Communication & Distributed Systems

RWTH AACHEN UNIVERSITY

Crowd-Sourced Privacy-Preserving Creation of Elevation Profiles using Barometers

Master Thesis

Nicolas Inden

RWTH Aachen University, Germany

Chair of Communication and Distributed Systems

Advisors:

Dipl.-Inform.	Jó Ágila Bitsch Link
Dipl.-Inform.	Henrik Ziegeldorf
Prof. Dr.-Ing.	Klaus Wehrle
Prof. Dr.	Bernhard Rumpe

Registration date: 2014-05-09

Submission date: 2014-09-26

I hereby affirm that I composed this work independently and used no other than the specified sources and tools and that I marked all quotes as such.

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 26. September 2014

Abstract

This thesis proposes a system for crowd-sourced privacy-preserving data collection with the focus on location dependent air pressure values. Currently, there is a lack of methods to anonymously contribute to location based systems. So far, such contributions rise the danger of being identified by homing or record linkage attacks. We implement and evaluate a system that enables users to anonymously contribute air pressure data for the creation of elevation profiles. In order to establish anonymity, our system ensures the *unlinkability* and *untraceability* of a contributor i.e., we can neither link from which source contributed data came from, nor can we trace the contributed traces back to a user by their content. To realize our system we implement an Android application to gather air pressure data and to act as an input peer to our system. Further, we use secure multi party computation in order to establish the mentioned anonymization on contributed data before it is finally collected by a potentially untrusted collector peer. We perform a thorough evaluation for feasibility and accuracy of air pressure based altitude differences and our anonymization system. Ultimately, we yield the result that the barometric altitude differences have an average error of 0.99m over all traces we collected during a one month test run. Compared to GPS based altitude, our results outperform GPS by a factor of six. Moreover, our secure multi party computation based anonymization system scales linearly with the trace length ultimately yielding a performance of four to six traces per minute on the smallest Amazon EC2 instances.

Acknowledgments

I am really happy that I found a place at ComSys to write this thesis in the best imaginable atmosphere. It was a great time for me working at “the chair”. A big “Thank you!” goes to all people at ComSys, especially to my supervisors Jó and Henrik who did a great job supervising me during the development process of my thesis. I would also like to thank Prof. Dr.-Ing. Wehrle for accepting my thesis. Further gratitude goes to Prof. Dr. Rumpe for being second examiner. Further thanks go to the students at ComSys who tried my Android application and hence helped me collecting the majority of data for the system evaluations. I finally want to thank my girlfriend Michaela and my family. Your emotional support helped me a lot!

Contents

1

Introduction

Gathering data has always been a prominent topic, especially in combination with the current possibilities of the internet. Services that benefit from user-contributed data like OpenStreetMap (OSM) [1] evolved. Such crowd-sourced services have shown to compete well with their closed counterparts as we can see looking at OSM or Wikipedia. But what are the privacy implications involved in the contribution in open services? No matter which service a user wants to contribute to, it is advisable to think of the risks of being identified on the basis of contributed data. The consequences of being identified are often not obvious, thus, it is even more advisable to prevent identification. To stick with the example of OSM one conclusion that one can draw is the home and work address based on traces that often connect the same two points on the map. Then, we can conclude that this trace is probably created by a person living at one of the endpoints. In this thesis, we propose a system that prevents the possibility of drawing conclusions on user-reported location-bound data and allows for a privacy-preserving contribution to OSM by means of the concrete example of air pressure based altitude information.

There is currently a lack of possibilities for potential users to contribute to services like OSM without being exposed to e.g., record-linkage attacks [12]. Moreover, OSM is known to miss high precision altitude information which could enable OSM to be used for a new class of navigation applications [8]. In this thesis, we thus propose a system to facilitate the anonymous collection of elevation profiles while coping with user data anonymization and reidentification prevention. Our system is focussed on information bound to OSM nodes and thus can be extended to gather arbitrary location based data that stays is relation to OSM nodes.

