. To share data, the user navigates to the piece of data that should be shared. In the proof of concept, this has only been implemented for GPS traces. Once the data has been selected, it can be shared with friends by tapping the share button. The sharing interface (cf. Figure 15) supports three granularity levels:

1. **Fine**: The full GPS trace including dates and GPS coordinates.
2. **Coarse**: Only information about the distance, time and duration of the run, without GPS coordinates
3. **Very coarse**: Identical to coarse in the case of GPS traces, but can offer a third alternative for other data types that may be implemented in the future

DATA BLOCK GENERATION Once the granularity has been selected, the data is serialized into a Protocol Buffer message, which is in turn wrapped in a wrapper message. The message is then serialized into binary data.

The wrapper message is necessary to indicate the type of serialized data for deserialization

The resulting bytes are encrypted with a random key k_d using the Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM),

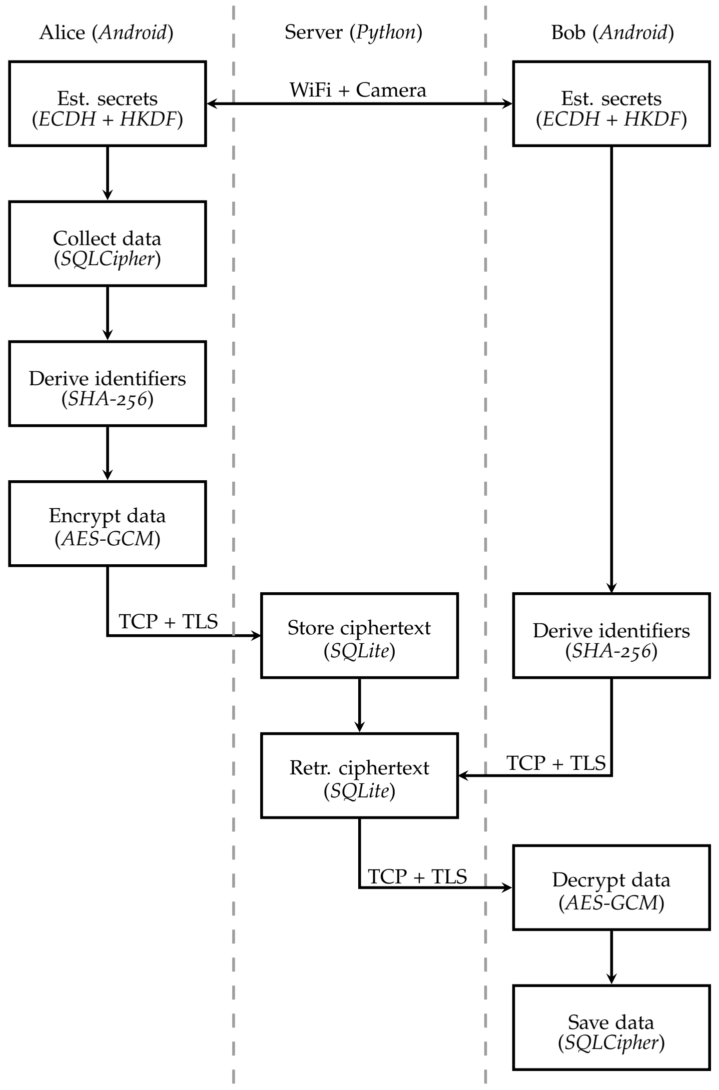

Alice (*Android*)
Server (*Python*)
Bob (*Android*)
Est. secrets (*ECDH + HKDF*)
WiFi + Camera
Est. secrets (*ECDH + HKDF*)
Collect data (*SQLCipher*)
Derive identifiers (*SHA-256*)
Encrypt data (*AES-GCM*)
TCP + TLS
Store ciphertext (*SQLite*)
Derive identifiers (*SHA-256*)
Retr. ciphertext (*SQLite*)
TCP + TLS
TCP + TLS
Decrypt data (*AES-GCM*)
Save data (*SQLCipher*)

Figure 14: High-level overview of the implementation of the sharing protocol

which provides not only confidentiality, but also integrity and authenticity with the same key. The encrypted data is then uploaded under the identifier id_d, which is derived by hashing a random, 256-bit nonce r using SHA-256. r is stored in the local database of the client and serves as a revocation token in case the data has to be removed from the server.

KEY BLOCK GENERATION In the second stage, the identifier id_d and key k_d are encrypted separately for all recipients of the data. For each recipient B, the client derives the identifier id_{AB} as described in Section 3.4.2 and encrypts the information using AES-GCM with the shared 256-bit key. AES-GCM requires an Initialization Vector (IV), which must only be used once with the same key. To achieve this, the client selects a random, 128 bit value r_{IV}, and appends it to the

AES-GCM uses the IV to derive a keystream for encryption and authentication. A reuse would, at the very least, disclose the XOR of the two plaintexts.

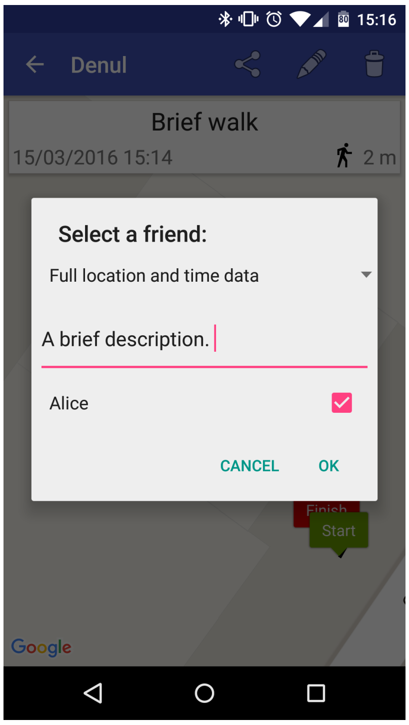


Figure 15: Selecting recipient(s), description and granularity for data sharing

current value of the counter, ctr_{AB}. The combined value is then used as the IV.

This ensures that the IV is always unique, even if the client can somehow be tricked into re-using the same counter value under the same key. It also ensures that the ciphertext is linked to the current counter value, thereby ensuring that the server cannot replace the ciphertext stored under a certain identifier with another ciphertext encrypted under the same key (e.g. a copy of an earlier ciphertext), as the decryption would fail due to an IV mismatch.

The random IV component, r_{IV}, is prepended to the ciphertext, and the result is uploaded under the identifier id_{AB}. The client will also remember which value ctr_{AB} had when the data was shared, so that it can derive the revocation token rev_{AB} later, if necessary. Afterwards, ctr_{AB} is incremented and the updated value saved in the local database.

4.6.2 *Data Retrieval and Decryption*

The data retrieval process currently has to be triggered manually by the user. Once the *refresh* button has been tapped, the client will establish a connection to the server and retrieve the VI-CBF the server maintains. It will then derive the expected next identifier for each friend, and query the VI-CBF for them, maintaining a list of all identifiers matched by the VI-CBF.

KEY BLOCK RETRIEVAL For each matched identifier, the client will send a GET request to the server. Assuming the VI-CBF did not return a false positive, the server will reply with $r_{IV} \| c_{AB}$ as generated in the previous section. The client can now reassemble the complete IV using r_{IV} and ctr_{AB}. Using the reassembled IV and the key k_{AB}, c_{AB} can be decrypted and split into id_d and k_d. As we are using AES-GCM, the successful decryption also serves to authenticate the data, and confirm that it was not tampered with by the server. Thus, id_d and k_d can be trusted to be authentic and correct.

DATA BLOCK RETRIEVAL In a second step, the client can now send a GET request for id_d, to which the server will respond by delivering c_d. This, in turn, can be decrypted using k_d (thereby also authenticating its integrity and authenticity). The resulting bytes can be deserialized into a Protocol Buffer wrapper message, containing the data message with the shared GPS track.

After the decryption was successful and the data saved into the local database, the client will send a deletion request to remove c_{AB} from the server. This serves to „clean up“ the now unnecessary data from the server, thereby reducing the load on the VI-CBF and server storage space.

FALSE POSITIVE HANDLING The proof of concept does not have specific code to handle false positives in the VI-CBF. If the download of a block fails, it will be re-queried on the next update request. This violates unlinkability, as described in Section 3.6.3. In a full implementation, the client should be able to handle this case by remembering the state of the relevant fields of the VI-CBF and only re-query the server if at least one of these fields has changed since the false positive occured. This method has been described in Section 3.6.3, but was not implemented due to time constraints.

4.6.3 *Data Deletion*

As defined in Section 3.6.4, the deletion procedure differs depending on who performs the deletion. In both cases, the client sends a DELETE message with the correct authenticator (rev_{AB} when deleting id_{AB}, r when deleting id_d, as defined in Section 3.6.4). The server will authenticate the revocation token by checking if it hashes to the correct identifier, and, if correct, delete the data and send a confirmation message.

If the *recipient* performed the deletion, the process is finished at this point. If the *sender* performed the deletion (e.g. because she has changed her mind and no longer wants to share the data), she has to upload a placeholder value under the same identifier (the reasons have been outlined in Section 3.6.4). This happens through a regular STORE message, storing the magic byte `0x42` under the identifier that has just been deleted.

The placeholder value indicates to the recipient that data has been deleted and that she should continue with the next identifier. The implementation will not give any visible indication about this deletion to the recipient.

4.7 RESEARCH FUNCTIONALITY

The research functions need dedicated functionality in the client and server, and also required writing a dedicated client that allows researchers to manage their studies and retrieve the results. The research client was written in Java, using *BouncyCastle*[17] and the JCA to provide the required cryptographic functions. All interactions with the research client happen using a text-based UI.

Many of the functions required by the research client were adapted from the Android client, as there are only minor differences between the two platforms. The database backend had to be re-implemented using *SQLite-JDBC*,[18] as the old implementation was Android-specific.

The differences include logging, database access, and some aspects of the cryptography APIs

As there is no version of SQLCipher for regular Java, the database is currently saved without encryption. An implementation for real-world use should add a layer of encryption to ensure the confidentiality of participant data and cryptographic keys.

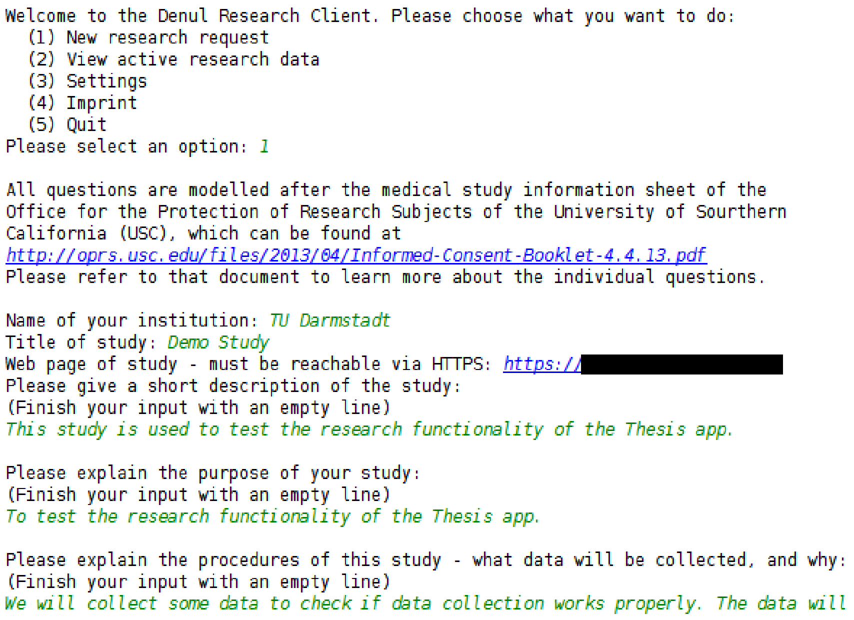
In the following sections, we will discuss the processes for creating and registering studies, retrieving and joining them with the Android client, uploading, retrieving and exporting study data, and terminating studies.

4.7.1 *Study Creation and Registration*

The creation of new studies follows the process outlined in Section 3.7.1. Figure 16a shows part of the study creation workflow in the client.

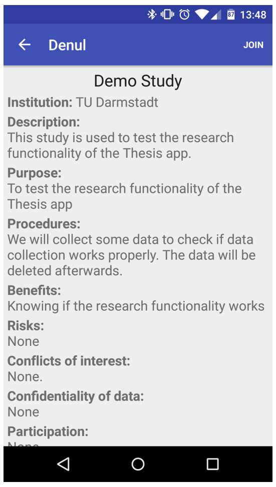
STUDY CREATION The information required to create a new study is modelled after the *informed consent* sheet of the Office for the Protection of Research Subjects (OPRS) of the University of Southern California (USC),[19] which includes questions about the use of the data, how it will be protected, and potential conflicts of interests of the researchers. The form aims to give research subjects enough infor-

17 See `https://www.bouncycastle.org/`, last visited March 13th, 2016
18 See `https://github.com/xerial/sqlite-jdbc`, last visited March 13th, 2016
19 See `https://oprs.usc.edu/files/2013/04/Informed-Consent-Booklet-4.4.13.pdf`, last visited March 13th, 2016



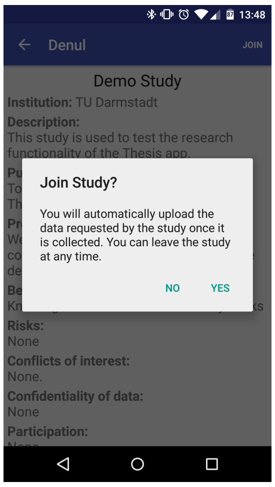
```
Welcome to the Denul Research Client. Please choose what you want to do:
  (1) New research request
  (2) View active research data
  (3) Settings
  (4) Imprint
  (5) Quit
Please select an option: 1

All questions are modelled after the medical study information sheet of the
Office for the Protection of Research Subjects of the University of Southern
California (USC), which can be found at
http://oprs.usc.edu/files/2013/04/Informed-Consent-Booklet-4.4.13.pdf
Please refer to that document to learn more about the individual questions.

Name of your institution: TU Darmstadt
Title of study: Demo Study
Web page of study - must be reachable via HTTPS: https://[redacted]
Please give a short description of the study:
(Finish your input with an empty line)
This study is used to test the research functionality of the Thesis app.

Please explain the purpose of your study:
(Finish your input with an empty line)
To test the research functionality of the Thesis app.

Please explain the procedures of this study - what data will be collected, and why:
(Finish your input with an empty line)
We will collect some data to check if data collection works properly. The data will
```

(a) Creating a study in the client



Demo Study
Institution: TU Darmstadt
Description:
This study is used to test the research functionality of the Thesis app.
Purpose:
To test the research functionality of the Thesis app
Procedures:
We will collect some data to check if data collection works properly. The data will be deleted afterwards.
Benefits:
Knowing if the research functionality works
Risks:
None
Conflicts of interest:
None.
Confidentiality of data:
None
Participation:

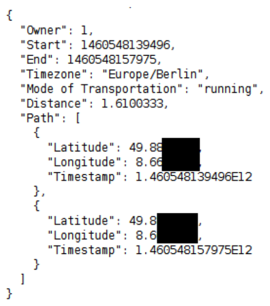
(b) Inspecting a study



Join Study?

You will automatically upload the data requested by the study once it is collected. You can leave the study at any time.

(c) Joining a study



```
{
  "Owner": 1,
  "Start": 1460548139496,
  "End": 1460548157975,
  "Timezone": "Europe/Berlin",
  "Mode of Transportation": "running",
  "Distance": 1.6100333,
  "Path": [
    {
      "Latitude": 49.88[redacted],
      "Longitude": 8.66[redacted],
      "Timestamp": 1.460548139496E12
    },
    {
      "Latitude": 49.8[redacted],
      "Longitude": 8.6[redacted],
      "Timestamp": 1.460548157975E12
    }
  ]
}
```

(d) Inspecting the data

Figure 16: Study creation, join, and data inspection

mation to make an informed decision about their participation in the study. Our client also requires the researcher to enter a TLS-protected URL containing further information about the study.

During the creation of the study, the researcher is also asked which data exactly she is interested in, and with which granularity. In the proof of concept, only GPS tracks are supported, but the system can be extended to cover additional data types. The system supports choosing more than one type of data per study.

The client needs to choose a *queue identifier* that will be used by study participants to upload their initial *study-join* message. This identifier is chosen by generating 16 random bytes. While this does not ensure uniqueness, the probability of colliding with another study

With n registered studies, the probability of a collision is $n/2^{128}$

is negligible, as there will only be a small number of active studies at any given time.

Over the course of the study creation, an RSA keypair and an ECDH key exchange using Curve25519 are generated. The RSA keypair is used to sign the study, and will later be used by participants to verify the authenticity of the study and encrypt their initial *study-join* message. The ECDH keypair will be used to generate the shared secrets as described in Section 4.5.2.

The RSA public key needs to be authenticated in order to establish a chain of trust between the researcher and the client verifying the study. Otherwise, a malicious server could replace parts of the study request and sign it with a new public key. The software offers three different methods to provide authentication data about the public key:

1. *File-based authentication*: The researcher uploads a file containing a hash of the public key to a location relative to the informational URL provided earlier.

2. *Meta-tag-based authentication*: If adding a file to that location is not possible, the researcher can also add a HTML `<meta>`-tag to the information site containing the hash of the public key.

3. *DNS-based authentication*: Finally, the researcher can add a `TXT` record containing the hash to the Domain Name System (DNS) data of the domain.

In the first two cases, the authentication ensures that the researcher can change the contents of the website she has provided. The trust in the authenticity of the hash comes from the CA system used to authenticate HTTPS connections.

The third case provides a weaker trust, as DNS records are almost never authenticated and could thus easily be spoofed by an active attacker with access to the connections of the client performing the verification. However, while weak, the guarantees it provides are better than having no authentication at all, as they at least require an active attacker that can modify DNS responses to the client.

The client will verify that the authentication data has been correctly provided using the selected method before signing the finished study request using RSA with PKCS#1 (cf. RFC 3447 [65]). The final, signed study request is then registered with the server.

STUDY REGISTRATION The completed, signed study request is uploaded to the server in a special *Study-Create* message. Before performing any other actions, the server will verify the signature on the study request to make sure it has been transmitted correctly. Note that the server will *not* verify the authenticity of the public key. As the server is not trusted either way, there is no benefit from performing this verification, and we would like to keep the server as lean as possible to increase performance and reduce the attack surface.

After verifying the signature, the server will check if the chosen *queue identifier* is available. If the queue identifier was not available, the researcher will receive an error message. If it is available, the study will be saved to the database and the queue identifier assigned to it, and the researcher will receive a confirmation.

4.7.2 *Study Retrieval and Participation*

Once a study has been successfully registered with the server, it will offer the study to all users of the system.

STUDY RETRIEVAL At the moment, users have to explicitly request a list of studies from the server by going to the study list and tapping the refresh button. The app will then connect to the server and retrieve the list of active studies, which will then be compared to the local copy of the list. Any changes, like new studies becoming available or old studies ending, will be added to the database.

The user will then be presented with the updated list of studies, and will be able to get further information by selecting one of them. The details of the study will display the data entered by the researcher, including the requested data and granularity (cf. Figure 16b).

JOINING A STUDY If the user chooses to join a study (cf. Figure 16c), the app will automatically verify the authenticity of the public key and check the signature. If the authenticity of the key cannot be confirmed, a warning will be shown, with the option to proceed anyway. However, if the signature verification fails, the user will be unable to proceed.

Once all verifications have taken place, the app will extract the public ECDH key from the study request and generate its own keypair to perform the key exchange. The key exchange and -expansion are performed as described in Section 4.5.3, with the researcher counting as the *initiator*.

Afterwards, the app will add its own public key to a *study-join* message, which will be encrypted using RSA-OAEP with SHA256 (cf. RFC 3447 [65]) and the public RSA key of the researcher, and uploaded with a normal STORE command to the *queue identifier* specified by the study request.

RETRIEVING JOIN MESSAGES Queue identifiers work differently from regular identifiers: They use a different number of bits (128 instead of 256), making them distinguishable to the server. They are also no simple key-value-pairs, but instead function as queues holding multiple entries until they are retrieved. The queues are linked to a specific study and the research client has to authenticate itself.

In the proof of concept, this authentication is achieved by signing the request for the data saved in that queue. This method is susceptible to a *replay attack* and should be replaced with a secure authentication protocol for a system intended for production use.

Once the new messages have been retrieved, they can be decrypted and any included key exchanges completed to derive the shared secrets required by the normal sharing protocol. All retrieved messages are automatically deleted from the server.

4.7.3 *Study Data Management*

Once a client has joined a study, it starts automatically providing data to it. Whenever a piece of data requested by a study is collected, it is automatically shared with the researcher. This automatic approach reduces the friction of participating in a study.

DATA STORAGE AND RETRIEVAL Sharing data with a researcher uses the same mechanisms as sharing it with regular friends. The process is described in Section 4.6.1 and 4.6.2.

DATA INSPECTION The researcher will save all received data into her local database. There, she is also able to inspect the data, which will be displayed in a JSON-like format (cf. Figure 16d). The proof of concept does not offer a function to save the data to a file, but could easily be extended to provide this functionality, enabling analysis of the data using external tools.

4.7.4 *Study Termination*

After a study is finished, it no longer makes sense to offer it to users and collect data. Instead, the researcher can delete the study from the server using a signed deletion request. The server will verify the signature and delete the study.