In order to prevent contributor reidentification, one possibility is to make a single contributor indistinguishable from others such that a record-linkage attack cannot identify one certain person but only a set of k candidates from a certain data set. A portion of contributed data hence can not be traced back to a single person. We call a set with this property *k-anonymous* [24]. However, in order to establish k-anonymity on incoming data before it is received by a potentially untrusted collector requires a

trusted third party (TTP), i.e. some entity that gets in possession of contributed data from single persons and creates an anonymous set of data from it for final collection. There are existing approaches like [5, 16, 14, 15] that propose e.g. the *mix zone* where user pseudonyms are mixed when they meet at the same room and the *time-to-confusion* criterion where a path cloaking algorithm chooses those traces that can be released anonymously. However, all these approaches aim at an interactive use of location information and suffer from the necessity of a TTP that establishes or recognizes anonymity on the data set. In our scenario data is collected at an arbitrary point in time such that there is no timely connection between the collected data and the real event of measurement. We further do not need a TTP as this part is emulated using *Secure Multi Party Computation* (SMPC). With SMPC we use a set of *privacy peers* (PPs) that can collaboratively do the anonymization calculations on shares of incoming data without knowing the real input. An adversary would have to take control over more than 50% of the PPs in order to reconstruct the raw non-anonymized data.

We apply our system to the crowd-sourced collection of air pressure based altitude information. Previous efforts that aim at the creation of elevation profiles lack the precision that is needed to facilitate useful altitude aided applications, or request high amounts of money for the data. The biggest effort in this area is done by the NASA's Shuttle Radar Topography Mission [11] that features a a grid resolution of 30m for the USA and 90m worldwide. This set of data is not accurate enough to e.g. find the building entrance with the least altitude difference for a wheelchair driver. To overcome the lack of precise elevation profiles we checked for a crowd-sourced solution based on sensors found in current smartphones. In [13] is stated that altitude data from GPS receivers has an accuracy of $\approx 45m$. It is thus no feasible source for exact altitude information. The solution for this problem is found in air pressure sensors built into many modern smartphones. With the values from such sensors we are able to calculate very exact relative altitude differences between two locations and hence are able to create elevation profiles from paths users walk while carrying their smartphone.

In this thesis we propose a system to enable the anonymous collection of crowd-sourced location based data on the practical example of air pressure based altitude information. This thesis is composed of eight chapters where, after giving some *background* information, the *Related Work* is done in the area of anonymous data collection and proper achievement of altitude information. As a good starting point of what we want to achieve we give an overview over the problematics to be solved and the requirements that need to be fulfilled by our solution in the *Problem Statement* chapter. The chapter *System Design* firstly introduces the parts our system is composed of and which roles they play, i.e. where and how is the data collected, where is it processed and where is it collected. We especially explain in this chapter how anonymity is established before the data reaches the data collector. The chapter *Implementation* deals with the technical interaction between the components and the used libraries. Finally, we will evaluate the performance of the anonymization process and the quality of the gathered altitude information in comparison to exact reference data for the region of Aachen. After a short reiteration on the evaluation results the *Conclusion* points out the feasibility of our approach for anonymizing crowd-sourced data as well as discusses how our system can be applied in other areas of data collection for future work.

2

Background

This chapter lists and explains the used techniques and foundations of our system. We start with the introduction of *OpenStreetMap (OSM)*, its functionalities and use-cases. Next up follows an introduction to the barometric altitude formula, i.e an explanation how air pressure and altitude correlate with each other. We continue with a short explanation of the *Android Platform* and how it easily delivers the functionality to collect data among a large set of contributors. Further, we describe the idea behind *k-Anonymity* and illustrate its implications on collected data with a simple example. The last two parts of this chapter deal with *Secure Multi Party Computation (SMPC)* as an instrument to anonymize a large set of input data and *Bloomfilters* as datastructures that facilitate simple calculation of set-intersections and unions under SMPC.