Upon the next refresh of the study list, the study will no longer be displayed to users unless they are already participating in it. At the moment, the application will not do anything to indicate the end of a study to participating users and they will keep sharing data to it. This is a limitation of the proof of concept implementation due to time constraints. An implementation meant for productive use should inform the user and stop sharing data with the study.

4.8 PROTOCOL SIMULATOR

In order to evaluate the performance of the system in a large deployment, we needed a simulator to model the evolution of a population of users, their relations, and their sharing behaviour. This allows us insight into how the different parts of the system perform over time and with different user counts. It also allows us to simulate large deployments and longer timeframes, which would be impossible with a real-world field trial.

The simulator was written in the *Python*[20] programming language, using the *SimPy*[21] simulation framework. The performance was increased by using an optimized version of python called *pypy*.[22] The resulting data was evaluated using python scripts and the *numpy*[23] scientific computing library.

We decided to use SimPy instead of another simulation system like *NetLogo* [117] or *LUNES* [33] because those are either written in programming languages we are not familiar with, or require a graphical user interface, which would prevent us from running the simulation on servers without a screen.

The simulator is first initialized with a starting population of users. Afterwards, the population is evolved in steps, during which new users may join the system, old users may leave it, and active users may share data with their friends. The individual steps are discussed in the following paragraphs.

4.8.1 *Initial Population Generation*

Social network usually also have the „small-world“ property, but this does not influence the simulation.

Previous work (e.g. Li *et al.* [74], Mislove *et al.* [84], and Ma *et al.* [77]) has shown that the degrees (i.e. number of friends) of users in social networks usually follows a *power law* [103]. This means that a large number of users have a small number of friends, while a small number of users have a large number of friends (see Figure 17 for an example distribution).

Networks that exhibit this behaviour are called *scale-free* networks [73]. These networks can be generated using the *preferential attachment* algorithm proposed by Barabási *et al.* [3] (originally discovered under the name *cumulative advantage* by Derek de Solla Price [94]).

In this algorithm, new nodes (i.e. users) that are being added to the graph (i.e. joining the system) are bidirectionally connected to a constant number of other nodes (i.e. their friends). The other nodes are chosen from the existing population depending on their existing degree d, with higher degrees corresponding to a higher probabil-

20 See `https://python.org`, last visited January 16th, 2016
21 See `https://simpy.readthedocs.org`, last visited January 16th, 2016
22 See `http://pypy.org/` last visited March 13th, 2016
23 See `http://www.numpy.org/`, last visited March 13th, 2016

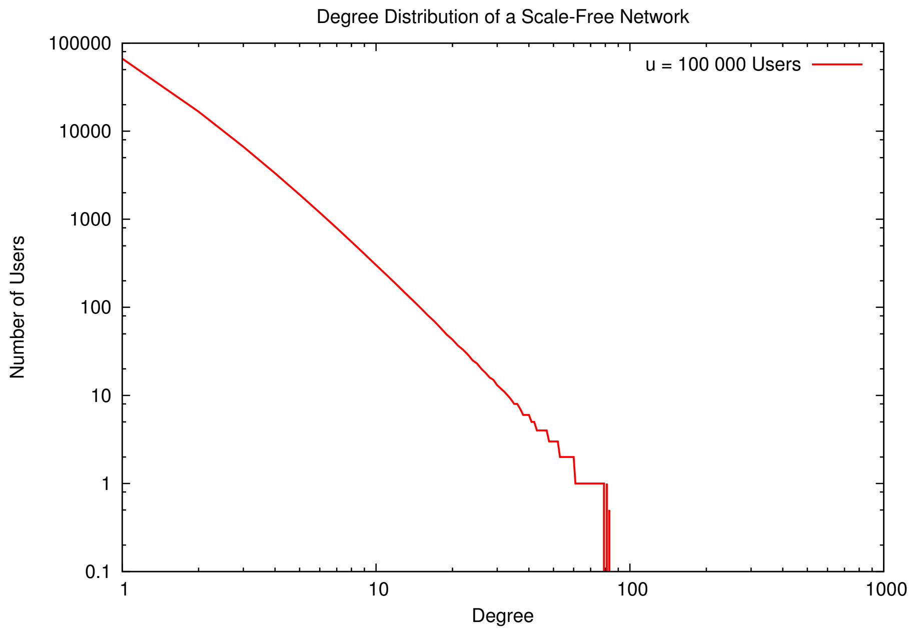


Figure 17: Example degree distribution in a scale-free network with 100 000 nodes, on a log-log scale

ity of the node being chosen. This leads to nodes with high degrees gaining even more connections, while nodes with low degrees are unlikely to significantly increase their degree, resulting in many nodes with a low degree, and few nodes with a high degree (i.e. a scale-free network).

The simulator is initialized with a scale-free network containing a predefined number u of users, generated using the preferential attachment algorithm. The resulting network is then used as the starting point for the simulation.

4.8.2 *Network Evolution*

The simulation proceeds in steps, where each step represents one day in the real world. On each day, some existing users will leave the system, and new users will join it, thereby simulating the normal *churn* any service experiences.

USER ARRIVAL The simulator assumes that the user base of the service is growing over time, as this is common for popular services. Thus, each step, it will add a random number of new users, between 1 and 5 percent of the initial user count. This results in a growth representing a new service that is quickly gaining new users and rapidly expanding its user base, which is the goal of every company and usually the point at which any scaling problems of their systems are discovered.

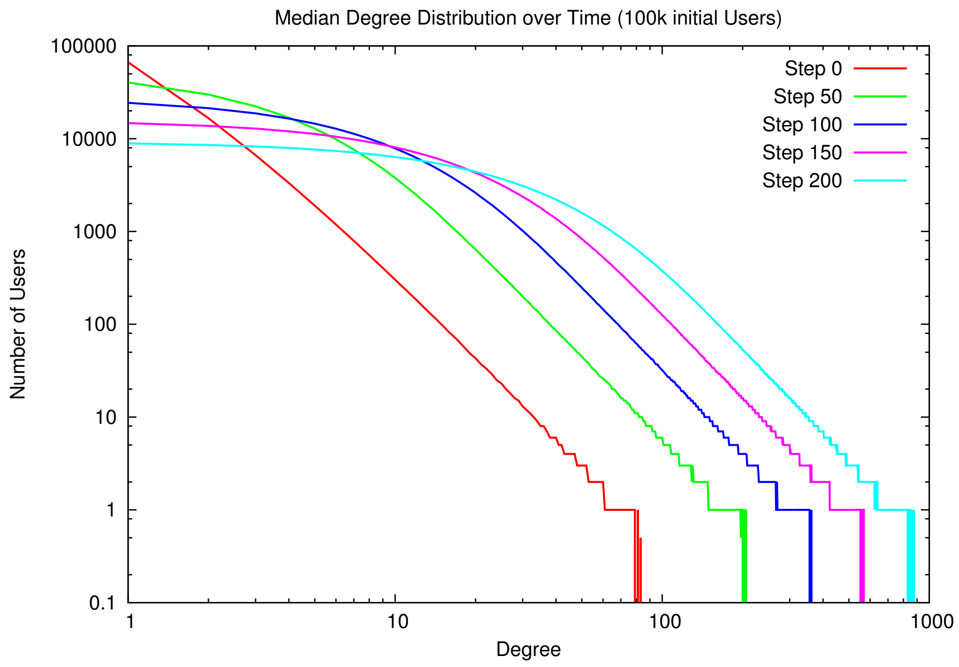


Figure 18: Development of the friend degree distribution over time (100 000 initial users)

The new users are added using an adapted version of the preferential attachment algorithm discussed in the previous section: They will choose a user uniformly at random from the current population of active users and check the number of friends this user has. Afterwards, they will select the same number of friends for themselves, using the weighted random selection from the preferential attachment algorithm. This way, new users need not necessarily start with only one or two friends, as people are often drawn to a new service because a large number of their friends use it.

USER DEPARTURE However, fast-growing companies, especially social networks, also often find users registering for a service, trying it out for a while, and then stopping to use it. This is a normal process for many companies and needs to be included in the model.

Each simulation timestep, each user has a probability of 1 percent to stop using the service. Users that stop using the service are removed from the list of candidates for the preferential attachment algorithm. This approximates the normal process of users stopping to use a service.

In reality, users are a lot more likely to leave a service shortly after joining it, and less likely to leave it if they have a large number of friends using it. However, this algorithm provides a sufficient approximation of the process, as we are not interested in individual users, but rather the behaviour of the system as a whole.

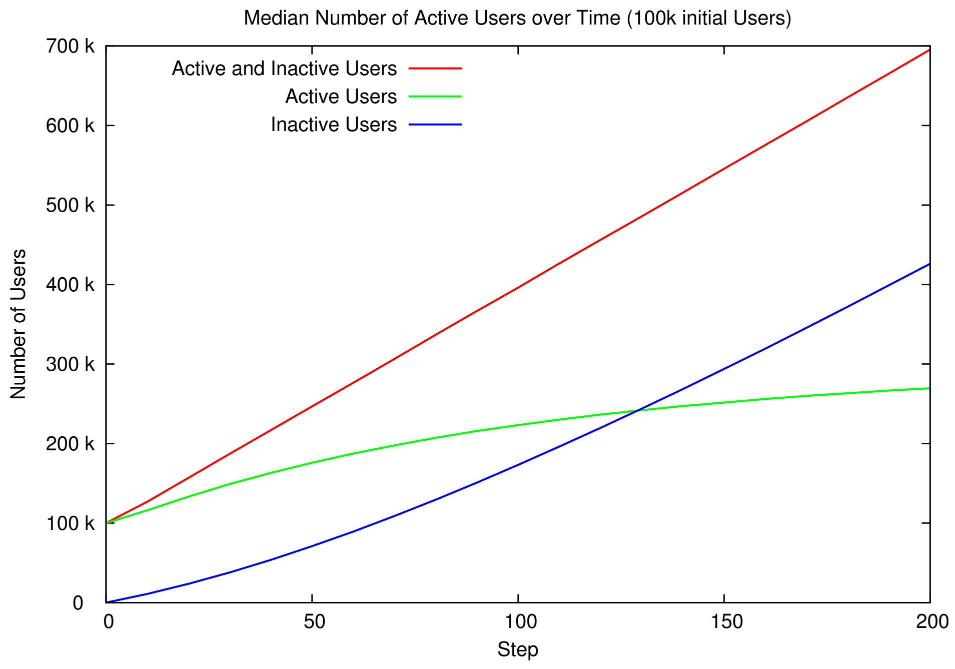


Figure 19: Development of the user count over time (100 000 initial users)

NETWORK DEVELOPMENT The resulting degree distribution at five different simulation timesteps is shown in Figure 18. The graph shows that over time, the system slowly deviates from the expected power-law distribution as new users are added and old ones leave the system. However, it remains within the variances shown by other social networks (as measured by Mislove *et al.* [84]).

Finally, Figure 19 shows the development of the number of active and inactive users over time. The number of active users shows signs of converging towards a stable value, while the number of inactive users slowly converges towards a linear growth.

This behaviour is expected, as we are adding a more-or-less constant number of users per round to the system, while the number of leaving users is linked to the number of active users and therefore grows as more users are added. This intuitively leads to the number of active users converging towards an equilibrium between new users being added and old ones removed. And since the number of users leaving the system is linked to the active user count, it will also converge towards a more-or-less constant value as the active user count converges.

4.8.3 *Sharing Behaviour*

During each step, each active user will, with a certain probability, share a piece of data with all of her friends. The probability is determined based on how long it has been since the last share operation.

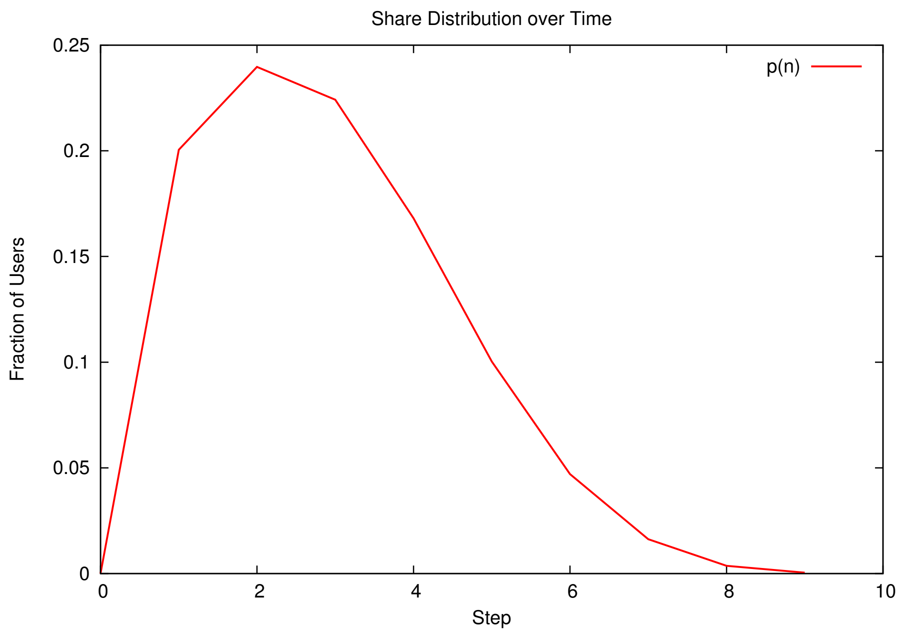


Figure 20: Probability distribution of sharing data n steps after last share

For each active user, a threshold t is computed based on the number n of steps since the last time this user shared data:

$$t = 0.1 + 0.1 * n$$

Afterwards, a random value r between 0 and 1 is chosen and compared to the threshold. If $r < t$, the user performs a share operation. Otherwise, the user does not share any data in this step.

The formula has been designed to ensure users are sharing data at most every step, and at least every 9 steps, with the majority of users sharing every second or third simulation step (cf. Figure 20). This serves to represents both frequent users (who use the system every day) and occasional users (that only log in once a week). While this does not cover all real-world use cases, it should represent the majority of users and thus give us a representative idea of how the system reacts to a large number of active users.

Data is always shared with all friends of the user. The simulator assumes that users leaving the system will not notify their friends. Thus, their friends will continue to share data with them, which will never be retrieved from the server. This is the worst-case assumption, as it puts additional load on the server and adds permanent entries to the VI-CBF distributed by the server, increasing its size.

STATISTICS COLLECTION The simulator will run for a set number of steps, collecting statistics about the system every 10 steps. These statistics include the number of active and inactive users, the number of friends each user has, the number of shares currently on the server, and other statistics. These statistics can then be aggregated over a large number of runs to determine the range of possible values over time, using normal statistical measures like mean, 1st and 3rd quartiles, and the minimum and maximum values. This process is sped up by running multiple simulations at the same time, using the Python *multiprocessing* feature, allowing us to parallelize the work over all available CPU cores.

4.8.4 *VI-CBF Parameter Approximation*

An interesting measure of the system performance is the size of the VI-CBF that has to be transmitted to each user on each connection. The size depends on the parameters of the VI-CBF: The number of counters m, the number of hash functions k, the number of entries n, and the base L of the D_L sequences that are used (cf. Section 2.4.5). In order to give a best-case estimation of the size of the bloom filter, we need to approximate the ideal parameters for a given number of entries to achieve a certain False-Positive-Rate (FPR).

Rottenstreich *et al.* show the FPR of a VI-CBF to be defined by the following formula (Source: [98, Eq. 4]):

$$\mathsf{FPR} = \left(1 - \left(1 - \frac{1}{m}\right)^{nk} - \frac{L-1}{L} * \binom{nk}{1} * \frac{1}{m} * \left(1 - \frac{1}{m}\right)^{nk-1} \right.$$
$$\left. - \frac{(L-1)(L+1)}{6L^2} * \binom{nk}{2} * \left(\frac{1}{m}\right)^2 * \left(1 - \frac{1}{m}\right)^{nk-2}\right)^k$$

They also show that a value of $L = 4$ achieves the best FPR. As the number of bloom filter entries n is given from the simulator, this leaves us with two parameters that influence the FPR: The number of counters m, and the number of hash functions k. As increasing the number of counters increases the size of the bloom filter, we would like to find the smallest value for m that can achieve the desired FPR for n entries.

APPROXIMATING IDEAL PARAMETERS For practical values of n, this solution can be solved using a *brute force* approach: We start out with a certain value of m and a fixed n and L, and calculate the FPR for a number of different values of k. If none of them are close enough to the desired FPR, the number of counters m is increased by a set amount (e.g. 1000), and the process is repeated until a solution within the acceptable deviation from the desired FPR is found. Once the solution is found, the values for m and k are returned and the process terminates.

For values of n of up to $100\,000\,000$, this process terminates within a few seconds. If even larger values should be computed, the process could be optimized by changing the step size depending on the distance from the desired FPR, thereby reducing the number of steps required to find the solution.

DERIVING VI-CBF SIZE Once the optimal parameters have been approximated, a VI-CBF with these parameters can be created and n different entries added to it. It can then be serialized to determine the expected size of the bloom filter, thereby estimating the overhead its transmission introduces into the system. This technique will be used to estimate the overhead of the system with different parameters.

4.9 SUMMARY

In this chapter, we gave an overview over the algorithms and libraries used in our proof of concept implementation. We discussed the different security features of the Android OS and our application, and we presented the simulator we designed for the evaluation. This concludes our discussion of the implementation. We will now move on to the evaluation of our system.

5 EVALUATION

In this chapter, we will evaluate the system design and proof of concept implementation. First, we will discuss the security of the local storage system used in the proof of concept. Afterwards, we will take another look at the security of the identifier generation scheme, and discuss some remaining open issues of the scheme. Next, we will discuss the security of the remote storage protocol against the adversaries discussed in our adversary model. We conclude with an investigation of the computational and networking performance of the system.

5.1 SECURITY OF LOCAL STORAGE

Each user's device holds not only the health and fitness data itself, but also a number of cryptographic secrets that need to be protected. While we assume the user's device to be trusted in our adversary model (cf. Section 3.2), it is always better to provide as much protection as practical for critical data. Additionally, in the real world, some adversaries (e.g. malware) may be able to access files and databases, but cannot easily compromise encryption functions.

The proof-of-concept implementation stores data and secrets in an SQLite database protected with *SQLCipher*.[1] SQLCipher is widely used by companies, government agencies and open source projects, including NASA, Samsung, SAP, and the Guardian Project.[2]

A full security analysis of SQLCipher is outside the scope of this thesis, but in the following paragraphs, we will provide a brief overview over how SQLCipher provides the security goals of confidentiality, integrity and authenticity. All information is taken from the official documentation and design documents.[3]

5.1.1 *Confidentiality*

SQLCipher uses the Advanced Encryption Standard (AES) in the Cipher-Block Chaining (CBC) mode of operation with 256-bit keys. The keys are derived from a user-provided passphrase using the Password-Based Key Derivation Function 2 (PBKDF2) (cf. RFC 2898 [66]) with a random salt specific to the database. This ensures that

1 See `https://www.zetetic.net/sqlcipher`, last visited December 7th, 2015
2 See `https://www.zetetic.net/sqlcipher/about/`, last visited December 7th, 2015
3 See `https://www.zetetic.net/sqlcipher/design/`, last visited December 7th, 2015

even if two databases are encrypted using the same passphrase, they will use different encryption keys.

The SQLite database is organized in individual pages of 1024 bytes, which are encrypted separately and with individual Initialization Vectors (IVs). SQLCipher also encrypts temporary files like journals. This ensures that no confidential information is leaked during database transactions.

5.1.2 *Integrity and Authenticity*

Integrity and authenticity are provided using a Hash-Based Message Authentication Code (HMAC) with SHA-1 as the underlying hash function. The Message Authentication Code (MAC) is written on each page write and checked on each page read, and uses a key derived from the encryption key using a 2-round PBKDF2 with a salt derived from the random database salt. This ensures that both accidental (integrity) and malicious (authenticity) manipulations of the page file are detected.

5.1.3 *Conclusion*

The algorithms used by SQLCipher are, as far as we know, secure. While SHA-1 is being phased out in favor of SHA-2 due to concerns about its collision resistance (cf. Wang *et al.* [114]), HMAC using SHA-1 is still secure as HMAC does not require its hash function to be collision-resistant (cf. Bellare [10]).

To the best of our knowledge, there has not been any security audit of the SQLCipher codebase. The cryptography functions are provided by OpenSSL's widely used *libcrypto* library, which is subject to scrutiny by many users. However, we use SQLCipher in the understanding that in practice, there is no such thing as perfect security.

5.2 SECURITY OF IDENTIFIERS

The identifier generation scheme we are using (cf. Section 3.4) is critical to some of our security goals, especially unlinkability. While we have already offered proofs for some of its properties, others have not yet been examined.

In this section, we will analyze our implementation choice of using SHA-256 in place of the random oracle. We will also examine some other properties of the system, try to identify open issues, and provide some ideas on how to solve any remaining issues of the system.

5.2.1 *Implementation Choices*

The use of SHA-256 in place of the random oracle seems counterintuitive, as SHA-256 is using the Merkle-Damgård construction [31, 83], which is vulnerable to a length-extension attack (cf. Tsudik [113]), making it unsuitable for use as a random oracle. However, we argue that this does not influence the suitability of SHA-256 for use in our scheme, for two reasons.

Firstly, we do not require our identifiers to be completely indistinguishable from random data, we only require them to be *unlinkable*. For the reasons outlined in Section 3.4.4, the length extension attack does not help an adversary to link our data.

Secondly, while there is no formal proof that SHA-256 generates random-looking, unlinkable data, no attack is known that would allow an adversary to link these inputs (i.e. break *correlated-input security* [55]). Such an attack could also potentially impact other systems like hash-based Pseudo-Random Generators (PRGs) (e.g. Hash-DRBG [4]), which improves the chances of such a weakness being found by researchers if it exists.