2.1 OpenStreetMap

OSM is a freely accessible and editable collaborative map service that grows through user contribution. Users can utilize GPS traces to extend the map with new highways, paths, buildings and further details. Especially in urban areas OSM benefits from the amount of contributors and has developed great detail regarding streets and buildings. Unfortunately, the available positional information is limited to latitude and longitude for the majority of OSM nodes, elevation information is rare and if available only coarse [8]. The current way to attach elevation information to OSM is to use the *Shuttle Radar Topography Mission (SRTM)* data collected by the NASA which has a grid resolution of 30m for the USA and 90m worldwide [8, 11].

Using our system we are able to collect altitude differences between neighboring OSM nodes with a typical altitude error of 1.21 meters measured over 93 traces collected in the region of Aachen, enabling OSM to be a basis for elevation data based applications with high accuracy requirements.

Elevation information are, e.g., used in a set of new navigation based applications. Some possibilities for such applications are:

- Bicycle and hiking navigation with difficulty indicator
- Fuel saving routes in car navigation
- Easiest routes for wheelchair users

OSM is represented by a graph with multiple layers where each layer contains nodes that have relations with each other. Relations can for instance represent parts of streets, or buildings where a number of connected nodes mark the outer wall. More over, each node is able to hold arbitrary information about the position it represents such as the name of a shop or the affiliation to a parking lot.

An exemplary entry of a OSM node in OSM-XML format is depicted in Listing 2.1.

```
<node id='25944' visible='true' \\
    lat='52.50711572260516' lon='13.374729176299503'>
  <tag k='anchor' v='stairs' />
  <tag k='direction' v='both' />
  <tag k='indoor' v='yes' />
</node>
```

Listing 2.1 OpenStreetMap node example.

The node is identified by its unique *ID* and has information about latitude, longitude and visibility. Further information are stored using tags which hold arbitrary key/value pairs. In this case the node is part of an indoor staircase. To create paths or buildings from multiple nodes OSM uses relations that declare what nodes belong to a group and how they are connected with each other.

A simple relation is depicted in Listing 2.2.

```
<way id='23186' visible='true'>
  <nd ref='19006' />
    ...
  <nd ref='16358' />
  <tag k='direction' v='both' />
  <tag k='indoor' v='yes' />
  <tag k='section' v='stairs' />
</way>
```

Listing 2.2 OpenStreetMap relation example.

Relations have *IDs* and list the nodes they own as *members*, where the members form a path in the order of appearance in the relation. Relations can also have tags for adding further information.

Nodes and relations are the core components of a OSM file representing a part of the global map. We aim to complement both nodes and relations with information about altitude differences between neighboring nodes, enabling the possibility to compute the difference in altitude for arbitrary paths in OSM.

Name	Symbol	Value
Sea level	h_0	0
Temp. gradient	a	$0,0065K$
Temp. at sea level	$T(h_0)$	$288,15K$
Molar mass	M	$0,02896$ kg mol^{-1}
Gravity constant	g	$9,807ms^2$
Gas constant	R	$8,314JK^{-1}mol^{-1}$

Table 2.1 Constants used in barometric altitude formula, see [10]

2.2 Barometric Altitude

Although the possibility of achieving altitude information through so called *Pressure Altimeters* already exists for long time, it has just recently become available to a broad user base. More and more current smartphones are equipped with pressure sensors hence empowering the use of these values in crowd-sourced applications. Previously, the only way to gather altitude information with smartphones was to read the GPS provided altitude which suffers from vertical errors of $\pm 45m$ [13]. A major improvement over the GPS altitude is delivered by the barometric altitude. The barometric altitude is based on the fact that air pressure decreases with increasing altitude. According to [25, 10, 18] we calculate the air pressure for a given altitude as follows:

$$p(h) = p(h_0) \cdot \left(1 - \frac{a\Delta h}{T(h_0)}\right)^{\frac{Mg}{Ra}} \tag{2.1}$$

Where $p(h_0)$ is the pressure at sea level, a is the absolute atmospheric temperature gradient stating how much temperature changes per 100m altitude change, Δh is $(h - h_0)$, $T(h_0)$ is the temperature at sea level in Kelvin, M is the molar mass, g is the gravity constant and R the gas constant. As in our setting we want to calculate the altitude h from a given pressure p we need to solve this equation to h as follows:

$$h(p) = -\frac{\left(\left(\frac{p}{p(h_0)}\right)^{\frac{1}{\frac{Mg}{Ra}}} - 1\right) \cdot T(h_0)}{a} \tag{2.2}$$

With the constant values according to Table 2.1 our equation looks as follows:

$$h(p) = -\frac{\left(\left(\frac{p}{1013,25mBar}\right)^{\frac{1}{5,255}} - 1\right) \cdot 288,15K}{0,0065K} \tag{2.3}$$

Using absolute heights calculated by this formula requires some caution as air pressure does not only change with altitude but also due to meteorological circumstances. There are two possible issues:

1. High- and low-pressure areas have influence on air pressure by definition and cause barometric altitude determination to yield different results in the orders of several 10 mbar per 24h for the same spot.

2. Moving between two locations incorporates that these locations may have different meteorological states and thus disrupt air pressure based altitude measurement.

We can see in the evaluation chapter that these issues do not have a noticeable impact on the operation of our system.

2.3 Android

Android is Google's operating system for the smartphone sector. It comes with wide support for diverse smartphone models and a comfortable Software Development Kit (SDK)[1]. Further, Android is an open system that facilitates free application development for all developers. Just like the creation of maps, the collection of altitude information requires a large set of contributors in order to gain a usable map coverage in a feasible amount of time. We thus decided to create an *Android* app that in first place empowers us to release this app to a very large user base and in second place provides us with an easy to use development platform to access the variety of sensors found in current smartphones. We use the smartphones GPS module to get the users current position, and we use the barometer to collect the current air pressure for this position. Both, GPS and barometer are easily accessible through the APIs provided by Android.

2.4 k-Anonymity

In many cases subsets of collected data can be traced back to the person who originally contributed it. This is especially valid regarding the collection of location based data [16] as done in our system. Even though contributed traces are not connected with a user identity it is possible to re-identify the user by recognizing her workplace or home [16]. Another possibility for user reidentification is shown in [12] called *record-linkage attack* where the user contributed data is correlated with publicly available information that - if a match is found - possibly reveals the user's identity. Hence, the contributed traces are so called *quasi identifiers* i.e., they do not directly identify the user but they can in combination with further knowledge like e.g. home or work addresses. The combination of those two information is thus a *unique identifier*. A way to avoid user reidentification is to make a user indistinguishable from others, i.e. there should be no way to definitely trace a portion of the final data set back to a certain user.

Figure 2.1 illustrates a 3-Anonymity scenario where three paths of three different users are merged:

[1] https://developer.android.com/sdk/index.html

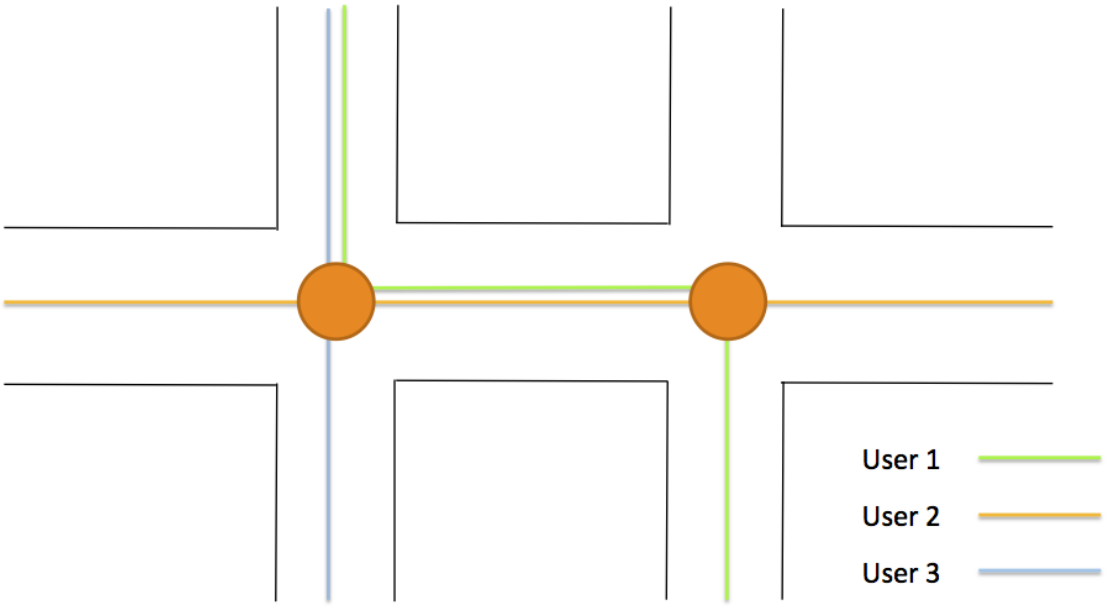

Figure 2.1 3-Anonymity: In the merged set of all paths we cannot distinguish which user went where at crossings.

When merging multiple traces like depicted in Figure 2.1 the crossings are *anonymization points*. At such points we cannot differentiate anymore between the users that passed there. In the example we can for instance not tell if *User 2* coming from the left turns to the left, to the right or passes straight at the crossing making her indistinguishable from *User 1* and *User 3*. Depending on the anonymity requirements we can choose how many intersections are needed until the union of the traces is released to the data collector. We speak of $k - Anonymity$ if every dataset entry related to a person is indistinguishable from at least $k-1$ other entries from $k-1$ other individuals [24].

2.5 Secure Multi Party Computation

Secure Multi Party Computation (SMPC) deals with the collaborative computation of a known function $\mathcal{F}(x_1, \ldots, x_n)$ in a secure way by a set of peers. We call the computation *secure* if none of the peers gain knowledge about the private inputs $x_1, \ldots, x_n$, and the computed result is guaranteed to be correct even if some players cheat [9].

In our system SMPC provides a secure way to compute anonymized sets of location traces that prevent the de-anonymization of the contributors at the data collectors site. Meaning, $\mathcal{F}$ is a function that takes several traces as input, checks them for intersections and if such exist, outputs the union of these traces, otherwise nothing is outputted.

The common participants in a SMPC scheme are *input peers (IPs)* and *privacy peers (PPs)* where each IP i provides one input x_i that is shared to all PPs using a *Secret Sharing Scheme*. The computation on the inputs is done by the PPs. To realize these

computations in a secure way we utilize two approaches of *Secret-Sharing* in SMPC, namely **Shamir's Secret Sharing** [22] and **Paillier Threshold Encryption** [20].

2.5.1 Shamir's Secret Sharing

Secret-Sharing is the technique used to securely share an IP i's private input x_i across the m PPs. It thus creates a *sharing* $[x_i]$ containing m *shares*, where $[x_i]_j$ is the j-th share of input i which is sent to PP j.

As the inputs should stay private a Secret-Sharing scheme must create shares that for themselves give no clue of the original input. To reconstruct an input x_i a previously determined amount of shares from sharing $[x_i]$ is needed, i.e. we talk of a (t, m) sharing if at least $t + 1$ out of m PPs need to collaborate and exchange their shares in order to be able to reconstruct the input x_i.

In Shamir's Secret Sharing a share of a (t, m) sharing is a point on a random polynomial of degree t. To share x_i IP i creates a random polynomial $f_{x_i}(x) \in \mathbb{Z}_p[X]$ of degree t where $x_i = f_{x_i}(0)$.