Proofs of this nature for actual algorithms are very uncommon

While this does not give us absolute certainty about the security of the scheme, it offers a solid foundation for trust.

5.2.2 *Open Issues*

While we have shown the system to fulfill the most critical security properties, there are still some additional properties that would be beneficial, but are not provided by the current design. In this section, we will discuss some of these properties.

5.2.2.1 *Post-Compromise Security*

Post-Compromise security was first formally defined by Cohn-Gordon *et al.* [26]. Informally, they define a system to provide Post-Compromise Security (PCS) if the system can still provide a security guarantee to a sender after the secrets of the recipient have been compromised (cf. [26, Def. 1] for the full informal definition).

PROPERTIES OF PCS In the case of our identifier system, the most important properties this includes are *forward secrecy* (also named *backtracking resistance* by the National Institute of Standards and Technology (NIST)) and *prediction resistance*. NIST defines these terms for Deterministic Random Bit Generators (DRBGs), a form of PRGs, in Special Publication 800-90A [4].

SP 800-90A is now (in)famous for the backdoored Dual_EC_DRBG

- A DRBG is *backtracking-resistant* (forward-secret) if an adversary gaining access to the secret state at time T cannot use it to distinguish previously-unknown outputs of the DRBG generated at time $t < T$ from true random bitstrings. (cf. [4, pp. 24])

- A DRBG is *prediction-resistant* if compromise of the secret state at time T does not allow the adversary to distinguish outputs of the DRBG at times $t > T$ from true random bitstrings. (cf. [4, pp. 25])

Informally, *backtracking resistance* means that an adversary cannot use the secret state of the generator to reconstruct old values, while *prediction resistance* means it cannot use it to predict future values.

These definitions can be translated to our identifier scheme in the following way:

- The scheme is *backtracking-resistant* (forward-secret) if compromise of k_{AB} and $\mathtt{ctr}_{AB}$ at time T does not allow the adversary to calculate any identifiers generated at time $t < T$.
- The scheme is *prediction-resistant* if compromise of k_{AB} and $\mathtt{ctr}_{AB}$ at time T does not allow the adversary to calculate any identifiers generated at time $t > T$.

It is clear that the system does not fulfill either of those properties, as k_{AB} is constant and $\mathtt{ctr}_{AB}$ changes in a predictable way:

- The adversary can backtrack (calculate old identifiers) by decrementing $\mathtt{ctr}_{AB}$ and calculating the identifier as normal (cf. Equation 1/Section 3.4.2).
- The adversary can predict the next identifiers by incrementing $\mathtt{ctr}_{AB}$ and calculating the identifier as normal.

PROVIDING PCS The intuitive solution would be the use of a forward-secret *key ratchet* (a cryptographic key management scheme) like *OTR* [21], its asynchronous successor *Axolotl* [79], or the ratchet used in the encrypted Voice over IP (VoIP) standard *ZRTP* [118].

A good explanation and comparison of OTR and Axolotl is given in [79]

All three have in common that they do not use a single key, but continually modify („*ratchet*") the key with every message, deleting old keys that are no longer required. The new keys are derived in a one-way operation that is impractical to reverse, leading to backtracking resistance/forward secrecy. Additionally, the new keys are derived using information included in sent and received messages, which makes it impossible to predict the next key until the message has been sent (prediction resistance).

Future secrecy usually requires at least one message exchange not observed by the adversary

Finally, well-designed ratchets can provide a third property called *future secrecy* by Marlinspike *et al.* [79]. While *forward secrecy* means that the attacker cannot „go back in time" to decrypt old messages after compromising key material, *future secrecy* means that the ratchet can actually *restore* secrecy to a previously-compromised session under certain circumstances (a property of PCS).

ISSUES OF RATCHETS Adding a ratchet to our protocol could provide these properties to our identifier scheme. It would also offer the same properties for the actual data encryption, as it uses the same keys as the identifier derivation. However, it would also introduce additional complexities that would need to be taken into account. For example, message revocation (deletion of a message by the sender before the recipient received it) could remove information critical to the ratchet, leading to a desynchronization.

CONCLUSION In conclusion, providing PCS to our protocol is possible, but introduces a number of additional problems. These problems can be solved with a properly-specified protocol, however, the additional complexity this introduces is out of scope for this thesis and is left for future work.

5.2.2.2 *Multi-Device Support*

So far, we have only discussed the case when Alice and Bob have exactly one device each. However, with the rising popularity of tablets and other secondary *smart* devices, more and more users own more than one device. With an account-based system (without unlinkability), it is comparatively easy to keep two devices by the same user synchronized. However, as we are not using accounts, synchronization between multiple devices is a non-trivial task.

In our case, we would need to either find a way to reliably, but privately, synchronize all keys and counters between devices. Alternatively, we could derive shared secrets for each pair of devices and combine them all under one unified *friend* in the application, sharing data to all devices of the friend if data is shared with that friend.

Building a desktop- or web-based version would involve similar problems

SYNCHRONIZING SECRETS Synchronizing the critical values introduces a large number of potential edge cases. Care would have to be taken to ensure that the values are always in sync when new share operations occur. Additionally, this would prevent us from deleting data from the server after retrieving it, as the retrieving device would not know if the other device(s) of the user have retrieved the data yet. This would put additional load on the server.

DERIVING MULTIPLE SECRETS Deriving multiple shared secrets could lead to a large number of edge cases and trust issues. The trivial solution would be to perform all key exchanges between all devices manually. However, this would be inconvenient to the users, as this would require n^2 key exchanges for two users with n devices each. A more convenient solution would be preferable.

DELEGATED KEY EXCHANGE It would be possible to delegate the key exchanges to one device. Assume Alice has two devices A_1 and

A_2. A_1 would have to pre-generate and cache a number of Elliptic Curve Diffie-Hellman (ECDH) keypairs and transmit the public keys to A_2 in a secure manner. A_2 would do likewise and transmit to A_1.

This idea is inspired by Signal, see [80]

When A_1 performs a key exchange with Bob's device B_1, it would provide not only its own ECDH public key, but also one of the keys A_2 sent to it. Likewise, B_1 would provide any additional public keys for secondary devices B_2. The verification would then compute the hash over the secrets derived between A_1 and B_2, and also include a second hash over all received public keys.

The hashes would have to be separate, as the second hash does not match between the two devices

If the verification is successful, a Man-in-the-Middle (MitM) can be ruled out and the received public keys are assumed to be authentic. As A_1 has received public keys for all other devices of Bob as well, it can also derive shared secrets with them. The public keys can also be delivered to A_2 (over a secure, authenticated channel, and with an indicator which of A_2's public keys was sent to the other party), which can perform its own key exchanges and key derivation to compute shared secrets with B_1 and B_2. As A_1 is trusted and the channel authenticated, A_2 can be sure that the received keys are authentic.

ISSUES OF DELEGATED KEY EXCHANGES This proposal would provide a better user experience, but would still have a number of edge cases. For example, what happens if Bob sells B_2 and picks up a new device, B_3? A new key exchange would somehow have to be performed between B_3 and all devices of all friends of Bob. The additional data uploaded to the server by sharing all data with multiple devices would also further increase the overhead of the protocol.

The proposal would also have to receive a full security evaluation

CONCLUSION In conclusion, while some ideas exist on how to handle the case of users with multiple devices, none of them is easy to implement, and they all require carefully specified protocols that rule out any edge cases. As such, it is out of scope for this thesis and left for future work.

5.2.2.3 *Backup Re-Synchronization*

Another issue concerns backups. Many users perform automated backups of the apps on their devices, using tools like *Titanium Backup*. These backup tools make a snapshot of the current state of the app, including all of its data, and can later restore the application to that state.

BACKUP DESYNCHRONIZATION This poses a problem in our case, as it can lead to a desynchronization between Alice and Bob. The actions leading to this desynchronization are shown in Table 1. After step 3, the data shared by Alice is no longer on the server. After step 4, Bob no longer knows that the data ever existed, and will never

#	Action	State_A	State_B
1	B performs backup	k_{AB}, ctr_{AB}	k_{AB}, ctr_{AB}
2	A shares data	$k_{AB}, ctr_{AB} + 1$	k_{AB}, ctr_{AB}
3	B retrieves data	$k_{AB}, ctr_{AB} + 1$	$k_{AB}, ctr_{AB} + 1$
4	B restores backup	$k_{AB}, ctr_{AB} + 1$	k_{AB}, ctr_{AB}
5	A shares data	$k_{AB}, ctr_{AB} + 2$	k_{AB}, ctr_{AB}

Table 1: Sequence of events leading to desynchronization

increment ctr_{AB} to a value where he will find data shared by Alice, as he will wait for the identifier $id_{AB} = h(h(k_{AB} \parallel ctr_{AB}))$, while Alice will never reuse ctr_{AB} and instead use $ctr_{AB} + 1$ and upwards.

RESYNCHRONIZATION The best way to ensure that backups will resynchronize would be to never delete any data from the server. However, this would put a large additional load on the server and, specifically, the Variable-Increment Counting Bloom Filter (VI-CBF). We will see in Section 5.5 why this would not be a good idea.

The second intuitive solution would be to occasionally increment ctr_{AB} by a few additional steps to check if we find any valid data at higher identifiers. However, we would need to set an upper bound by how many steps we want to increment at most, and this threshold would inevitably be tuned incorrectly for some users. Additionally, there is no guarantee that Alice will even *have* sent any shares that we can find this way - maybe Alice has not shared any new data since we restored our backup.

Finally, we could introduce a method to keep track of the current value of our ctr_{AB} using the server. However, care would need to be taken to store it in a manner that would be both predictable to us (even after restoring an outdated backup) and still not provide the server with a way of linking our sessions when we update it.

CONCLUSION The problem of resynchronizing after restoring a backup has no trivial solution and would require extensions to the protocol. The problem would also be even worse if we were to use a key ratchet, as proposed in Section 5.2.2.1. In this case, even k_{AB} would change, which would prevent any re-synchronization that could be achieved by resynchronizing the counter.

Solving this problem in a secure and scalable manner is out of scope for this thesis and is left for future work.

5.3 SECURITY OF REMOTE STORAGE

After showing the security of the indentifier scheme, we are now going to analyze the security of the remote storage system. We will consider each of the adversaries specified in Section 3.2 in turn, analyzing potential attacks and how the system defends against them.

5.3.1 *Honest-but-Curious Server*

The effects of allowing an active adversary are discussed in Section 6.1.1

The most obvious potential adversary would be the central server that is used for data storage. In our adversary model, we limit the server to the *honest-but-curious*-model, which forces the server to follow the protocol correctly, but lets it try to infer as much information as possible from the protocol flow.

This type of adversary can attempt a number of attacks on different security goals. We are now going to give an overview of these attacks and show what the adversary can and cannot achieve.

DATA CONFIDENTIALITY The server can attempt to read the data uploaded to it. However, the data is encrypted using strong cryptography, which the server is incapable of breaking (according to our adversary model). Without the cryptographic keys, the server is incapable of gaining access to the encrypted data and can only determine general information, like an upper bound on the size of the plaintext.

NETWORK-LAYER LINKABILITY The adversary can attempt to link users on the network layer by checking the IPs from which they connect to it. However, we have specified in our adversary model that the protocol must be run over an anonymized connection (e.g. using *Tor* [41]), which will hide the true IP of the user. The adversary will only see the IP of the *exit node* of the anonymizer, which does not help it to link the user.

PROTOCOL-LAYER LINKABILITY - IDENTIFIERS The adversary can attempt to link users based on their protocol interactions. While the identifiers themselves are unlinkable (i.e. it is impossible to determine that $\mathtt{id}_{AB}$ and $\mathtt{id}'_{AB}$ are related if they were generated with different values of $\mathtt{ctr}_{AB}$), they are also unique - a collision between two identifiers is highly unlikely if we are using a cryptographic hash function, even taking into account the birthday paradox.

The susceptability of hash functions to the birthday paradox was evaluated by Bellare et al. [7]

Disregarding data block identifiers, which may be accessed by different people, this implies that identifiers can serve to re-identify a user if she requests the same identifier twice. We introduced the VI-CBF in Section 3.6.3 to prevent this attack. As a user will only request the same identifier twice *iff* she encountered a false positive when querying the VI-CBF, it follows that the probability of being able to use an identifier to re-identify a user is related to the False-Positive Rate (FPR) of the VI-CBF.

This FPR is a system parameter that can be changed in order to find a tradeoff between the privacy provided by the VI-CBF and the overhead its size introduces (which will be evaluated in Section 5.5). It should be chosen in a way that makes this type of linkability unlikely to succeed in a large portion of cases.

Note that, while the server determines this parameter, clients could set acceptability thresholds and refuse higher values

PROTOCOL-LAYER LINKABILITY - BLOCK ACCESS The server can attempt to link uploads by the same user. Assume Alice is sharing data with Bob and Carol. According to the protocol, she will create one *data block*, c_d, and two *key blocks*, c_{AB} and c_{AC}. These blocks will be uploaded under id_d, id_{AB} and id_{AC}, respectively.

If we use the network connection efficiently, we will upload all three items using the same connection. This allows the server to determine that they are being sent by the same person. Additionally, it can make an educated guess that one of them will be a data block and the two others key blocks, as we will typically only share one piece of data at a time. This guess can be confirmed by tracking how the blocks are accessed: Blocks that are accessed once and then deleted are most likely key blocks, as this is how they are typically used. Blocks that are accessed more than once, each time directly after an access to a (suspected) key block, and never deleted, are probably data blocks. This also allows the server to determine with how many people a data block was shared by counting the number of key blocks uploaded with it.

The intuitive counter would be to perform each upload and query over its own connection. However, this approach has several problems: Firstly, unless the server is constantly receiving a large number of messages in very short timeframes, it will be able to connect the uploads based on their time. Secondly, the server could still identify which of the uploads are key blocks and which are data blocks based on the access patterns. Finally, establishing network connections over an anonymizer like *Tor* requires some time (sometimes up to multiple seconds), so this approach would impact the performance perceived by the user.

An efficient solution for this problem has not yet been found. Intuitively, this does not significantly impact the privacy of users, as their sessions are still unlinkable - the server may know that someone shared data with two friends, but not who was involved. However, it could serve as a precondition for other attacks, one of which we will discuss next.

LINKABILITY - METADATA The server can combine multiple sources of metadata to at least make guesses about the identities of users. For example, from the previous attack, the server can know with how many users people share data. It can also log at which times these shares occured, and when they were accessed. All of this data can be aggregated and searched for patterns.

For example, if Alice goes jogging for one hour every friday at 5 pm and shares the data with Bob and Carol afterwards, the server will see a share on each friday at around 6 pm, shared with two other users. If Bob and Carol usually check their phones at 8 pm, they will retrieve the data at 8 pm, which can also be detected by the server. This could allow the server to determine with a high probability that two sets of blocks (the ones from last week and this week, for example) are related.

The exact performance of this attack depends on the number of users of the system, and on how many users have the same habits as Alice, Bob and Carol. The more they fit into the general behaviour of users, the better their anonymity, as their anonymity set (cf. Chaum [23]) is larger.

Quantifying the practical danger of this attack is not trivial. Franz *et al.* [51] propose using entropy as a metric for unlinkability against an attacker with context information. Similarily, Diaz *et al.* [38] propose metrics to determine the anonymity offered in Mix network where individuals with known social connections communicate. Due to time constraints, we leave an analysis using these methodologies for future work.

While their evaluation applies to Mix networks, it could be adapted for our case

Past experience (e.g. De Montjoye *et al.* [35] for credit card data, Narayanan *et al.* [85] for movie preferences, Krumme *et al.* [70] for consumer visitation patterns) has shown that this could be a promising avenue to attack the system, and one that is very hard to defend against. To the best of our knowledge, the problem of efficient metadata obfuscation remains unsolved for this scenario.

A related problem is the ability of adversaries to infer lower and upper bounds on the size of the plaintext from the ciphertext. This may allow the server to infer the type of data that was shared. For example, a GPS track will be measurably larger than a step counter value. This problem could be reduced by using a *length hiding* strategy (e.g. Tezcan *et al.* [110]). However, if the plaintexts have a sufficiently different length, length hiding may introduce a large overhead: step counter values would have to receive several kilobytes or even megabytes of padding to be indistinguishable from long GPS tracks.

STUDY PRIVACY The server can trivially determine how many people have joined a study, as *study-join* messages are distinguishable from regular messages because they are saved in special queues. However, the server cannot determine *who* has joined a study (due to network-layer anonymity), or *how much data* individual participants are contributing (as that data is sent using the regular data sharing strategy, which makes study submissions indistinguishable from normal data sharing).

Data related to studies with a large number of participants may still be detected, as these would result in a large number of requests from the researcher when downloading new data. Such large amounts of queries are unlikely to come from regular users, as typical users do not have thousands or tens of thousands of friends. This is, again, an issue of metadata that we cannot easily prevent. However, this does not give the server any information about *which* participant sent the data.

CONCLUSION An honest-but-curious server can attempt to break the privacy of its users using hard-to-conceal metadata. However, it is not capable of determining the *identity* of individual users, even when linking their sessions, as this identity is still protected by network-layer anonymity. It is also incapable of reading the contents of the shared data. This shows that, even with the reduced privacy under a metadata-based attack, the system still provides better privacy guarantees than current commercial services.

5.3.2 *Malicious User*

The privacy of a user may be threatened not only by the server, but also by other users. Examples of potential privacy threats include stalkers, curious employers, and criminals looking for blackmail material. This problem is exacerbated by the anonymity afforded to users - while anonymity provides privacy to legitimate users, it also protects the identity of misbehaving users and prevents their exclusion from the system. Thus, we need to ensure adequate protection against malicious users.

UPLOADING ARBITRARY DATA In our adversary model, malicious users are allowed to deviate from the protocol. This means that they can store arbitrary information on the server. A malicious user can use this to upload a large amount of data, spread over many identifiers. The server would have to increase the size of the VI-CBF to achieve the same FPR as before, thereby increasing the overhead experienced by all users.

Additionally, the adversary could use the storage of the server to store arbitrary data and make it available to others. This could be used to distribute (potentially illegal) content anonymously, as long as the identifier(s) can be anonymously distributed to the recipients.

Due to the lack of an account system, misbehaving users cannot be detected or excluded. The attack could be made more expensive, if not prevented, by using rate limiting or a *proof-of-work* system, which would require users to solve a computational puzzle before being allowed to upload data. However, the practicality (cf. Laurie *et al.* [72]) and cost (cf. Becker *et al.* [6]) of proof-of-work systems is in question. Additionally, the impact on regular users would have to be taken into account: While we want to prevent the adversary from overloading the server with data, legitimate users, even those sharing data with many friends, should not be significantly impacted.

A potential solution may be an anonymous authentication system with privacy-preserving revocation like PEREA by Tsang *et al.* [112], or its successor PERM by Au *et al.* [1] which added privacy-preserving reputation tracking. While this would introduce additional complexity into the protocol, it could be used to exclude misbehaving users without compromising the privacy of users. However, the practical impact on the performance of the system would need to be evaluated.

ENUMERATING STORED DATA A malicious client would also be able to request any piece of data from the server. The main difficulty here would be to determine which identifiers are actually available on the server. The client could use the VI-CBF to locally check if an identifier is available on the server without communicating with the server for each identifier. This would increase the performance compared to querying the server, but still requires multiple hash computations for each query, thereby slowing the adversary.

Additionally, the adversary could not use any technique more efficient than exhaustive search of the identifier space, which has 2^{256} entries, as no efficient way to invert the VI-CBF (i.e. find out which items have been saved into it) exists. Finally, the usefulness of downloading the data is questionable, as it is encrypted and the identifiers unlinkable. Much as the server, the user would have no efficient way to gain access to the data.

UPLOADING FORGED DATA Another version of uploading arbitrary data is uploading forged data that generally conforms to the protocol. Examples include sending forged GPS tracks to their friends, or even submitting incorrect data to researchers to skew their results.

These problems aren't specific to our system, but a general problem: How can you ensure that data you receive hasn't been tampered with if the person generating and sending you the data is the adversary? The literature proposes using existing infrastructure like wireless access points to certify GPS tracks (cf. Pham *et al.* [91]), but there is no general solution applicable to all types of data. This problem has to be solved through vigilance, with users and researchers alike trying to spot any implausible data.

DELETING DATA An adversary may try to delete other people's data from the server. However, this requires knowledge of the revocation token, which cannot be efficiently obtained without knowing the secrets used to generate it (cf. Section 3.4.3). Thus, the adversary is again reduced to an exhaustive search of the revocation token space, which is identical to the identifier space as they use the same hash function.

Of course, the adversary is assisted by economies of scale: If she is not intent on deleting (i.e. inverting) a *specific* (or even *all*) identifier(s), but only wants to delete *any one* identifier, she has a higher chance of finding a valid revocation token due to the larger number of targets. The computations required for this are highly parallelizable and could be distributed across a network of compromised computers. However, the computational cost would still be massive.

DETERMINING RECIPIENTS Finally, our adversary may in fact be friends with one of its victims. An example could be a jealous spouse who wants to know *who else* their partner is sharing data with. In this case, the adversary knows the identifier and key of the data block the other recipients will access, but has no way of determining who else has accessed this data. Thus, the complete list of recipients remains private, even to the recipients themselves.