It then shares m random points $(x, f_{x_i}(x))_{x \neq 0}$ across the PPs. Because of the chosen degree t, we need at least $t + 1$ points to reconstruct the polynomial f_{x_i} using LaGrange interpolation such that we are able to find $x_i = f_{x_i}(0)$. [22]

Shamir's Secret Sharing scheme allows for basic mathematical operations on the input such as addition and multiplication [22]. To add different inputs, the PPs just need to add their shares, meaning they calculate:

$$[x + y]_i = [x]_i + [y]_i \tag{2.4}$$

Multiplication of an input with a scalar value can be done offline without the need of communication between the PPs as follows:

$$[x \cdot s]_i = [x]_i \cdot s \tag{2.5}$$

Multiplying two inputs works in the same manner as adding inputs besides that multiplying shares increases the degree of the polynomial [22]. Hence, the PPs are required to do an extra communication round where they share their values $z_i = [x]_i \cdot [y]_i$ and calculate a share $[xy]_i$ over a polynomial of degree t.

We use Shamir's Secret Sharing to share bloomfilters of the user-contributed traces and discover intersections between them. Intersecting traces are merged which increases the anonymity of the resulting set by one, i.e. the result set contains data from one more individual that is indistinguishable from the others. Individuals are indistinguishable as we cannot tell the path each individual took at the intersection points.

2.5.2 Paillier Threshold Encryption

Paillier Threshold Encryption (PTE) is a secret-sharing scheme based on the discrete logarithm problem [19]. The crypto system consists of a public key k_{pub} that is used by IPs to encrypt their input x_i such that $[x_i] = E_{k_{pub}}(x_i)$ and a private key $k_{priv} = (k_{priv_1}, \ldots, k_{priv_m})$ where m is the number of PPs. Hence, each PP holds one part of the private key. The creation of the key-pair needs three parameters that are b, the length of the keys in *bit*, l the amount of parts the private key should have and t the threshold that determines how many parts of the private key are needed to reconstruct the secret. We thus speak of a *threshold encryption* (t, m) where t out of m parts of the key are required to reconstruct the secret. Our system assumes the existence of a *trusted dealer* that creates a key-pair and distributes the respective parts of it to the IPs and PPs.

Shares of a PTE have the following properties [19]:

- **Additive homomorphic:** Secrets can be added by adding their shares.
- **Self-Blinding:** Shares can be re-randomized without affecting the secret itself.

The first property enables us to perform distributed calculations on the secret. More importantly, the second property is a fundamental part of the shuffling process we use. In this process, we randomize the node order of a traces. Therefor, each PP applies a random permutation to the node order and re-randomizes the node's shares to prevent the next PP from recognizing and reordering the nodes.

2.6 Bloomfilters

As seen in the SMPC section, calculations under SMPC are based on simple primitives like addition and multiplication which allow us to create any computable function upon them. In order to perform more complex set-intersection and set-union operations on traces we use SMPC in combination with the *bloomfilter (BF)* representation of a trace. A BF is a probabilistic datastructure that can hold any hashable data. A simple (non-counting) BF is a bit-array defined by its length n and the amount of hash functions m used to insert an element. The operations on BFs are:

Inserting an element e into a BF f involves hashing e m times using a seed-based hash function with m different but fixed seeds. To insert e into the bit-array we set the bits on positions $h(seed_1, e) \bmod n$, $h(seed_2, e) \bmod n$, $\ldots$, $h(seed_m, e) \bmod n$ to 1.

Checking the existence of element e in the BF is done by calculating the hashes and checking if the corresponding bits are already set to 1.

Due to the hashing BFs provide no way to reconstruct the elements that they contain. Further, BFs are probabilistic datastructures, meaning the check for a certain element may yield false-positives. With rising amount of added elements also the number of 1s in the BF bit-array rises, leading to an overpopulated BF, where:

- Adding further elements will have no significant change to the BF anymore as most bits are already set to 1.
- There is a high probability that an element hashes to positions that have been set to 1 by other elements, hence yielding a false-positive if checked for existence.

It is therefor important to find feasible values for the BF length n and the number of hashes m. This holds especially for our scenario where bloomfilters are handled in a SMPC system where computations are expensive. A proper evaluation of the best value for our system is given in the evaluation chapter.