The case of colluding adversaries is discussed in Section 6.1.2

5.3.3 *Malicious Researcher*

As we are offering users the possibility of submitting data to researchers in the understanding that their identity will be protected, we need to ensure that malicious researchers cannot identify participants in their studies. To some extent, distinguishing between malicious users and -researchers is unnecessary, as anyone can submit studies to the server. However, we assume that our malicious researcher is actually a legitimate researcher at some respected institution and may be running legitimate studies, but try to deanonymize participants.

In addition to any actions a regular malicious user could take, a malicious researcher can use any research-related functionality. This also means that any attack mentioned in Section 5.3.2 applies here as well, and will not be listed again.

REGISTERING STUDIES A malicious researcher can start studies that have nothing to do with their academic interest, but are used purely to satisfy their personal curiosity or for other gains. They may even submit studies that contain advertisements in order to serve those ads to people looking through the list of available studies. In this, they are no different to a malicious user, which could do the same.

As we are using an open, automated submission system for studies, this cannot really be prevented. We could switch to a system where studies are manually reviewed before being made available to users.

This could filter out any obviously bogus studies. It would place more control in the hands of the server operator(s), however, as the operators control the server, they have full control over the studies they serve either way. Other than that, user education and -vigilance are the only way to defend against such studies.

IDENTIFICATION OF PARTICIPANTS Researchers have one advantage over normal users when it comes to identifying users: Upon joining a study, the user's device will communicate with a web page specified by and under the control of the researcher who created the study, in order to retrieve the data required to verify the study (cf. Section 3.7.2). The time of this access could be correlated with the time of new participants joining the study to determine which participant accessed the website from which IP.

The only method to prevent this would be to perform all connections required for the authentication through the same anonymizer we are already using for connections with the server. This would hide the IPs of participants from the adversary, making the attack useless.

5.3.4 *Conclusion*

Our evaluation has shown that, while some attacks exist, there are no promising attacks on the anonymity. The confidentiality of the data is secured, and so is the authenticity.

We have identified two attack vectors on the unlinkability, and have shown that they are general problems without a known (efficient) solution. The same is true for attacks on the availability and trustworthiness of data, which are notoriously hard to prevent.

Overall, our solution provides stronger privacy guarantees than current commercial solutions. However, the strongest privacy guarantees are useless if the system is no longer efficient enough for productive use. This is why we will evaluate the performance of our system next.

5.4 COMPUTATIONAL PERFORMANCE

After showing the security of the system, we also need to consider its computational cost. Most of the system (e.g. collecting sensor data, saving it to the disk, serializing and deserializing it for sharing, ...) can be assumed to be efficient. Thus, we will only analyze cryptographic operations like en- or decryption and hash calculations.

5.4.1 *Cryptographic Operations*

An important part of the computational performance of a system is the number and size of cryptographic operations. Depending on the used algorithms, the computational time required for cryptographic operations may be significant. Fortunately, we are frequently using hash operations and symmetric cryptography, which are both very fast (at least compared to asymmetric cryptography, which we need to use only rarely).

FRIEND DISCOVERY The friend discovery process (cf. Section 3.5) happens only rarely and thus does not have a significant impact on the overall system performance. It uses the key agreement protocol ECDH, using Curve25519 [13] as the underlying elliptic curve for its efficiency and security.

We use ECDH instead of regular Diffie-Hellman because it is faster.

ECDH key agreements are quite efficient, and their processing time is dominated by the delay introduced by the network connection they communicate over. The resulting shared secret is expanded into the correct length using the HMAC-Based Extract-and-Expand Key Derivation Function (HKDF) [67], which uses symmetric cryptography and is also quite fast. Finally, the verification of the derived secrets uses a hash function, which is also computationally efficient.

A consumer-grade CPU can compute more than 2 million SHA-256 hashes per second.

SENDING SHARES Sending shares (cf. Section 3.6.2) requires a number of cryptographic operations. First, a random, symmetric 256 bit key k_d and a random value r must be generated. Afterwards, the data needs to be encrypted and authenticated using k_d and a symmetric cryptographic algorithm (in our case AES in Galois/Counter Mode (GCM)). The time this requires depends on the size of the data to be encrypted, but will be quite low for reasonably sized pieces of data. We also need to compute the hash of r for use as the identifier id_d. Afterwards, the encrypted data can be uploaded under that identifier, which will take one Round-Trip Time (RTT) (the time it takes to send a message to the server and receive a reply).

AES-GCM usually achieves >100 MB per second of throughput

After the encrypted data has been successfully uploaded, id_d and k_d need to be distributed to all recipients. For each recipient, this requires two hash operations to derive rev_{AB} and id_{AB}, one piece of random data for the IV, and one encryption operation using k_{AB} to encrypt the data for the recipient. After the process has finished for all n recipients, the data can be uploaded in bulk to the server, which takes only one RTT, regardless of the number of recipients, for an optimal implementation.

The proof of concept implementation takes n RTTs, as it is not optimized

If we instead use the alternative protocol proposed in Section 3.8, the results change slightly: We no longer encrypt the data once and then encrypt id_d and k_d. Instead, we encrypt the data directly for each of the n recipients and upload it under their respective identifiers. For n recipients, this requires n encryption operations on variable-length data, $2n$ hash operations (to derive the identifiers), and one RTT to upload the data.

RECEIVING SHARES To retrieve a share (cf. Section 3.6.3), the client needs to first calculate the expected identifier, id_{AB}. This requires 2 hash operations. It also needs to retrieve the VI-CBF from the server, which takes one RTT. Afterwards, the VI-CBF can be queried for id_{AB}.

VI-CBF performance will be evaluated in Section 5.4.2

If the identifier comes up positive in the VI-CBF, the client can send a GET request to the server to retrieve the data, leading to another RTT of delay. Once the data was retrieved, it can be decrypted using k_{AB} and the included IV to receive id_d and k_d. The client can then retrieve id_d and delete the data saved under id_{AB}, which (in an optimal protocol that allows sending more than one command per message) takes only one RTT. After the encrypted data was retrieved, it can be decrypted using k_d and inserted into the local database of the client.

If n friends have shared data with the user since the last retrieval, the commands can be combined, keeping the number of RTTs constant. The other operations are all multiplied with n in this case - n constant-sized decryptions, n dynamic-sized decryptions, and $2n$ hash function evaluations.

If we use the alternative protocol instead, the client needs to derive the identifier(s), retrieve and query the VI-CBF, and can then query the server for each available identifier. It can bundle a deletion request for each retrieved identifier at the same time, thereby retrieving and deleting the data from the server in one instead of two RTTs. Finally, the retrieved data can be decrypted and saved. This leaves us with $2n$ hash function evaluations, n dynamic-sized decryptions, and 2 RTTs for n retrieved shares.

STUDY CREATION The creation of research studies (cf. Section 3.7.1) happens only rarely. Additionally, the time required for any cryptographic operation is dwarfed by the time required by the user to fill out the fields of the study request. Any long-running key generation tasks can be performed in the background while the user is typing, thereby hiding the delay they introduce. Nevertheless, we will take a look at the operations the study creation requires.

The study requires two asymmetric key pairs: One ECDH keypair for the bundled key exchange data, and one Rivest, Shamir, Adleman (RSA) keypair to authenticate the study and allow encrypted study join messages. These two keypairs are generated in the background, so the multiple seconds of delay they introduce are not perceived by the user. Once the keypairs have been generated, the public key is hashed to generate the token that will later be used to authenticate the source of the study (cf. Section 3.7.2).

Once the study request has been filled out, it will be serialized and represented as a series of bytes. These bytes will then be signed with the private RSA key. The resulting signed study request will then be uploaded to the server, which requires one RTT.

STUDY JOIN To join a study (cf. Section 3.7.2), the client first needs to retrieve the list of all available studies. This list is distributed by the server, and retrieving it requires one RTT. When the user decides to join a study, the first step is to authenticate the included public key. For this purpose, the hash of the public key is compared with the verification data that can be retrieved from the chosen verification system (e.g. a website protected by Transport Layer Security (TLS)), which will require at least one RTT. If the verification succeeds, it will verify the signature on the study request using the now-authenticated public key.

If the signature is authentic, the client will generate its own ECDH keypair and perform an ECDH key agreement using the data included in the study request. It expands the shared secret using HKDF. Finally, it creates the *study join* message, includes its own ECDH public key, and encrypts the message using the public RSA key provided in the study request. The resulting data is then submitted to the server, taking another RTT.

At this point, the client has joined the study and can start submitting data. The data submission process is identical to the regular sharing process and thus has the same performance characteristics.

The researcher who created the study will periodically retrieve new *study join* messages from the server. She will decrypt them using the studies private RSA key, extract the included ECDH public key, and perform her own local key exchange using the data. She can then proceed to derive the required secrets using HKDF and start receiving shares from the new study participant using the previously-described share retrieval process.

PRACTICAL EVALUATION SETUP To get an idea about the required computation time of each operation, we performed a short evaluation using the proof of concept implementation, written in Java (cf. Chapter 4). For each frequently used function (generating a random identifier $\mathtt{id}_\mathtt{d}$, encrypting a piece of data with a random key $\mathtt{k}_\mathtt{d}$, generating an identifier $\mathtt{id}_\mathtt{AB}$ for a recipient, encrypting $\mathtt{id}_\mathtt{d}$ and $\mathtt{k}_\mathtt{d}$ for a recipient), a small test harness setting up the required inputs was created.

The function was evaluated 10 000 times with different inputs, and the evaluation time (without the setup process) was measured using the `System.nanoTime()` function. The resulting measurements were aggregated and the median, 1st and 3rd quartiles, minimum and maximum calculated.

Function	median	Q1	Q3	min	max
Generate $\mathtt{id}_r$	0.054	0.050	0.055	0.047	11.222
Generate $\mathtt{id}_{AB}$	0.048	0.046	0.050	0.043	7.321
Encrypt 1 KB	0.851	0.786	0.933	0.718	102.711
Encrypt 10 KB	3.916	3.790	4.062	3.406	98.956
Encrypt $\mathtt{id}_d \parallel k_d$	0.469	0.426	0.553	0.372	14.003

Table 2: Overview of processing time for operations in proof of concept implementation (results in ms, 10 000 samples)

The evaluation was performed on an Android device running Android 5.1.1, with a Qualcomm Snapdragon 801 processor (4x 2.5 GHz, Krait 400 architecture) and 3 GB RAM. The display was active during the test.

Many phones reduce their processor speed while the display is inactive to save energy

RESULTS While the implementation is not optimized for speed, the results, shown in Table 2, should serve as an idea of what processing time we can expect. The large maximum values are most likely related to the Java garbage collector running and/or the process being paused while Android was performing background tasks on the CPU.

As expected, the bulk of the processing time appears to be spent on encrypting the data we want to share, where it scales with the length of the input. Generating identifiers takes, on average, about 0.05 milliseconds, which makes it negligible compared to the encryption function. But even the 4 ms spent encrypting 10 KB of data is small compared to the expected delay introduced by sending the data over the network, where latencies of 50 ms and more are normal.

These results show that the computational overhead of the encryption and identifier generation are unlikely to have a noticeable impact on the user-perceived latency of the system, unless large amounts of data are shared at once.

SUMMARY An overview of the number of required operations to achieve certain tasks is shown in Table 3. It shows that computationally expensive operations (i.e. asymmetric cryptography) are used only rarely. The more frequently used functions, like sending and receiving shares, exclusively use faster, symmetric algorithms like AES, where processing times are negligible compared to the delay introduced by the network.

We also note that only local operations like encryption and identifier generation are impacted by the number of recipients - the number of RTTs is constant. As networking delays will make up the majority

Operation	AG	AC	SG	cSC	dSC	H	RTT
Friend Discovery	1	1	-	1	-	1	1
Sending n Shares (P1)	-	-	1	n	1	$1+2n$	2
Sending n Shares (P2)	-	-	-	-	n	2n	1
Recv. n Shares (P1)	-	-	-	n	n	2n	3
Recv. n Shares (P2)	-	-	-	-	n	2n	2
Study Creation	2	1	-	-	-	1	1
Study Join	1	3	-	1	-	1	3

AG = asym. key generation, AC = asym. cryptography

SG = sym. key generation, cSC = sym. cryptography on const.-sized data

dSC = sym. cryptography on dynamic-sized data, H = hash operations

RTT = required RTTs (optimal implementation)

Table 3: Overview of required cryptographic operations for Protocol 1 (P1) and 2 (P2)

of the delay in most cases, this should help increase the perceived performance of the system for users.

An evaluation using the unoptimized proof of concept implementation showed that cryptographic operations are unlikely to be a bottleneck in system performance, as the processing times are in the single-digit milliseconds. Additional optimizations of the code could reduce this even further, if necessary.

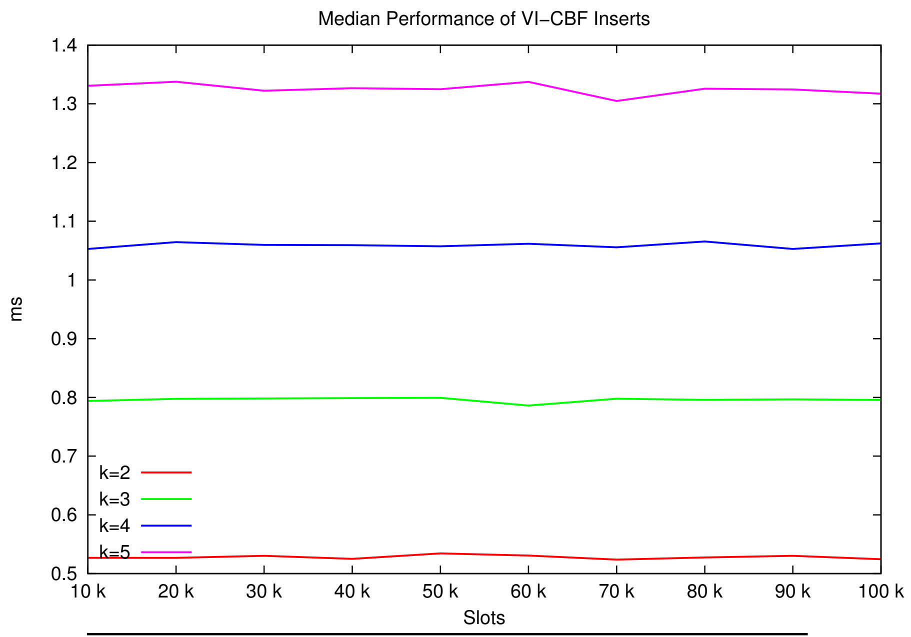
One critical part we have not yet evaluated is the performance of the underlying VI-CBF. As it influences the performance of share retrieval operations, we will evaluate it next.

5.4.2 *VI-CBF Operations*

Operations on a VI-CBF have the advantage of being more-or-less constant-time. Both inserts and queries have a strong upper bound on the number of required operations. As in the previous evaluation, we also wanted to get an idea about the required computation time for each VI-CBF operation.

While the proof of concept is not optimized, it can at least provide an upper bound on the computational time for different parameters. Optimizations could further reduce the computational time required for these operations.

The evaluation procedure was identical to the previous performance evaluation: A test harness was setting up the preconditions and the computation time of the operation was measured using `System.nanoTime()`. Each operation was repeated 10 000 times and the results aggregated into median, 1st and 3rd quartile, minimum and maximum. The eval-



k	m	median	Q1	Q3	min	max
2	10 000	0.526	0.495	0.558	0.445	21.286
2	50 000	0.534	0.505	0.560	0.446	38.649
2	100 000	0.524	0.487	0.542	0.451	77.160
3	10 000	0.793	0.747	0.822	0.672	87.004
3	50 000	0.799	0.762	0.841	0.675	30.238
3	100 000	0.795	0.753	0.832	0.681	82.977
4	10 000	1.052	0.990	1.096	0.900	34.637
4	50 000	1.057	0.994	1.097	0.904	112.510
4	100 000	1.062	1.005	1.113	0.909	140.405
5	10 000	1.330	1.259	1.398	1.135	42.717
5	50 000	1.325	1.247	1.376	1.136	50.050
5	100 000	1.317	1.243	1.359	1.135	138.064

Figure 21: Overview of processing time for VI-CBF inserts in proof of concept (Java) implementation for different hash function numbers (k) and slot counts (m) (results in ms, 10 000 samples)

uation used the Java implementation on the same Android device as before.

INSERTS For a VI-CBF with k hash functions, inserts require k hash calculations to determine the slot, k hash calculations to determine the increments, and k memory accesses to update the counters (cf. Section 2.4.5). Assuming a reasonably sized bloom filter (i.e. all counters fit into RAM and do not require swapping to access), most of the computation time will be spent on the hash functions. This also implies that the computation time will scale with the number of used hash functions, while the number of used counters will only have a negligible effect on the computational performance (as long as no swapping is required).

Figure 21 shows the results of the practical evaluation. The data confirms the prediction that the performance is dominated by the number of hash functions. Even increasing the number of slots by a factor of 10 does not significantly alter the processing time.

As before, the high maximum values are probably related to scheduling or garbage collection

The results show that VI-CBF inserts are unlikely to prove a bottleneck of the system, considering that the server also maintains a file-based database where operations are orders of magnitude slower.

QUERIES For queries, we distinguish three types:

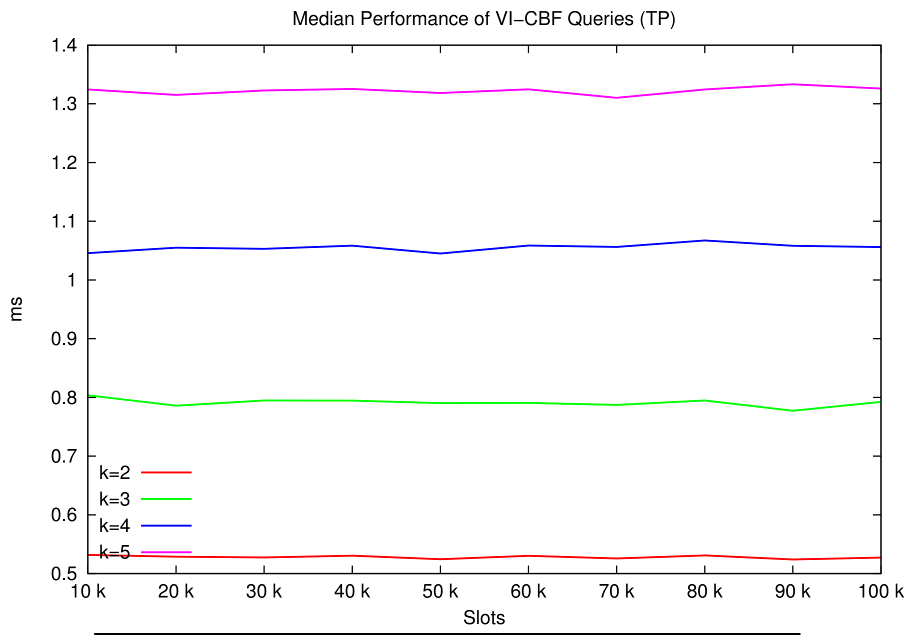
In our system, VI-CBFs will never give false negative results.

1. *true positive*: queries that correctly determine that the input value has been inserted into the VI-CBF
2. *false positive*: queries that result in a positive reply even though the value has not been inserted into the VI-CBF
3. *true negative*: queries that correctly result in a negative reply

A true positive or false positive query requires k hashes to determine the slots, k hashes to determine the increment, and k counter comparisons. True negative queries can be answered once the first verification fails, thus potentially saving a number of hash operations. Thus, negative results will, on average, have lower processing times than true or false positive results.

The practical evaluation confirms this: Figure 22 shows the results for true positive queries, while Figure 23 shows those for false positive and true negative queries. The performance for true positives is almost identical to the performance of inserts, thereby confirming the role of hash functions as the major performance factor.

The results for true negatives and false positives are related to the False-Positive-Rate (FPR) of the VI-CBF, which is in turn related to the number of entries, hash functions, and slots the bloom filter has. False positives take longer to process than true negatives, as they require the full $2k$ hash calculations, whereas true negatives can abort the computations earlier. With lower slot counts, the overall FPR of the bloom filter is higher, thereby resulting in more false positives and average computation times. As we increase the slot count, the FPR is decreased, leading to fewer false positives and an overall lower average processing time. We also note that the minimum processing time is identical (within measurement tolerances) for all parameters, showing the lower bound for the query processing time - the time it takes to calculate the first slot and increment.

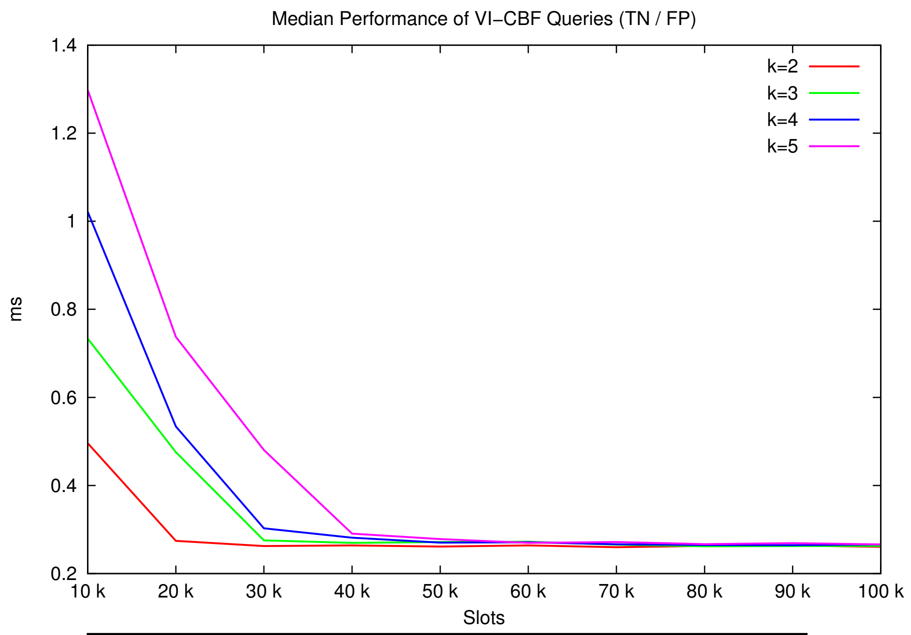


k	m	median	Q1	Q3	min	max
2	10 000	0.531	0.502	0.560	0.451	3.066
2	50 000	0.524	0.493	0.541	0.448	6.724
2	100 000	0.527	0.494	0.550	0.450	3.041
3	10 000	0.803	0.758	0.849	0.675	13.522
3	50 000	0.790	0.735	0.823	0.672	4.432
3	100 000	0.792	0.749	0.824	0.678	8.514
4	10 000	1.045	0.979	1.078	0.901	3.826
4	50 000	1.045	0.977	1.083	0.901	5.116
4	100 000	1.055	0.999	1.093	0.899	8.176
5	10 000	1.324	1.268	1.371	1.134	4.582
5	50 000	1.318	1.246	1.370	1.139	19.656
5	100 000	1.325	1.257	1.383	1.127	16.207

Figure 22: Overview of processing time for true-positive VI-CBF queries in proof of concept (Java) implementation for different hash function numbers (k) and slot counts (m) (results in ms, 10 000 samples, VI-CBF with 10 000 entries)

Overall, the processing time required to perform queries, even with larger parameters, remains negligible compared to the delay introduced by retrieving the bloom filter from the server, supporting over a thousand queries per second.

SERIALIZATION Serializing the VI-CBF for transmission to clients is by far the most expensive operation: All counters have to be con-

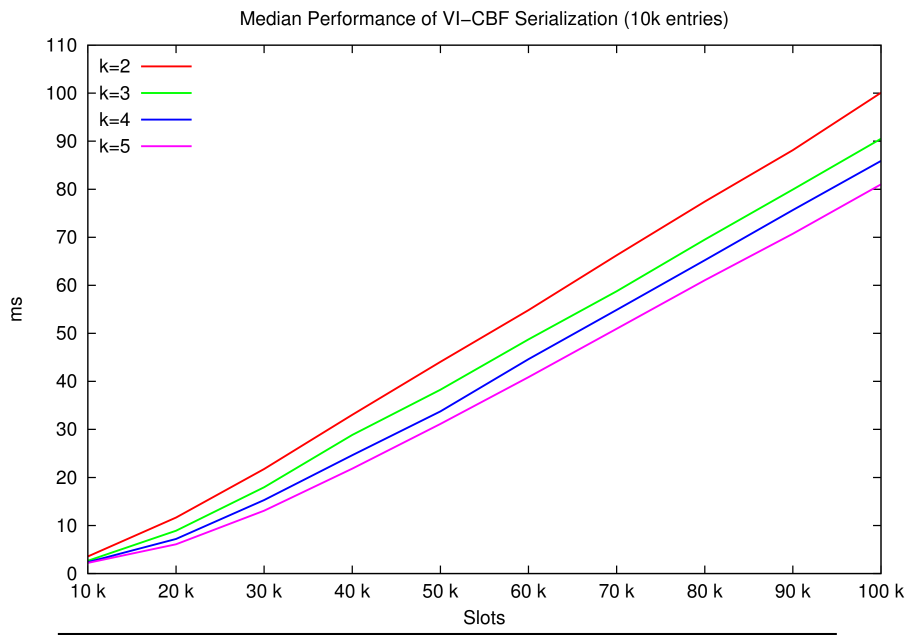


k	m	median	Q1	Q3	min	max
2	10 000	0.496	0.277	0.546	0.224	69.661
2	50 000	0.261	0.243	0.273	0.225	57.640
2	100 000	0.260	0.241	0.272	0.223	29.907
3	10 000	0.733	0.512	0.800	0.225	62.010
3	50 000	0.271	0.257	0.398	0.223	30.453
3	100 000	0.262	0.247	0.280	0.223	30.081
4	10 000	1.021	0.875	1.082	0.227	61.430
4	50 000	0.270	0.255	0.479	0.225	72.877
4	100 000	0.266	0.249	0.287	0.223	46.590
5	10 000	1.298	1.190	1.354	0.227	34.520
5	50 000	0.278	0.261	0.524	0.223	37.075
5	100 000	0.266	0.250	0.290	0.225	152.173

Figure 23: Overview of processing time for false positive and true negative VI-CBF queries in proof of concept (Java) implementation for different hash function numbers (k) and slot counts (m) (results in ms, 10 000 samples, VI-CBF with 10 000 entries)

verted to 8-bit unsigned integers (if they were in a different representation before, as is the case with the Python implementation we are using), appended to each other, and the result needs to be compressed using a compression algorithm like DEFLATE. This means that the processing time should scale approximately linearily with the number of slots. However, the result can be cached until the VI-CBF is changed, thereby reducing the practical overhead.

See Appendix A.1 for the serialization algorithm



k	m	median	Q1	Q3	min	max
2	10 000	3.561	3.543	3.582	3.514	4.700
2	50 000	44.074	43.930	44.318	43.504	98.907
2	100 000	100.088	99.896	100.437	99.625	115.614
3	10 000	2.671	2.659	2.699	2.640	4.529
3	50 000	38.278	37.934	38.407	37.598	41.003
3	100 000	90.517	90.377	90.766	90.120	98.996
4	10 000	2.369	2.341	2.437	2.309	6.562
4	50 000	33.746	33.685	33.872	33.561	48.344
4	100 000	85.906	85.674	86.102	85.395	105.426
5	10 000	2.216	2.207	2.231	2.190	4.344
5	50 000	31.158	31.079	31.298	30.257	51.238
5	100 000	81.021	80.893	81.403	80.742	102.298

Figure 24: Overview of processing time to serialize a VI-CBF with 10 000 entries in proof of concept (Python) implementation for different hash function numbers (k) and slot counts (m) (results in ms, 1 000 samples, no compression)

As the serialization takes place on the server, which is written in Python, we used the Python implementation of the VI-CBF for the evaluation. The evaluation was run on a Laptop with an Intel Core i3-3110M CPU (4x 2.4 GHz, Ivy Bridge architecture) and 8 GB RAM, running Python 2.7.6 on an Ubuntu 14.04-based Linux operating system.

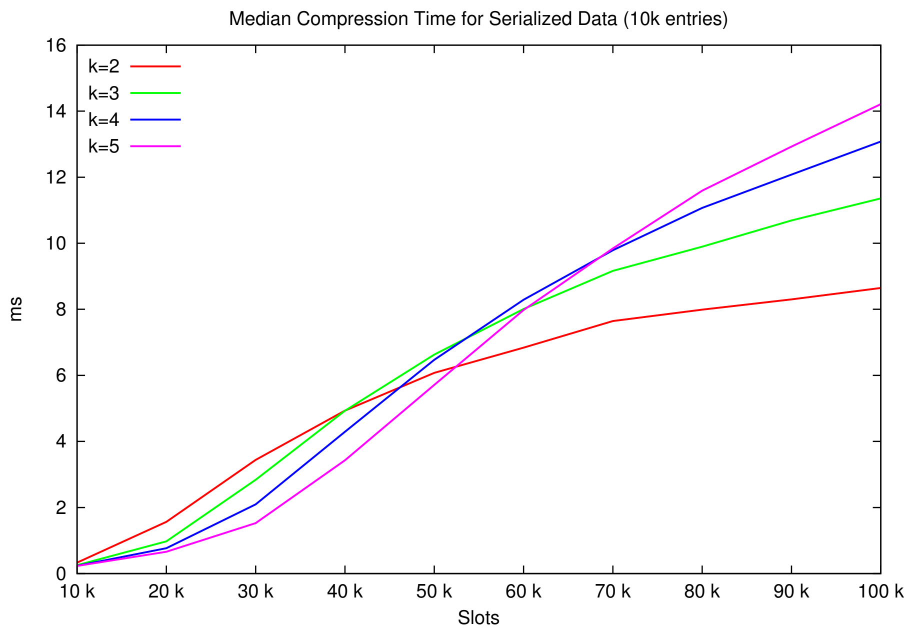


Figure 25: Median processing time required to compress serialized VI-CBF with 10 000 entries in proof of concept (Python) implementation for different hash function numbers (k) and slot counts (m) (results in ms, 1 000 samples, DEFLATE compression)

While this means that the results are not directly comparable with the other evaluation results, they should still serve to give an idea about the required time, and how it scales with changing parameters. Like the other evaluations, it uses unoptimized proof-of-concept code. The measurements did not include the compression with DEFLATE.

The results are shown in Figure 24. As expected, the processing time scales linearily with the number of slots, ranging from around 2.2 ms for 10 000 to up to 100 ms for 100 000 slots.

There are some deviations between the performance for different numbers of hash functions. Those are related to how the system is implemented: The implementation tries to save memory space by storing counters in a hashmap and leaving empty counters unset. Thus, trying to access an empty counter will throw an exception. This exception handling introduces additional overhead during serialization, where each counter is accessed once. A higher number of hash functions will lead to more counters being set for the same number of entries, meaning that fewer exceptions are thrown during serialization, reducing the overhead and improving performance.

If we add the compression step, the processing time is further increased. We use the DEFLATE compression algorithm, as implemented by the python *zlib* library, with a compression level of 6. The additional processing time required is shown in Figure 25. The results show that the growth of the serialization time seems to be dominated by the slot count. This is not unexpected, as the slot count determines the length of the serialization, and more data obviously takes longer to compress.

However, the number of hash functions also has a measureable effect. This is likely related to the number of non-zero counters in the serialization: Compression algorithms work best on uniform data, so the more counters are zero, the better and faster the compression will be. Fewer hash functions lead to fewer counters being set for the same amount of entries, which translates to a higher compression performance.

Regardless of the number of used hash functions, the compression only has a small effect on the performance, compared to the time the serialization itself requires.

SUMMARY All in all, the VI-CBF only introduces a fairly small additional computational load on the system. The processing time required for inserts is negligible compared to the network latency and database access times. Queries are also very efficient, allowing a throughput of more than 1000 queries per second on the test device, depending on the number of hash functions. The serialization requires more time, but the performance could be improved through an optimized implementation, and the results can be cached and reused until the VI-CBF changes.

However, the computational load is only part of the overhead experienced by the system. More important for the actual performance of the system is the networking overhead of the protocol, which we will investigate next.

5.5 NETWORKING PERFORMANCE

Any networked protocol needs to be evaluated in terms of the overhead it introduces. This is especially true for privacy-preserving protocols, which typically introduce additional overhead compared to a privacy-agnostic protocol.

In our case, most parts of the protocol do not introduce significant overhead.

- GET requests carry a fixed-length identifier
- GET responses carry the requested variable-length encrypted data and/or a status code
- STORE requests carry a fixed-length identifier and variable-length encrypted data
- STORE responses only carry a status code indicating success or failure

All of this data is necessary for the operation of the protocol, and most of it would be required no matter which protocol we are using.

Additionally, sharing and receiving data can usually be done within two or three RTTs (cf. Table 3).

Underlying security protocols like TLS may introduce additional RTTs.

The only component that may introduce significant overhead is the VI-CBF we need to transmit to clients, which is going to be the focus of the following evaluation.

For the evaluation, we are using the simulator described in Section 4.8 to approximate the behaviour of the real system, as this makes it easier to evaluate the system with a large number of users. We are going to evaluate the standard protocol (Protocol 1) as described in Chapter 3. Where it differs from the standard protocol, we are also going to evaluate the protocol variant described in Section 3.8 (Protocol 2).

5.5.1 *Stored Key-Value-Pairs*

One of the most important measures influencing how well the system scales is the number of key-value-pairs that are currently stored on the server. Each key-value-pair adds a new entry to the VI-CBF. This increases its size, as while adding new keys to a bloom filter does not directly increase its size, it increases the FPR and thus requires the use of larger bloom filters to achieve the target FPR. This, in turn, increases the transmission overhead experienced by clients using the system.

DEFINITIONS In this context, we are especially interested in key-value-pairs that reside on the server, but will never be deleted from it. We call these pairs *orphaned pairs*. A key-value-pair can become orphaned in one of two ways:

1. A **Type I** Orphan will be accessed, but never deleted (e.g. a *data block* created by Protocol 1, cf. Section 3.6.2).
2. A **Type II** Orphan will never be accessed and/or deleted (e.g. because the recipient has stopped using the system and will never retrieve shared destined for them).

While *Type I* orphans can be avoided by using Protocol 2 (cf Section 3.8), *Type II* orphans cannot be prevented. Additionally, *Type II* orphans cannot be reliably detected by the server because they are indistinguishable from a regular key-value-pair that has not been accessed yet, but will be in the future.

METHODOLOGY Given the importance of these records for the scaling of the system, we used the simulator described in Section 4.8 to simulate use of the system, gathering statistics about the number of stored key-value-pairs and orphaned records.

Unless specified otherwise, the system was initialized with a network of 100 000 users and run for 200 timesteps, gathering statistics

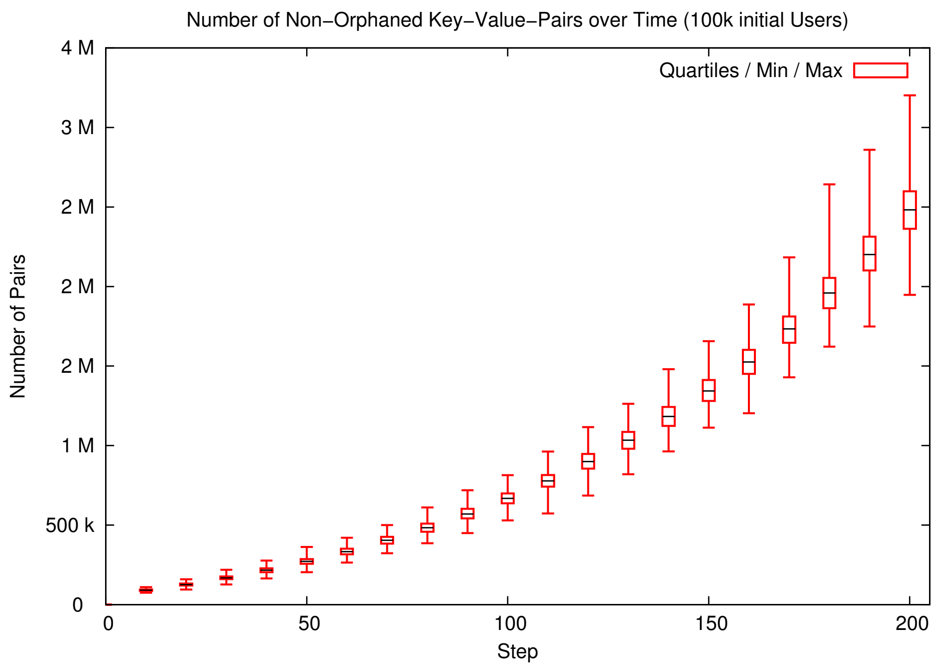


Figure 26: Number of non-orphaned records at different steps of the simulation (100 000 initial users, 200 steps, 1000 samples)

every 10 steps. The simulation was repeated 1000 times and the results aggregated, calculating mean, 1st and 3rd quartile, as well as the minimum and maximum value. The raw and aggregated simulation results are available on the DVD included with this thesis and on GitHub.[4]

RESULTS (NON-ORPHANS) Before we discuss the issue of orphans, we will take a brief look at how many non-orphaned key-value-pairs are in the system at any given time. We counted the number of key-value-pairs and subtracted the *Type I* and *Type II* orphans. The result is equivalent to the number of key-value-pairs in a hypothetical system using Protocol 2 (thereby eliminating *Type I* orphans) in which no *Type II* orphans occur (e.g. because the network is static, without any users leaving it).

The results are shown in Figure 26. They show that we can expect the number of non-orphaned records to increase over time. This is related to two factors: The total number of active users of the system is growing over time, leading to more users sharing data. Additionally, the average number of friends is slowly increasing as well due to the way new users are added (cf. Figure 18). As one key-value-pair is required for each recipient of a share, this means that the number of records on the server will increase.

4 See `https://github.com/DenulApp/data`

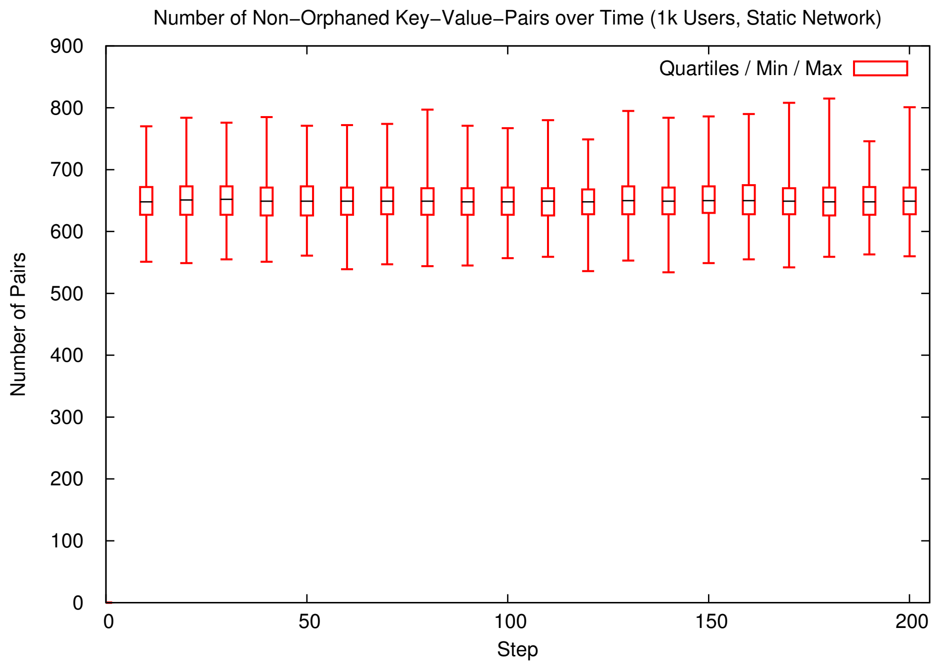


Figure 27: Number of non-orphaned records at different steps of the simulation (1 000 initial users, 200 steps, 1000 samples, static network)

This theory can be validated by running the simulation with a static network, without adding or removing any users. In this case, the number of non-orphaned records on the server stays more-or-less constant as new shares are added and old ones retrieved and removed (cf. Figure 27 for the results of a static network with 1000 users).

Now that we know what number of non-orphaned key-value-pairs we can expect, we will investigate the number of orphaned records that occur during normal use of the system.

RESULTS (ORPHANS) The number of orphans is shown in Figure 28 (combined), 29 (*Type I*) and 30 (*Type II*). They show the number of *Type I* orphans increasing almost linearily, with a very small upwards trend as the number of users increases. This is expected, as *Type I* orphans only depend on the number of performed shares, which increases by a more-or-less constant fraction of the number of active users. The number of active users in turn slowly increases over time, as shown in Section 4.8.2.

The number of *Type II* orphans increases exponentially, overtaking the *Type I* orphans somewhere between step 60 and 70. This is related to the growing number of users that have stopped using the system but keep getting data shared to them.

INFLUENCE OF USER COUNT In order to determine how the results change if we modify the number of initial users, the simulation was repeated with 1000 initial users, leaving the other parameters un-

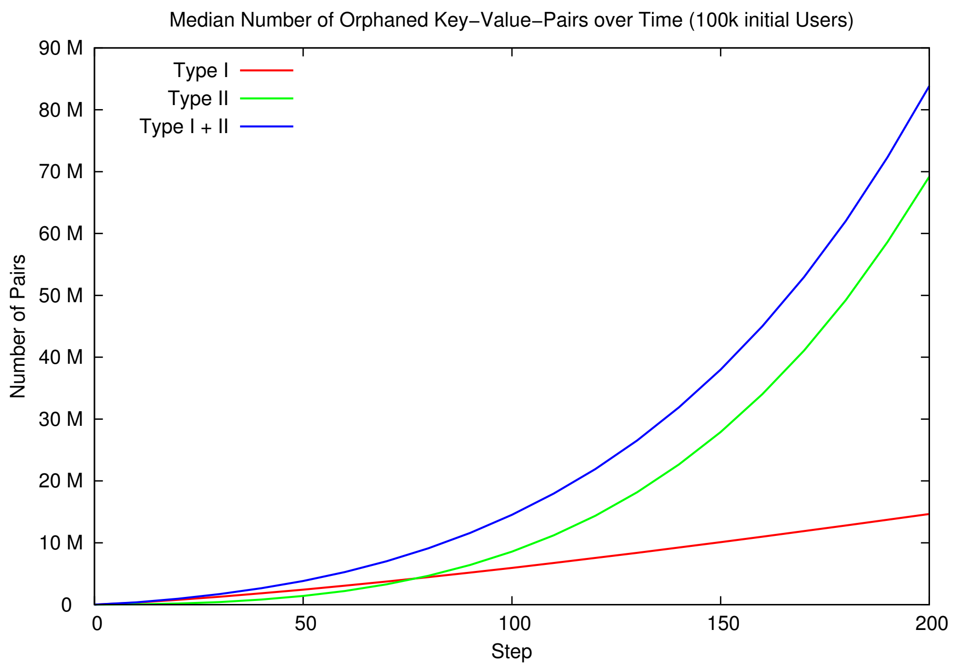


Figure 28: Median number of *Type I* and *Type II* orphans (100 000 initial users, 200 steps, 1000 samples, Protocol 1)

changed. The resulting data is shown in Figure 31. The results show that the number of orphans develops in almost exactly the same way as before, only with smaller counts. By dividing the number of initial users by 100, we also divided the number of orphans roughly by 100.