3

Related Work

Privacy-preserving data collection is a well researched topic where many protocols have been elaborated and many techniques have been developed to realize them. Our system is composed of the following known techniques. We use secure multi party computation (SMPC) [9, 22, 20], user anonymity by *unlinkability* [6] i.e, anonymizing the source of contributed data, user anonymity by *untraceability* [24, 5, 16, 14, 15] i.e., preventing the drawing of conclusions from contributed data content, and utilization of air pressure to yield relative altitude differences [21, 26]. In this chapter we classify our system in the context of the mentioned techniques and talk about the advantages and disadvantages.

3.1 Privacy-Preserving Data Collection

Privacy-Preserving data collection deals with two major concerns of contributor anonymity. The first is contributor *unlinkability* which means that the data collector is not able to determine who is the source of a certain piece of contributed data. In this concern the data content is disregarded. The second concern deals with data content *untraceability* where an untrusted entity could process the data content in order to find hints on the identity of the recording contributor.

3.1.1 Data Source Unlinkability

Brickell et. al. elaborate on a protocol in [6] that allows for a data miner to collect responses from a set of respondents while eliminating the connection between response and respondent i.e., the data miner has no knowledge about which response belongs to which respondent. To achieve unlinkability they shuffle the responses under encryption before releasing them to the data miner. Each respondent has a permanent primary public key pair that is registered with a certification authority,

and a secondary public key pair that is freshly generated for each shuffle process. The shuffling process works as follows for each respondent:

- Generate a secondary key pair.
- Sign the secondary public key with the primary key and send it to the data miner.
- The data miner forwards the public key to the other respondents.
- The respondent encrypts his data with the miners public key and all secondary public keys of the other respondents.
- Repeat the previous step with the other respondents primary public keys.
- Send the cipher text to the data miner.
- The data miner forwards all cipher texts to all respondents.
- Each respondent removes her layer of encryption by decrypting all cipher with her primary private key.
- The respondents shuffle the order of cipher texts and send them back to the data miner.
- The data miner forwards all cipher texts to all respondents.
- Each respondent verifies that her piece of encrypted data is available in the shuffling.
- The respondents check the signatures of the shuffled pieces.
- If signatures are ok, all respondents send their secondary private key to the data miner.
- The data miner can successfully decrypt the shuffled data.

Their protocol works in the absence of a TTP as the response shuffling is entirely handled among the respondents. In order to prevent respondents from seeing other respondents data they use a IND-CCA2 encryption scheme that allows for an arbitrary order in applying and removing encryption on responses using different keys. The advantages of Brickell's approach are on one side the linear message complexity where each of n participants sends $O(n)$ messages and on the other side the absence of a TTP. Moreover, they show that shuffling is an appropriate way to establish unlinkability. We implement shuffling to establish unlinkability in our system as well, but still rely on a SMPC emulated TTP. Brickell's approach assumes that the respondents responses are not linkable by content [6] which is not given in our system. Hence, we have to take additional measures to overcome the second major concern of contributor anonymity i.e., data content *untraceability*.

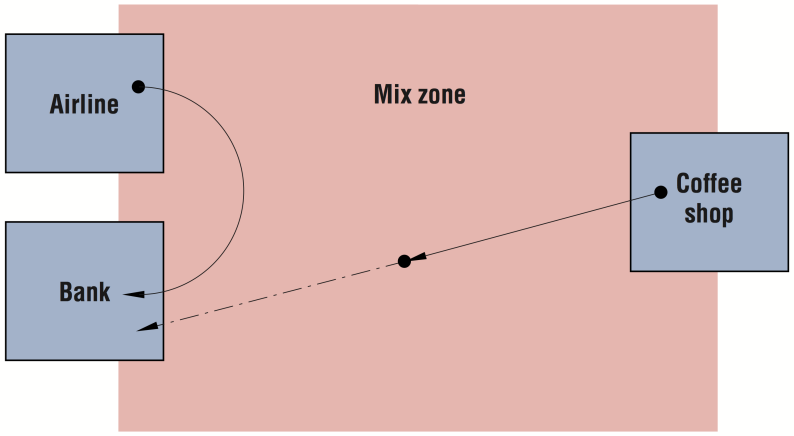

Figure 3.1 The mix-zone principle covers a user's trace in areas with other users.