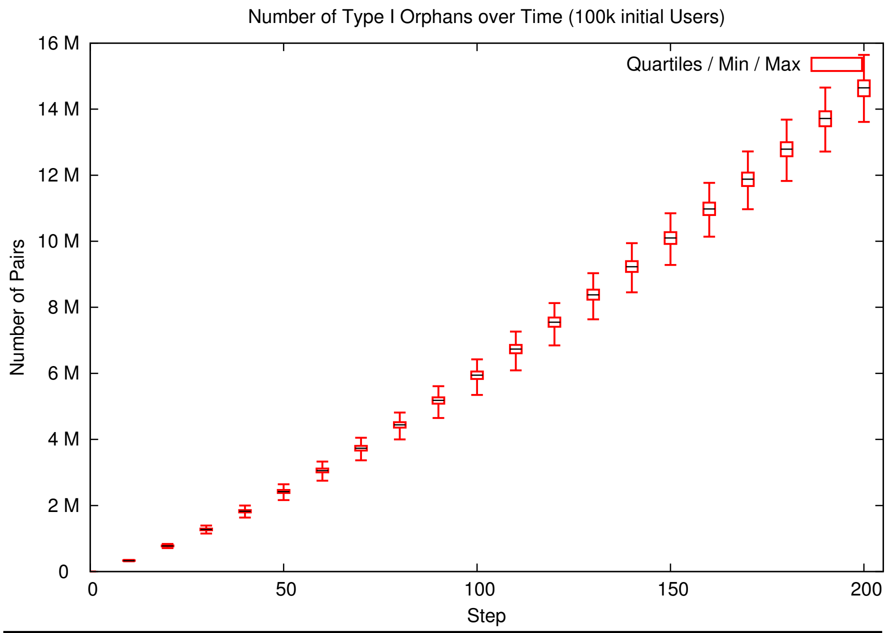
SUMMARY These results show that the development of the system, at least in the simulation, scales approximately linearily with the number of initial users. However, there is still a small deviation, which can be explained by the network characteristics of a scale-free network: A higher user count results in a higher average degree (i.e. friend count), which in turn increases the probability of having inactive friends and thereby generating *Type II* orphans.

The generally large number of orphans poses a problem for the VI-CBF, which we will investigate in the following section.

5.5.2 *VI-CBF Transmission Overhead*

The VI-CBF has to be transmitted to every client that wants to retrieve data, each time they connect to the server. This makes the size of the serialized bloom filter an important metric for the overhead of the protocol.

METHODOLOGY To determine the overhead, we used the previously generated statistics to find the number of key-value-pairs stored on the server at each recorded simulation step. We then used the



Step	median	Q1	Q3	min	max
10	331 573	326 576	336 432	309 270	351 362
50	2 420 074	2 378 103	2 468 673	2 162 176	2 642 755
100	5 944 236	5 830 275	6 057 255	5 347 213	6 426 516
150	10 100 802	9 921 270	10 275 960	9 282 759	10 848 379
200	14 647 354	14 393 899	14 872 305	13 614 062	15 642 286

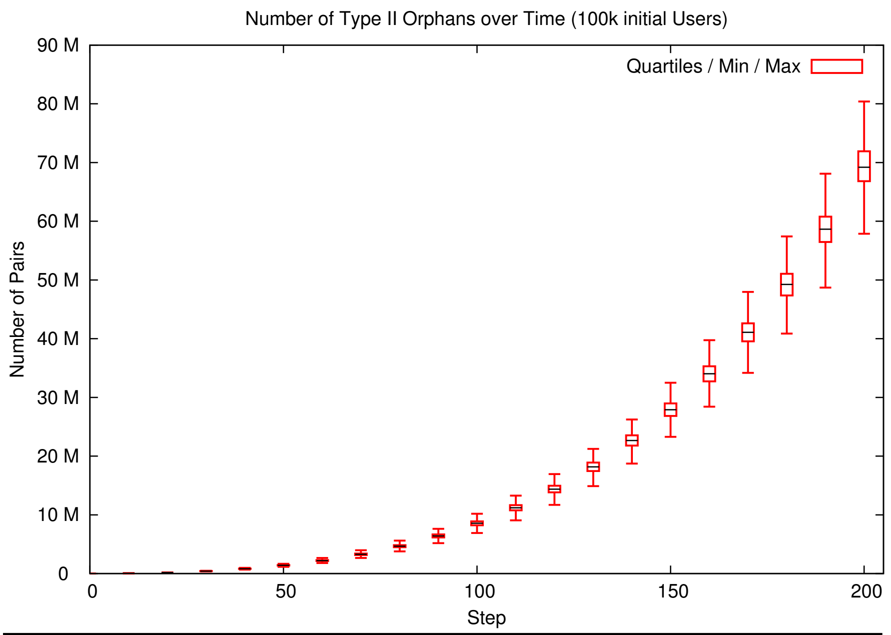
Figure 29: Number of *Type I* orphans at different steps of the simulation (100 000 initial users, 200 steps, 1000 samples, Protocol 1)

methods described in Section 4.8.4 to approximate the ideal parameters for the VI-CBF for that number of entries and a fixed FPR.

Afterwards, we created a VI-CBF with these parameters and added the specified number of entries to it before serializing it. The serialized data was then compressed using the DEFLATE algorithm with a compression level of 6. Finally, the resulting compressed and uncompressed size was determined and saved.

For some bloom filter parameters, this was not possible, as the unoptimized VI-CBF code resulted in a high memory overhead and the bloom filter no longer fit into the RAM of the device we used to run the evaluation. However, the other evaluation results showed that the ratio from uncompressed size to compressed size converged towards a fixed value, letting us approximate the remaining values. The uncompressed size can in turn be trivially determined by taking the number of slots of the VI-CBF and adding 10 bytes for the serialization headers, giving us the length of the serialized bloom filter in bytes.

This assumes 8-bit counters



Step	median	Q1	Q3	min	max
10	39 052	38 035	39 852	37 883	42 648
50	1 400 087	1 352 324	1 451 325	1 136 583	1 669 229
100	8 567 589	8 222 667	8 898 996	6 899 616	10 179 648
150	27 900 109	26 838 226	28 970 898	23 282 749	32 489 530
200	69 201 063	66 822 345	71 918 931	57 862 904	80 395 256

Figure 30: Number of *Type II* orphans at different steps of the simulation (100 000 initial users, 200 steps, 1000 samples, Protocol 1)

This process gives us an indication of which amount of overhead we have to expect for certain FPRs and entry counts.

SIMULATION RESULTS (ORPHANS ONLY) The results for 100 000 initial users and a target FPR of 0.01 (1%) are shown in Figure 32. As expected, the size of the VI-CBF follows the exponential growth of the number of orphans, quickly growing into the tens and hundreds of megabytes, even after compression. Switching to Protocol 2 (cf. Section 3.8) and thereby eliminating *Type I* orphans slightly reduces the overall size of the VI-CBF, but does not impact the exponential growth.

The compression reduces the uncompressed size by a more-or-less constant factor of 0.52 ± 0.01. This factor was used to approximate values after step 90, as they could no longer be experimentally determined due to the RAM limitations of the evaluation machine.

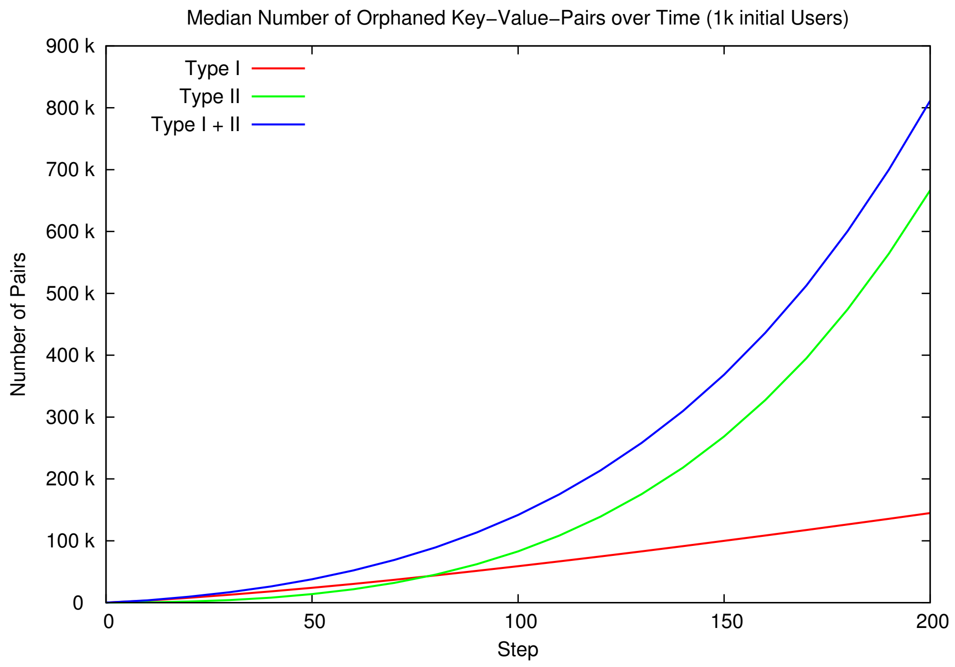


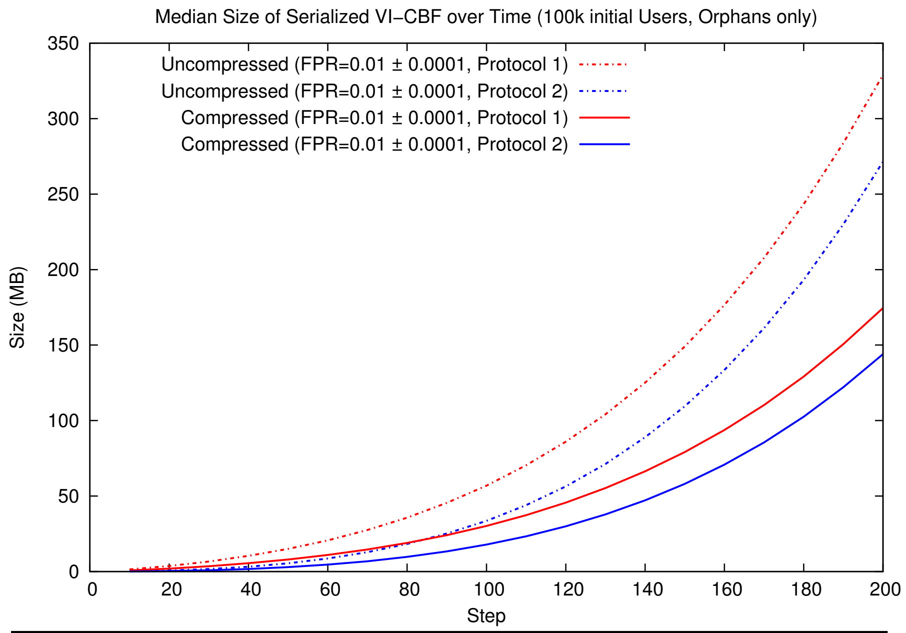
Figure 31: Median number of *Type I* and *Type II* orphans (1 000 initial users, 200 steps, 1000 samples, Protocol 1)

influence of target fpr Another interesting question is the impact of the target FPR on the size of the serialization. We repeated our parameter and size approximation for two additional FPRs: 0.001 (0.1%) and 0.1 (10%). The results are shown in Figure 33. To ensure readability, only the data for Protocol 1 is shown. Protocol 2 behaves as expected from the previous results, following the trend at a small offset caused by the elimination of *Type I* orphans.

The results show that decreasing the target FPR from 0.1 to 0.01 changes the compressed size by a factor of about 2.2 for the same number of entries, while decreasing it again to 0.001 changes it by a factor of another 1.45. The slot count shows a similar behaviour: The first decrease in the FPR changes the slot count by a factor of 1.96, while the second decrease shows a factor of 1.47. Repeating the simulation with 1 000 initial users shows almost exactly the same behaviour, with the small deviations explained in Section 5.5.1.

results (no orphans) To get an idea for a lower bound on the VI-CBF size, we return to the hypothetical orphan-less system discussed in Section 5.5.1. The results are shown in Figure 34. As expected from the previous evaluation, the growth is still exponential, but much slower, reaching only about 15 MB in round 200 for an FPR of 0.1%.

More importantly, the system now scales directly with the number of active users and their friends, and is no longer impacted by users leaving the system. This represents a lower bound for the system,



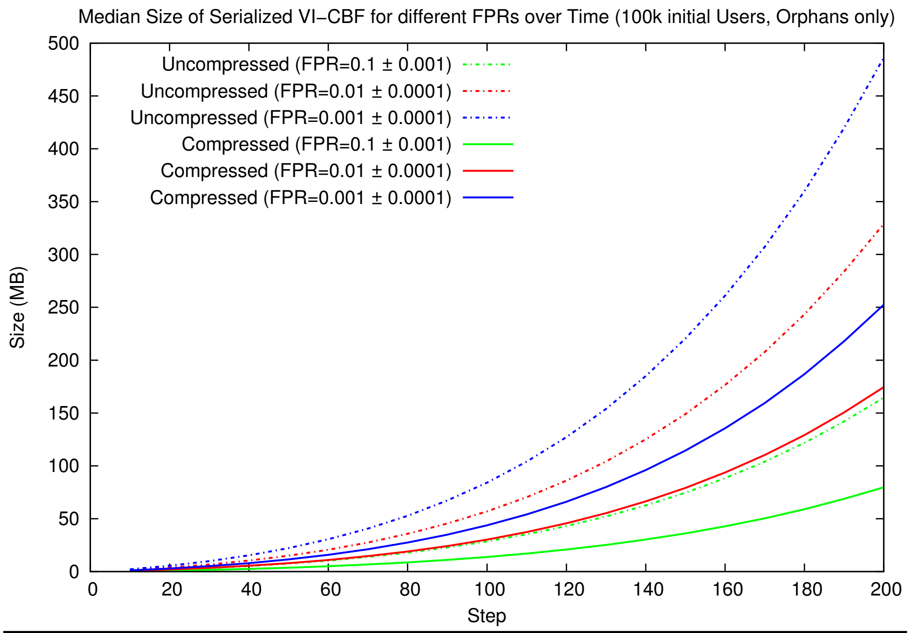
	Protocol 1			Protocol 2		
Step	m ($\times 10^6$)	u (MB)	c (MB)	m ($\times 10^6$)	u (MB)	c (MB)
10	1.52	1.45	0.77	0.16	0.15	0.08
50	15.71	14.98	7.95	5.76	5.49	2.92
100	59.69	56.92	30.21	35.24	33.61	17.83
150	156.30	149.06	79.10	114.75	109.44	58.07
200	344.87	328.90	174.54	284.63	271.44	144.03

Figure 32: Uncompressed (u) and compressed (c) size in MB and number of slots m of optimal VI-CBF containing only median number of orphans (100 000 initial users, FPR $= 0.01 \pm 0.0001$, number of hash functions $k = 5$, 1000 samples, values after step 90 approximated)

using Protocol 2 to eliminate *Type I* orphans and assuming that *Type II* orphans are prevented in another way. Achieving this performance in practice would, however, be difficult due to the difficulties in detecting and removing *Type II* orphans.

5.6 SUMMARY

Our evaluation has shown that the system is secure against all considered adversaries. We have discovered a number of open issues that would have to be solved to increase the security even further and enable advanced features like using multiple devices per user. However, none of these issues impact the security against the specified adversaries.



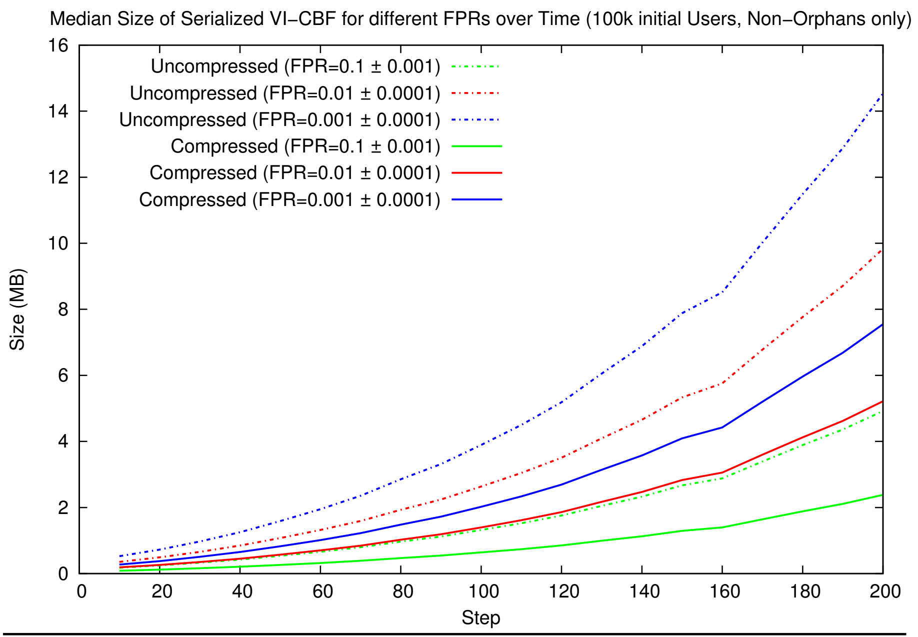
	FPR= 0.1 (k = 2)		FPR= 0.01 (k = 5)		FPR= 0.001 (k = 7)	
Step	m ($\times 10^6$)	c (MB)	m ($\times 10^6$)	c (MB)	m ($\times 10^6$)	c (MB)
10	0.76	0.35	1.52	0.77	2.25	1.12
50	7.86	3.63	15.71	7.95	23.23	11.50
100	29.87	13.79	59.69	30.21	88.26	43.70
150	78.23	36.11	156.30	79.10	231.11	114.42
200	172.60	79.69	344.87	174.54	509.95	252.48

Figure 33: Number of slots (m) and compressed (c) size in MB of optimal VI-CBF containing only median number of orphans for different FPRs (100 000 initial users, Protocol 1, 1000 samples)

The performance evaluation shows that while the computational cost of the protocol is unlikely to cause any problems, the networking overhead introduced by the transmission of the VI-CBF is significant. For a target FPR of 1%, the VI-CBF quickly grows to over 150 MB for a system with 100 000 initial users, and even an impractically high FPR of 10% still results in VI-CBFs of 80 MB and upwards. The sizes also show an exponential growth, which will make the transmission overhead more and more prohibitive as the system ages and grows.

Switching to the alternate protocol proposed in Section 3.8 will slow the growth, but does not change the exponential nature, as it is caused by *Type II* orphans, which are not prevented by the protocol.

Even though the simulation makes pessimistic assumptions (like users continuing to share to inactive friends indefinitely instead of stopping when the friend leaves the system), the results are discouraging. Given that many potential users are using mobile networks



	FPR= 0.1 ($k = 2$)		FPR= 0.01 ($k = 5$)		FPR= 0.001 ($k = 7$)	
Step	m ($\times 10^6$)	c (MB)	m ($\times 10^6$)	c (MB)	m ($\times 10^6$)	c (MB)
10	0.19	0.09	0.35	0.19	0.55	0.27
50	0.56	0.26	1.08	0.57	1.67	0.83
100	1.38	0.64	2.63	1.40	4.08	2.02
150	2.80	1.29	5.33	2.83	8.27	4.09
200	5.16	2.38	9.83	5.22	15.24	7.55

Figure 34: Number of slots (m) and compressed (c) size in MB of optimal VI-CBF containing only median number of non-orphans for different FPRs (100 000 initial users, 1000 samples)

with both low bandwidths and a low traffic quota, even a 50% lower overhead than the simulation indicates would make the system impossible to use for them. Ideally, the overhead should be below a megabyte per request, as the low speeds of mobile network connections would otherwise induce delays visible to the user. However, it is clear that VI-CBFs will never offer this level of performance.

In practice, unless a way is found to eliminate *Type II* orphans, only very small deployments with 3- or 4-digit user counts or more-or-less static networks (without users leaving the system) could achieve an acceptable overhead. Im these deployments, the number of possible senders and recipients of a message (i.e. the *anonymity set*, cf. Chaum [23]) would be naturally limited, making it easier for an adversary to identify users through other means, such as metadata or other *a priori* knowledge.

Part III

DISCUSSION AND CONCLUSIONS

After the evaluation, we discuss the effects of allowing additional types of adversaries and name a number of open problems that require further work to solve. Finally, we finish with a conclusion.

6 DISCUSSION

In the evaluation, we have shown the strengths and weaknesses of our proposed system. Now, we want to discuss the results further. We discuss how the system would hold up against more powerful adversaries. For this, we will also consider the effect of multiple colluding attackers. Finally, we will discuss a number of avenues for potential future work that may make the system more secure or more efficient.

6.1 ADVERSARY MODEL

While we have shown our design to be secure against the adversary model specified in Section 3.2, the security may not hold under different assumptions. In the following sections, we will discuss how allowing an active, malicious server or collusion between adversaries would affect the security of the system.

6.1.1 *Active Adversaries*

The most obvious variant of the adversary model would be to allow the server to become an active adversary that is no longer required to respect the protocol, but can send arbitrary messages and change any data it has access to. This change in the adversary model opens up a number of new attack vectors, which we will now discuss.

Note that all other adversaries are already active

DENIAL OF SERVICE An active server adversary can choose to stop serving (a portion of) clients at any time. This is an attack on the availability and fundamentally impossible to prevent, as clients cannot force the server to serve them data. As such, we will disregard any attacks on availability.

DATA MODIFICATION The adversary can attempt to change any data saved on it before delivering it to the clients. However, modifying a *key-* or *data-block* will be detected, as we are using an authenticated cipher that will check the integrity of the ciphertext on decryption. This means that any modification of the data will result (with overwhelming probability) in a decryption failure, transforming this into an attack on the availability.

The case of the server modifying the VI-CBF will be discussed separately

Another possibility would be for the server to modify the registered *studies* that it offers to the clients. Any tampering on these would be detected by the signature over the study request, unless the server also switches out the public key included in the study. Switching out the public key would, however, be detected by the public key authentication system that would check the website of the study for the fingerprint of the key.

The server could obviously also change the study URL, but at this point, it becomes equivalent to the server discarding the registered study and creating its own, copying the description. The detection of forged studies (studies that claim to be from one institution but aren't) can't be done automatically and is an issue of user education.

Finally, the server can modify the messages sent by clients attempting to *join* a study. As they are not authenticated and use a publically known public key, their contents can easily be replaced by the server. However, this is equivalent to the server discarding the old message (which would be an attack on availability) and creating a new one. The effect would be that one user thinks she is participating in a study but in fact is not. However, it would *not* allow the server to read the data submitted by the client.

REPLAY ATTACK An active adversary is capable of remembering old messages and replaying them to clients. In our case, this is mostly interesting for the results of GET requests. For our analysis, we distinguish two cases: The server is either trying to replay a *key block* or a *data block*.

If the server is trying to replay a key block, the client will detect the manipulation because the decryption will fail for one of two reasons. The data will either be...

1. ...encrypted with the *wrong key* if the key block was not destined for this client, or

2. ...encrypted with a *different Initialization Vector (IV)* if the key block was destined for this client, but originally uploaded under a different identifier.

The second case will fail because the IV depends on the value of the used $\mathtt{ctr}_{AB}$, which will be different for the replayed message because it was originally uploaded under a different identifier, generated with a different $\mathtt{ctr}_{AB}$.

If the server instead replays a *data block*, the decryption will fail as that data block will (with overwhelming probability) be using a different encryption key. Encryption keys are randomly generated for each individual data block, making it unlikely that any two share the same key.

Finally, the server can replay *study-join* messages to researchers. This could be detected by the research client by checking if the Elliptic Curve Diffie-Hellman (ECDH) public key contained in the message has been used before. Failing that, the effect would be that the researcher has a duplicate study participant in her database, which would not have any significant adverse effect.

SEND ERROR MESSAGES The server can also claim errors where none exist. There are a number of different variants of this we will discuss.

Upload Error: If the server claims an error during the upload process, the client will either re-encrypt the data and try the next identifier (if the error claimed the identifier was already taken) or abort the process and try again later (in case of other error messages).

In the first case, this would give the server information about two linked identifiers that could later be used to link two queries by the recipient. However, due to the general unlinkability of the identifiers, this would not help the adversary to link any other identifiers. It would also give the adversary two messages with the same plaintext, but encrypted under different IVs (in the case of a *key block*) or keys (for a *data block*). The usefulness of this is questionable, as we are aware of no attacks on AES in Galois/Counter Mode (GCM) that would benefit from this.

If the client retries later, it will encrypt data under the same key with the same value of $\mathtt{ctr}_{AB}$, but the IV will still be different with overwhelming probability, as it also contains a random component. This prevents any attacks on AES-GCM that are based on encrypting data under the same key and IV more than once.

Download Error: A download error can force the client to re-request the same identifier later. This could be used to link two sessions of a client, but at the price of denying service to it for at least one identifier. Once implemented, this would also trigger the „dirty marking“ of the Variable-Increment Counting Bloom Filter (VI-CBF), which is supposed to prevent sending too many queries based on a false positive in the VI-CBF (cf. Section 3.6.3). However, this could be countered by the server introducing small changes to the relevant slots of the bloom filter, which it can determine because it knows the identifier.

This attack is impossible to prevent. It could however potentially be detected by tracking the behaviour of the server and warning the user if it looks like the server is performing this attack (because in each batch of queries, at least one fails).

DECRYPTION ORACLE In cryptography research, a *decryption oracle* is a „black box“ available to the adversary, who can use it to decrypt (parts of) arbitrary messages without knowing the key. The server can attempt to use a legitimate user as a decryption oracle in the following way: If the user sends its first batch of GET requests in the session, the server can be reasonably sure that these queries are targeting *key blocks*, as those need to be retrieved in order to find the identifier of the *data blocks* containing the actual data.

The server will now send the client encrypted data. The client will attempt to decrypt the data using the key k_{AB} associated with the queried identifier id_{AB}. If the decryption is successful, an authentic data block will have the format $id_d \parallel k_d$. The client will then send another query, GET(id_d), to the server.

The server can use this by sending arbitrary encrypted data. If the client successfully decrypts it, the first $|id_d| = 256$ bit of the plaintext will be interpreted as an identifier and sent to the server as a GET-query. This will disclose that part of the plaintext to the server.

However, the server is extremely unlikely to succeed in this when sending anything except the correct key block. As the used algorithm, AES-GCM, also checks authenticity and integrity, the server would have to be very lucky to generate a ciphertext that actually decrypts without failure. Additionally, it cannot decrypt any other key blocks with this method, for the reasons outlined in the paragraphs about the *replay attack*.

In conclusion, while using clients as a decryption oracle is a theoretical attack, the practical ramifications are minimal.

FORGED VI-CBF The VI-CBF is used to determine which identifiers are present on the server without disclosing said identifiers to the server (cf. Section 3.6.3). However, the server controls the VI-CBF it distributes, and could forge a bloom filter that will return *true* for any query by setting all slots to a high value. This would lead to clients always assuming their identifiers to be available on the server, so they would be sending them every time they connect, thereby identifying themselves.

This can be countered by letting the clients calculate the current False-Positive-Rate (FPR) of the bloom filter they received, and rejecting it if the FPR is too high. This would turn the attack into an attack on the availability, which we explicitly do not defend against.

CONCLUSION This concludes our analysis of the effects of an active server adversary. While the power of an active adversary enables a number of additional attacks, most of them can be defeated by a well-designed client. However, an active server adversary can still be dangerous if it colludes with others. This is what we will discuss next.

6.1.2 *Colluding Adversaries*

So far, we have only discussed individual adversaries. However, some additional attacks may become possible if we allow adversaries to collude and share information. In this section, we will briefly discuss a selection of potential new attacks in this scenario.

LINKING SESSIONS In this scenario, the (honest-but-curious) server colludes with a malicious user or researcher to link a number of sessions. The colluding user, Chuck, will provide the server with the identifiers used by his friend Alice to share data with him. This allows the server to link all sessions in which Alice is sharing data with or receiving it from Chuck.

This attack could be improved if Chuck stores a new message for Alice every time Alice has retrieved the last message, thereby forcing Alice to indirectly identify herself by retrieving the data every time she checks for new shares from her friends. The server would inform Chuck and let him store a new message directly after Alice has disconnected. Chuck could use the revocation placeholder `0x42` (cf. Section 3.6.4/4.6.3) as the stored value, as this will not give any visible indication to Alice that any data has been retrieved. A similar attack would be possible with a colluding researcher.

This attack cannot really be defended against, as it isn't possible to keep Chuck from disclosing his secrets to the server. The system requires Alice to trust her friends not to work against her interests.

CIPHERTEXT REPLACEMENT An active, malicious server could also collude with Chuck to change the contents of messages in the following way: Alice shares a piece of data with both Chuck and Bob. This requires her to upload a *data block* containing the encrypted data c_d and two key blocks containing id_d and k_d, stored under id_{AB} and id_{AC}.

Chuck would retrieve his key block, stored under id_{AC}, and decrypt it to receive id_d and k_d. He can now retrieve and decrypt c_d to receive $m = Dec(k_d, c_d)$. However, he now knows the key used to encrypt the data and can forge a new message by encrypting any data under k_d to receive $c'_d = Enc(k_d, m')$.

This attack is prevented by Protocol 2, as it shares the data directly with each recipient

Alone, he cannot replace the c_d stored on the server, as he does not know the revocation token x so that $h(x) = id_d$. However, with a colluding, malicious server, he can provide the new ciphertext to the server and ask it to replace c_d with c'_d. If Bob now retrieves his key block and the new data block, it will decrypt to m'. Bob cannot detect the forgery, as the data is only authenticated by k_d, which is under the control of the adversary.

This attack can be prevented with a small change in the protocol: Instead of setting the key block to $c_{AB} = Enc(k_{AB}, id_d \parallel k_d)$, Alice will set it to $c_{AB} = Enc(k_{AB}, id_d \parallel id_d \parallel h(c_d))$. Even with the assistance of Chuck, the server cannot change c_{AB}, and the hash of the message will let Bob detect if the message has been modified. The additional overhead is negligible (one hash operation, 256 bit additional encrypted data), and it prevents this message forgery attack.

We omit the IV specification for readability

6.2 FUTURE WORK

Over the course of this thesis, we have made a number of design decisions that drastically influenced the system. These decisions include the use of bloom filters, the decision against using Private Information Retrieval (PIR), and (most importantly) the use of a client-server architecture. Due to time constraints, we leave the evaluation of these alternatives as future work and only briefly discuss them here.

6.2.1 *Cuckoo Filters*

Early on in the design process, we decided to use VI-CBFs in the data retrieval process. These have turned out to be a major performance bottleneck of the system (cf. Section 5.5). Thus, finding a more efficient alternative could allow the system to scale better.

A possible replacement for VI-CBFs would be so-called *cuckoo filters*. Originally proposed by Fan *et al.* [48], they are similar to bloom filters, but use a technique called *cuckoo hashing* (cf. Pagh *et al.* [89]) to allocate the slots in the data structure.

Cuckoo filters work by determining two candidate slots (called *buckets* in the proposal) for an item using two different hash functions. Buckets can contain a fixed number b of items.

When inserting an item x, the two buckets are determined and a fingerprint of x is inserted into one of them which still has free space. If both buckets are full, the fingerprint is inserted into one of the two buckets at random, evicting an old fingerprint y and inserting it into its alternate location. If the alternate location is full, one of the existing items is evicted. This process continues until either all items have been successfully inserted or a maximum number of evictions has taken place without finding a bucket with enough free space, in which case the insertion fails.

When querying for an item x, the two potential buckets are determined. If x has been inserted into the cuckoo filter, its fingerprint is guaranteed to be in one of its two possible buckets. This means that a query has to check at most two buckets with b items each before either finding the fingerprint or being certain that it is not in the cuckoo filter.

Cuckoo filters have been shown to be more space-efficient than regular bloom filters. However, there has been no direct comparison between cuckoo filters and VI-CBFs, and it is not clear if it would improve the overhead. Determining their relative size depending on different target false-positive-rates and numbers of inserted items could give insight into possible ways to improve the overhead of the system.

6.2.2 *Private Information Retrieval*

While designing the system, we decided against using PIR to retrieve data from the server, citing the high computational and transmission overhead. However, now that we have experimental results of the overhead introduced by using a VI-CBF instead of PIR, a real comparison becomes possible.

Future work could attempt using a PIR scheme like XPIRe [82] or an ϵ-private PIR system (cf. Toledo *et al.* [111]) in place of the current retrieval system and compare the real-world performance.

ϵ-private PIR trades weaker privacy for better performance

6.2.3 *Alternative Security Goals*

The scaling problems of the system are due to our method to ensure unlinkability. If we decide that a pseudonymous system without unlinkability is acceptable, the bloom filter could be removed from the design, thereby removing all scaling issues. However, in this case, pseudonymous account-based system designs would likely achieve a better user experience for the same security.

6.2.4 *Metadata Obfuscation*

Hiding some of the remaining metadata, like the number of people we are sharing data with, could further improve the privacy offered by our system. However, the intuitive solution of uploading dummies to obfuscate the number of actual data items we want to upload would put additional strain on the system. Additionally, dummy strategies should be carefully evaluated (e.g. using the methodology by Oya *et al.* [88]), otherwise they may not provide the intended protection.

An example where dummies fail to provide privacy is shown in [59]

Future work could attempt to design and evaluate metadata obfuscation strategies that improve the privacy of the system without unduly impacting its performance.

6.2.5 *Study Data De-Identification*

Past experience (e.g. for credit card transactions [35], the *Netflix* dataset [85], or the *Yahoo* search dataset[1]) has shown that pseudonymization of data is not sufficient to ensure that the users cannot be re-identified. Our system currently takes no extra measures to protect the identities of users participating in a study, which would make re-identification attacks possible.

1 See `http://query.nytimes.com/gst/abstract.html?res=9E0CE3DD1F3FF93AA3575BC0A9609C8B63`, last visited March 24th, 2016

A body of past work on (health) data anonymization exists [44, 52, 54, 76, 78, 86, 93], but most deal with anonymization by a researcher who wants to share her existing datasets with others. Future work could investigate if and how additional anonymization and minimization techniques can be used to further protect the identity of users, even against the initial researcher, without rendering the collected data useless.

6.2.6 *Distributed/Peer-to-Peer Infrastructure*

Finally, we decided to use a centralized server instead of a distributed infrastructure, as the implementation complexity of building an efficient Peer-to-Peer (P2P) system on mobile devices seemed prohibitive. However, using a P2P data structure like a Distributed Hashtable (DHT) to store our information could improve the privacy and performance of our system. Alternatively, an anonymous messaging system like *Riposte* [27] could be used to distribute the data.

Future work could implement a distributed variant of our system and report the performance and privacy characteristics.

6.3 SUMMARY

This concludes the discussion of our system. We have shown areas where more work would be necessary, both to ensure the security and privacy of the system and to potentially improve the efficiency. However, significant progress in the area of PIR or Private Set Intersection (PSI) would be necessary to reduce the overhead to a practical level. Until such progress is made, a distributed infrastructure may offer a more efficient solution.

7 CONCLUSIONS

In this thesis, we designed and evaluated a system for privacy-preserving health and fitness data sharing. Our intent was to design a system that does not require any Trusted Third Parties (TTPs) while ensuring the maximum achievable privacy, at the cost of reduced efficiency compared to less-private solutions.

Our design uses a centralized *client-server* system, with the server acting as a *key-value*-store. Privacy is achieved by foregoing a user account system in favor of a hash-based identifier system. The identifiers are created based on shared secrets and unlinkable to outsiders while being predictable to the intended recipient. The privacy is further improved by using a bloom filter to check for the presence of an expected identifier on the server before requesting it. Data confidentiality, integrity and authenticity is ensured through symmetric encryption in an authenticated encryption mode like Galois/Counter Mode (GCM).

A proof of concept with limited functionality was implemented for the *Android* mobile operating system. We also implemented a basic simulator to evaluate the performance of the system in a large-scale deployment with realistic user behaviour.

In a formal and theoretical analysis, we showed that our system fulfills the required security and privacy properties. The computational performance was evaluated using the proof of concept implementation and found to be satisfactory. The major bottleneck of the system is the networking performance, as the overhead of transmitting the bloom filter to clients grows prohibitive in larger deployments ($> 100\,000$ users), quickly reaching tens or even hundreds of megabytes. Much of this overhead is introduced through user churn, which is unavoidable in a real-world deployment.

Finally, we discussed a number of open issues of the system, including post-compromise security and potential methods to reduce the overhead of the system. Areas for future work include comparing its performance to existing methods for Private Information Retrieval (PIR), adding Post-Compromise Security (PCS) [26], and building an alternative, distributed (*peer-to-peer*) version of the system to evaluating its privacy and security.

The results illustrate the difficulty of efficiently providing strong privacy guarantees, especially unlinkability, in a system with a centralized server. This is a fundamental limitation of the client-server model, and would require further advances in the area of PIR and related areas like Private Set Intersection (PSI) to overcome. Until these advances are made, distributed systems may provide more efficient solutions to these problems.

In conclusion, our system provides strong privacy guarantees in a classical adversary model. However, the networking overhead quickly grows prohibitive, making it unsuitable for a large-scale deployment. Further work is required to design a practical system for data sharing in the health context with strong privacy guarantees.

Part IV

APPENDIX

A APPENDIX

A.1 VI-CBF SERIALIZATION ALGORITHM

As the Variable-Increment Counting Bloom Filter (VI-CBF) is transmitted frequently, finding a space-efficient serialization format is key in reducing the overhead it introduces. Some values always have to be transmitted: The number of hash functions k, the number of counters m, the base L of the D_L-sequence, the number of bits per counter b, and the current number of entries in the VI-CBF (which is used to calculate the current False-Positive-Rate (FPR)). All of these values are serialized by converting them to bytes and used as a header for the serialized data.

While the VI-CBF is in memory, only non-zero values of the VI-CBFs counters are stored, and once a counter becomes zero, it is deleted. This is done to reduce the memory footprint of the VI-CBF. However, this strategy is not efficient for serialization: If we only write out the counters with a non-zero value, we have to indicate which slot the counter value belongs to. This adds additional bits to the serialization.

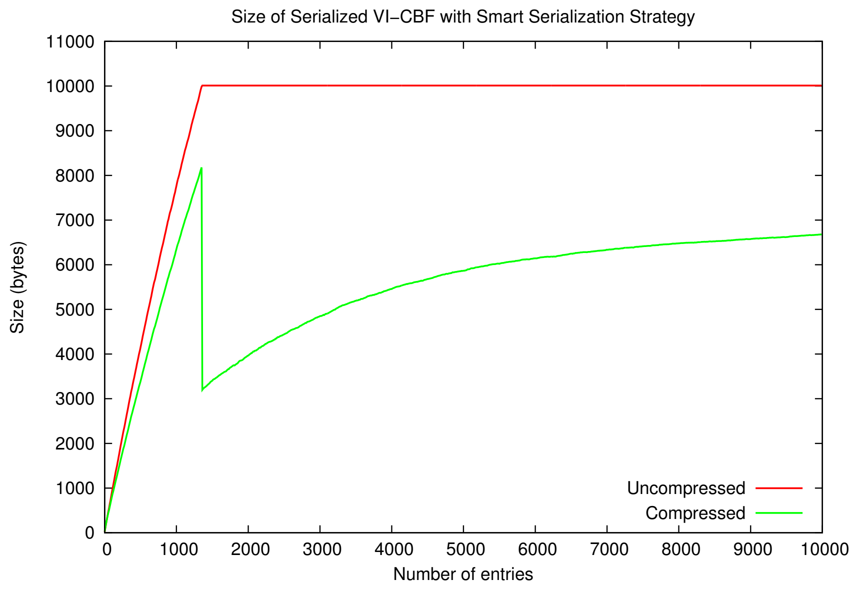
We call this the *smart* serialization, and the alternative, simply writing out all slot values (including zeroes), the *full* serialization strategy. We can estimate the size of the *smart* and *full* serialization and dynamically choose the more efficient strategy during serialization.

However, in the implementation, the serialized output will be compressed using DEFLATE [36] before being transmitted. DEFLATE compresses best if the data has a lot of redundancy, like the data generated by the *full* strategy. Figure 35 shows the results of both strategies, evaluated in an actual implementation. The size reduction through compression is so significant that even with only a few non-zero counters, the *full* serialization strategy is still more efficient.

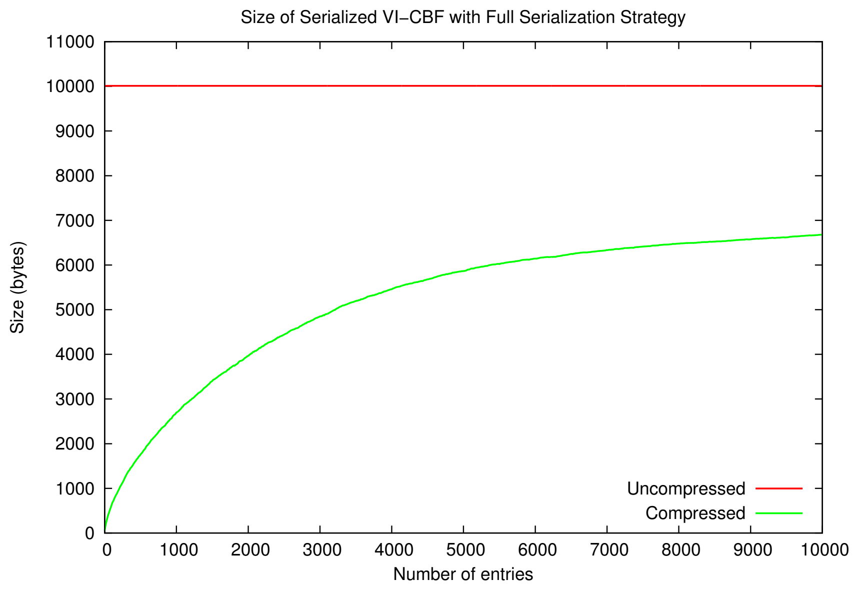
For this reason, we chose to disregard the *smart* strategy and only use the *full* strategy, combined with a DEFLATE compression, in the implementation.

A.2 SOURCE CODE AND RAW DATA

The enclosed DVD contains all code and data used for the thesis. In case you are reading the PDF of this thesis, we will also provide a link to an online version of the files. We will now step through the different software projects and data files. Each folder also contains a



(a) *Smart* serialization size



(b) *Full* serialization size

Figure 35: Comparison of serialization size with *smart* and *full* strategy, with and without compression (10 000 slots, 3 hash functions)

README file with further information. The software was developed under the project name *Denul*.

ANDROID APP The proof of concept Android application can be found in the `Android` folder on the enclosed DVD, or on GitHub.[1] It is written in Java and follows the standard folder structure of an

1 See `https://github.com/DenulApp/app`

Android development project using Android Studio. A compiled version is placed in the root of the `Android` folder, named `Denul.apk`.

SERVER The server application is required to let two instances of the proof of concept application communicate. It can be found in the `Server` directory of the DVD, or on GitHub.[2] Further documentation can be found in its README file.

Note that the address of the used server is fixed to the DNS entry of my server, denul.velcommuta.de. If you want to use your own server, you will have to change the `host` variable in the following two files:
`de.velcommuta.denul.util.ShareManager.java`
`de.velcommuta.denul.util.StudyManager.java`.

RESEARCH CLIENT The research client is written in Java and can be found in the `Research` folder, or on GitHub.[3]

PROTOCOL BUFFERS Both the server and the clients require compiled Protocol Buffer files for their communication. The protocol buffer definition files can be found in the `Protobuf` folder, or on GitHub.[4] The server, Android application, and research client already contain the compiled protocol buffer files, no further actions are necessary. The source files are included for completeness only.

SIMULATOR The simulator is written in Python, and the source code can be found in the `Simulator` folder or on GitHub.[5] Further information can be found in the README file.

VI-CBF The project required two separate VI-CBF implementations. The python version is called pyVICBF,[6] while the Java version is called libvicbf.[7] Their respective source code is included in the folders of the same name on the DVD.

EXPERIMENTAL DATA Finally, the experimental data is included in the `Data` folder. The online version can be partially found on GitHub,[8] while the larger files have been uploaded to Zenodo.[9]

The folder contains the scripts used to perform the evaluation in the `code` subfolder. It also contains the script used to generate the plots used in this thesis. Simply run the `plot.py` python script - if you have GnuPlot installed, it should generate the appropriate .eps

2 See `https://github.com/DenulApp/server`
3 See `https://github.com/DenulApp/research-client`
4 See `https://github.com/DenulApp/protocol`
5 See `https://github.com/DenulApp/simulator`
6 See `https://github.com/malexmave/pyVICBF`
7 See `https://github.com/malexmave/libvicbf`
8 See `https://github.com/DenulApp/data`
9 See `https://zenodo.org/record/49441`

Component	**`github.com/`**	**License**
Android App	DenulApp/app	GNU GPL v3
Server	DenulApp/server	GNU GPL v3
Research Client	DenulApp/research-client	GNU GPL v3
Protocol Buffers	DenulApp/protocol	GNU GPL v3
Simulator	DenulApp/simulator	GNU GPL v3
Dataset	DenulApp/data	Public Domain
Evaluation Scripts	DenulApp/data	Public Domain
VI-CBF - Python	malexmave/pyVICBF	Apache License v2
VI-CBF - Java	malexmave/libvicbf	Apache License v2

Table 4: Software and dataset licenses

files in the `output` subfolder. Naturally, this does not work on the DVD itself, as it is read-only.

LICENSE The different components are licensed under different licenses. Check Table 4 for details.

VERSION CONTROL The version history of the code is included in the online repositories, using the Git version control system. In case development of the proof of concept resumes, the versions that were used for the evaluation have been marked with a Git *tag*. This ensures that results are reproducible, even if the software is changed in the meantime.

The tags (and most commits) are signed with my GPG key to ensure authenticity. The key fingerprint is `84C4 8097 A3AF 7D55 189A 77AC 169F 9624 3408 825E`.

BIBLIOGRAPHY

[1] M. Au and A. Kapadia. PERM: practical reputation-based blacklisting without TTPS. *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 929–940, 2012.

[2] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 135–146, 2009.

[3] A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, oct 1999.

[4] E. Barker and J. Kelsey. Recommendation for random number generation using deterministic random bit generators (revised). *NIST Special publication*, 800(March):90, 2007.

[5] M. A. Barrett, O. Humblet, R. A. Hiatt, and N. E. Adler. Big Data and Disease Prevention: From Quantified Self to Quantified Communities. *Big Data*, 1(3):168–175, Mary Ann Liebert, sep 2013.

[6] J. Becker, D. Breuker, T. Heide, J. Holler, H. P. Rauer, and R. Böhme. Can we afford integrity by proof-of-work? scenarios inspired by the bitcoin currency. In *The Economics of Information Security and Privacy*, pages 135–156. Springer, 2013.

[7] M. Bellare and T. Kohno. Hash function balance and its impact on birthday attacks. *Lecture Notes in Computer Science*, 3027:401–418, Springer, 2004.

[8] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. *Advances in Cryptology - CRYPTO'96*, 1109:1–15, Springer, 1996.

[9] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation. *Fast Software Encryption (FSE)*, pages 389–407, Springer, 2004.

[10] M. Bellare. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology - CRYPTO 2006*, pages 602–619. Springer, 2006.

[11] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. *Proceedings of the 1st ACM conference on Computer and communications security*, (November 1993):62–73, 1993.

[12] M. Bellare, A. Boldyreva, and A. O. Neill. Deterministic and Efficiently Searchable Encryption. *Advances in Cryptology - CRYPTO 2007*, 4622:535–552, Springer, 2007.

[13] D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.

[14] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On the indifferentiability of the sponge construction. In *Advances in Cryptology - EUROCRYPT 2008*, pages 181–197. Springer, 2008.

[15] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3:30, Citeseer, 2009.

[16] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 321–334. IEEE, 2007.

[17] M. Blaze. A cryptographic file system for UNIX. *Proceedings of the 1st ACM conference on Computer and communications security*, pages 9–16, 1993.

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[19] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *Advances in Cryptology - CRYPTO 2001*, pages 213–229. Springer, 2001.

[20] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public Key Encryption with Keyword Search. *Advances in Cryptology - Eurocrypt 2004*, pages 506–522, Springer, 2004.

[21] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, pages 77–84, 2004.

[22] G. Brassard, C. Crépeau, and J. Robert. All-or-nothing disclosure of secrets. *Advances in Cryptology—CRYPTO' 86*, pages 234–238, Springer, 1987.

[23] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1): 65–75, Springer, 1988.

[24] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[25] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–982, 1998.

[26] K. Cohn-Gordon, C. Cremers, and L. Garratt. On Post-Compromise Security. *Cryptology ePrint Archive*, Report 2016/221, 2016.

[27] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An Anonymous Messaging System Handling Millions of Users. *IEEE Symposium on Security and Privacy (SP)*, pages 321–338, 2015.

[28] L.-A. Cutillo, R. Molva, and T. Strufe. Safebook: A Privacy Preserving Online Social Network Leveraging on Real-Life Trust. *IEEE Communications Magazine*, 47(12):94–101, 2009.

[29] L. A. Cutillo, R. Molva, and T. Strufe. On the Security and Feasibility of Safebook: A Distributed Privacy-Preserving Online Social Network. *Privacy and Identity Management for Life*, (217141):86–101, Springer, 2010.

[30] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002. ISBN 3540425802.

[31] I. B. Damgård. A design principle for hash functions. In *Advances in Cryptology—CRYPTO'89 Proceedings*, pages 416–427. Springer, 1989.

[32] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 2–15. IEEE, 2003.

[33] G. D'Angelo and S. Ferretti. LUNES: Agent-based simulation of P2P systems. In *Proceedings of the International Workshop on Modeling and Simulation of Peer-to-Peer Architectures and Systems (MOSPAS 2011)*. IEEE, 2011.

[34] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security*, pages 143–159. Springer, 2010.

[35] Y.-A. de Montjoye, L. Radaelli, V. K. Singh, A. S. Pentland, Y.-a. D. Montjoye, L. Radaelli, and V. K. Singh. Unique in the shopping mall: On the reidentifiability of credit card metadata. *Science*, 347(6221):536–539, jan 2015.

[36] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

[37] C. Devet and I. Goldberg. The best of both worlds: Combining information-theoretic and computational pir for communication efficiency. In *Privacy Enhancing Technologies*, pages 63–82. Springer, 2014.

[38] C. Diaz, C. Troncoso, and A. Serjantov. On the impact of social network profiling on anonymity. In *Privacy Enhancing Technologies*, pages 44–62. Springer, 2008.

[39] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[40] W. Diffie and M. E. Hellman. Privacy and Authentication: An Introduction to Cryptography. *Proceedings of the IEEE*, 67(3): 397–427, 1979.

[41] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. 2004.

[42] D. Dolev and A. C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, pages 198–208, 1981.

[43] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, number 1, pages 789–800, 2013.

[44] B. S. Elger, J. Iavindrasana, L. Lo Iacono, H. Müller, N. Roduit, P. Summers, and J. Wright. Strategies for health data exchange for secondary, cross-institutional clinical research. *Computer Methods and Programs in Biomedicine*, 99(3):230–251, Elsevier Ireland Ltd, 2010.

[45] T. Emnid. Datenschutz - Die Sicht der Verbraucherinnen und Verbraucher in Deutschland. 2015. URL `http://www.vzbv.de/sites/default/files/downloads/Datenschutz_Umfrage-Sicht-Verbraucher-Ergebnisbericht-TNS-Emnid-Oktober-2015.pdf`.

[46] L. Ertaul, A. M. Mehta, T. K. Wu, C. Dong, L. Chen, and Z. Wen. Implementation of Oblivious Bloom Intersection in Private Set Intersection Protocol (PSI). In *Proceedings of the International Conference on Security and Management (SAM)*, page 1, 2014.

[47] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[48] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of*

the 10th ACM International on Conference on emerging Networking Experiments and Technologies, pages 75–88. ACM, 2014.

[49] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[50] A. Fiat and M. Naor. Broadcast Encryption. *Advances in Cryptology — CRYPTO' 93, LNCS*, 773:480–491, Springer, 1993.

[51] M. Franz, B. Meyer, and A. Pashalidis. Attacking Unlinkability: The Importance of Context. *Privacy Enhancing Technologies*, 4776: 1–16, Springer, 2007.

[52] D. Galindo and E. R. Verheul. Pseudonymized Data Sharing. In *Advanced Information and Knowledge Processing*, volume 51, pages 157–179. 2010.

[53] P. Gallagher. Secure Hash Standard (SHS) FIPS PUB 180-4. *Processing*, FIPS PUB 1(October), 2012.

[54] A. Gkoulalas-Divanis, G. Loukides, and J. Sun. Publishing data from electronic health records while preserving privacy: A survey of algorithms. *Journal of Biomedical Informatics*, 50:4–19, Elsevier Inc., 2014.

[55] V. Goyal, A. O'Neill, and V. Rao. Correlated-Input Secure Hash Functions. *Theory of Cryptography*, 6597:182–200, Springer, 2011.

[56] S. Guha, K. Tang, and P. Francis. NOYB: Privacy in Online Social Networks. *Proceedings of the first workshop on online social networks*, pages 49–54, ACM, 2008.

[57] D. Guo, Y. Liu, X. Li, and P. Yang. False negative problem of counting bloom filter. *IEEE Transactions on Knowledge and Data Engineering*, 22(5):651–664, 2010.

[58] A. Harvey, A. Brand, S. T. Holgate, L. V. Kristiansen, H. Lehrach, A. Palotie, and B. Prainsack. The future of technologies for personalised medicine. *New Biotechnology*, 29(6): 625–633, Elsevier B.V., 2012.

[59] D. Herrmann, M. Maass, and H. Federrath. Evaluating the Security of a DNS Query Obfuscation Scheme for Private Web Surfing. In *ICT Systems Security and Privacy Protection*, volume 428 of *IFIP Advances in Information and Communication Technology*, pages 205–219. Springer, 2014.

[60] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? *19th Network and Distributed Security Symposium (NDSS)*, (February):5–8, The Internet Society, 2012.

[61] B. A. Huberman, M. Franklin, and T. Hogg. Enhancing Privacy and Trust in Electronic Communities. *Proceedings of the 1st ACM conference on Electronic commerce*, pages 78–86, 1999.

[62] M. Humbert, E. Ayday, J.-P. Hubaux, and A. Telenti. Addressing the concerns of the lacks family: quantification of kin genomic privacy. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*, pages 1141–1152, 2013.

[63] R. Impagliazzo and S. Rudich. Limits on the Provable Consequences of One-way Permutations. *Proceedings of 21st Annual ACM Symposium on Theory of Computing*, pages 44–61, 1989.

[64] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Sufficient Conditions for Collision-Resistant Hashing. *Theory of Cryptography, LNCS-3378*, pages 445–456, 2005.

[65] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), February 2003.

[66] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000.

[67] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869 (Informational), May 2010.

[68] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

[69] T. Krovetz and P. Rogaway. The OCB Authenticated-Encryption Algorithm. RFC 7253 (Informational), May 2014.

[70] C. Krumme, A. Llorente, M. Cebrian, A. Pentland, and E. Moro. The predictability of consumer visitation patterns. *Scientific Reports*, 3:4, Nature, apr 2013.

[71] E. Kushilevitz, R. Ostrovsky, and T. Bellcore. Replication is not Needed: Single Database, Computationally-Private Information Retrieval. *IEEE Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[72] B. Laurie and R. Clayton. Proof-of-work proves not to work. *The Third Annual Workshop on Economics and Information Security*, (May):1–9, 2004.

[73] L. Li, D. Alderson, J. C. Doyle, and W. Willinger. Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications. *Internet Mathematics*, 2(4):431–523, Taylor & Francis, jan 2005.

[74] N. Li and G. Chen. Analysis of a Location-Based Social Network. *Computational Science and Engineering, 2009. CSE'09. International Conference on*, 4:263–270, IEEE, 2009.

[75] Y. Lindell. Anonymous Authentication. *Journal of Privacy and Confidentiality*, 2(2):35–63, Carnegie Mellon University, 2007.

[76] G. Loukides, J. Liagouris, A. Gkoulalas-Divanis, and M. Terrovitis. Disassociation for electronic health record privacy. *Journal of Biomedical Informatics*, 50:46–61, Elsevier Inc., 2014.

[77] X. Ma, G. Chen, and J. Xiao. Analysis of an online health social network. *Proceedings of the 1st ACM international health informatics symposium*, pages 297–306, 2010.

[78] B. a. Malin and L. Sweeney. A secure protocol to distribute unlinkable health data. *AMIA Symposium*, pages 485–489, American Medical Informatics Association, 2005.

[79] M. Marlinspike. Advanced Cryptographic Ratcheting, 2013. URL `https://whispersystems.org/blog/advanced-ratcheting/`.

[80] M. Marlinspike. Forward Secrecy for Asynchronous Messages, 2013. URL `https://whispersystems.org/blog/asynchronous-security/`.

[81] D. McGrew and J. Viega. The Galois/Counter mode of operation (GCM). *Submission to NIST*, 2004.

[82] C. A. Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIRe: Private Information Retrieval for Everyone. Technical report, Cryptology ePrint Archive, Report 2014/1025, 2014.

[83] R. C. Merkle. One Way Hash Functions and DES. *Advances in Cryptology - CRYPTO'89*, pages 428–446, Springer, 1989.

[84] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement - IMC '07*, pages 29–42, 2007.

[85] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. *Proceedings - IEEE Symposium on Security and Privacy*, pages 111–125, 2008.

[86] T. Neubauer and J. Heurix. A methodology for the pseudonymization of medical data. *International Journal of Medical Informatics*, 80(3):190–204, Elsevier Ireland Ltd, 2011.

[87] R. Ostrovsky and W. E. Skeith III. A Survey of Single-Database PIR: Techniques and Applications. *Public Key Cryptography - PKC 2007*, pages 393–411, Springer, 2007.

[88] S. Oya, C. Troncoso, and F. Pérez-González. Do dummies pay off? Limits of dummy traffic protection in anonymous communications. *Privacy Enhancing Technologies*, pages 204–223, Springer, 2014.

[89] R. Pagh and F. F. Rodler. Cuckoo Hashing. *ESA 2001: 9th Annual European Symposium, 2001 Proceedings*, pages 121–133, Springer Berlin Heidelberg, 2001.

[90] A. Pfitzmann and M. Hansen. A terminology for talking about privacy by data minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management. *Technical University Dresden*, pages 1–98, 2009.

[91] A. Pham, I. Bilogrevic, I. Dacosta, and J.-p. H. Securerun. SecureRun: Cheat-Proof and Private Summaries for Location-Based Activities. *IEEE Transactions on Mobile Computing*, PP(99): 1–14, 2016.

[92] B. Pinkas, T. Schneider, and M. Zohner. Faster Private Set Intersection Based on OT Extension. *23rd USENIX Security Symposium (USENIX Security 14)*, pages 797–812, 2014.

[93] G. Poulis, G. Loukides, A. Gkoulalas-Divanis, and S. Skiadopoulos. Anonymizing data with relational and transaction attributes. In *Machine learning and knowledge discovery in databases*, pages 353–369. Springer, 2013.

[94] D. d. S. Price. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American society for Information science*, 27(5):292–306, Wiley Online Library, 1976.

[95] K. S. Raynes-Goldie. *Privacy in the age of facebook: discourse, architecture, consequences*. Phd thesis, Curtin University, 2012.

[96] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, feb 1978.

[97] P. Rogaway and T. Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. *FSE 2004: Fast Software Encryption*, pages 371–388, Springer, 2004.

[98] O. Rottenstreich, Y. Kanizo, and I. Keslassy. The Variable-Increment Counting Bloom Filter. In *IEEE INFOCOM*, pages 1880–1888. IEEE, mar 2012.

[99] S. Sankararaman, G. Obozinski, M. I. Jordan, and E. Halperin. Genomic privacy and limits of individual detection in a pool. *Nature Genetics*, 41(9):965–967, sep 2009.

[100] P. Schartner. Random but System-Wide Unique Unlinkable Parameters. *Journal of Information Security*, 3(01):1–10, Scientific Research Publishing, 2012.

[101] E. J. Schwartz, D. Brumley, and J. M. Mccune. Contractual anonymity. In *In Proceedings of the 17th Annual Network and Distributed System Security Symposium*. The Internet Society, 2010.

[102] R. Shirey. Internet Security Glossary, Version 2. RFC 4949 (Informational), August 2007.

[103] H. A. Simon. On a Class of Skew Distribution Functions. *Biometrika*, 42(3/4):425–440, JSTOR, dec 1955.

[104] R. Sion and B. Carbunar. On the computational practicality of private information retrieval. *Proceedings of the Network and Distributed Systems Security Symposium*, The Internet Society, 2007.

[105] J. Sun, X. Zhu, and Y. Fang. A Privacy-Preserving Scheme for Online Social Networks with Efficient Revocation. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, mar 2010.

[106] M. Swan. Emerging patient-driven health care models: An examination of health social networks, consumer personalized medicine and quantified self-tracking. *International Journal of Environmental Research and Public Health*, 6(2):492–525, Molecular Diversity Preservation International, 2009.

[107] M. Swan. Crowdsourced Health Research Studies: An Important Emerging Complement to Clinical Trials in the Public Health Research Ecosystem. *Journal of Medical Internet Research*, 14(2):46, JMIR Publications Inc., 2012.

[108] M. Swan. Health 2050: The Realization of Personalized Medicine through Crowdsourcing, the Quantified Self, and the Participatory Biocitizen. *Journal of personalized medicine*, 2(3):93–118, Molecular Diversity Preservation International, 2012.

[109] M. Swan. Scaling crowdsourced health studies : the emergence of a new form of contract research organization. *Personalized Medicine*, 9(2):223–234, Future Medicine, 2012.

[110] C. Tezcan and S. Vaudenay. On hiding a plaintext length by preencryption. *Applied Cryptography and Network Security*, 6715 LNCS:345–358, Springer, 2011.

[111] R. R. Toledo, G. Danezis, and I. Goldberg. Lower-Cost epsilon-Private Information Retrieval. *ArXiv ePrint, Report 1604.00223*, pages 1–17, apr 2016.

[112] P. P. Tsang, M. H. Au, A. Kapadia, and S. W. Smith. PEREA: Towards practical TTP-free revocation in anonymous authentication. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 333–344, 2008.

[113] G. Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5): 29–38, 1992.

[114] X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. *Advances in Cryptology - CRYPTO 2005*, pages 17–36, Springer, 2005.

[115] D. Whiting, R. Housley, and N. Ferguson. Counter with CBC-MAC (CCM). RFC 3610 (Informational), September 2003.

[116] J. R. Whitson. Gaming the quantified self. *Surveillance and Society*, 11(1/2):163–176, Surveillance Studies Network, 2013.

[117] U. Wilensky. NetLogo, 1999. URL `http://ccl.northwestern.edu/netlogo/`.

[118] P. Zimmermann, A. Johnston, and J. Callas. ZRTP: Media Path Key Agreement for Unicast Secure RTP. RFC 6189 (Informational), April 2011.

[119] P. R. Zimmermann. *The official PGP user's guide*. MIT press, 1995.

ERKLÄRUNG

Hiermit versichere ich gemäß der Allgemeinen Prüfungsbestimmungen der Technischen Universität Darmstadt (APB) § 23 (7), die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 15. April 2016

Max Jakob Maaß