3.1.2 Data Content Untraceability

The most general approach to establish *untraceability* on data content is k-anonymity as presented in [24]. We speak of k-anonymity if a contributor is indistinguishable from k-1 other contributors in a set of data. However, establishing k-anonymity has to be defined separately for each kind of data content.

In our scope we need to apply k-anonymity to locations and location traces. For instance, *Beresford and Stajano* [5] introduce *mix-zones* to establish anonymity for users moving in the same area. Their system provides location information about users to applications that register with the system. To trigger location information, they separate space into application areas and mix-zones where application areas are points of interest for the registered applications and mix-zones are areas in between where multiple users move. An example of a mix-zone is depicted in Figure 3.1. The larger the amount of users in the mix-zone, the larger is the anonymity set as the application can not tell which users moves on to which application area in a mix-zone. A very similar mechanism is used in our system where intersections of traces from multiple users correspond to mix-zones because we can not tell where which user joins and leaves the intersection. Hence, we establish k-anonymity for k intersecting traces from k contributors.

Hoh et. al. [16] focus on collecting live traffic information using GPS traces. They introduce the *time-to-confusion* criterion that shows an adversary's time to follow an anonymous user sticking to a specified level of confidence. Their system is composed of the following components:

- Location trace contributors.
- Anonymity establishing path-cloaking algorithm that decides when to release user locations based on the time-to-confusion criterion.
- Trustworthy party to apply the path-cloaking algorithm.

Based on the time-to-confusion criterion their uncertainty-aware path-cloaking algorithm chooses which user locations can be released while maintaining user anonymity.

However, their approach is based on the existence of a trusted privacy server that receives location updates from all users and applies the path-cloaking algorithm. Moreover, they aim for a live recording of location-change events which makes higher demands on anonymization as live data can be subject to the *target attack* where one certain user can be reidentified by following her location updates.

In our system we anonymize complete traces such that there is no timely connection to their recording time. We further emulate the trusted third party using SMPC rendering a single privacy server unnecessary.

3.2 Barometric Altitude

Measuring altitude differences using air pressure is commonly used in altimeters [2]. In recent time, barometers shrinked and are nowadays included in many smartphones. *Parviainen* et. al. [21] discuss the integration of barometers in devices for personal navigation. Personal navigation includes but is not limited to the determination of the current floor in buildings as well as finding the correct position on overlapping roads and highway crosses. *Parviainen* et. al. give a thorough evaluation of error sources for air pressure measurements in cars. In particular, they state that different car blower settings have an impact on air pressure inside cars as well as passing tunnels. Their evaluation shows differences of up to 4m in altitude measurement depending on the blower setting. The impact in a still standing car, and impact caused by passing tunnels is shown in Figure 3.2.

Parviainen et. al. use a *reference* and a *mobile* barometer for their measurements. The reference barometer must be located at a known altitude over sea level. This way, they are able to calculate absolute altitudes from the mobile barometer. To determine the error of high distances between reference and mobile barometer they place two barometers with 10km distance and compare their measurements. They yield a maximal aberration of 0.1mbar between both accounting for an altitude difference of 80cm [21]. However, we have to take into consideration that a repetition of this experiment under different weather circumstances will yield different results. Overall, *Parviainen* et. al. conclude that air pressure is suitable to calculate precise altitudes. To make results even more accurate we need to incorporate measures to circumvent the mentioned error sources inside cars.

Zhu et. al. [26] introduce an accurate measurement process for barometric altitude incorporating correction of temperature drift errors and digital filtering. Their tests yield an accurate operability of MEMS sensors in the range of -750m up to 10000m altitude with a resolution of $\leq \pm 0.62m$. Their measurement system has the following features:

- Temperature drift correction
- Bias and lag compensation
- Digital filtering

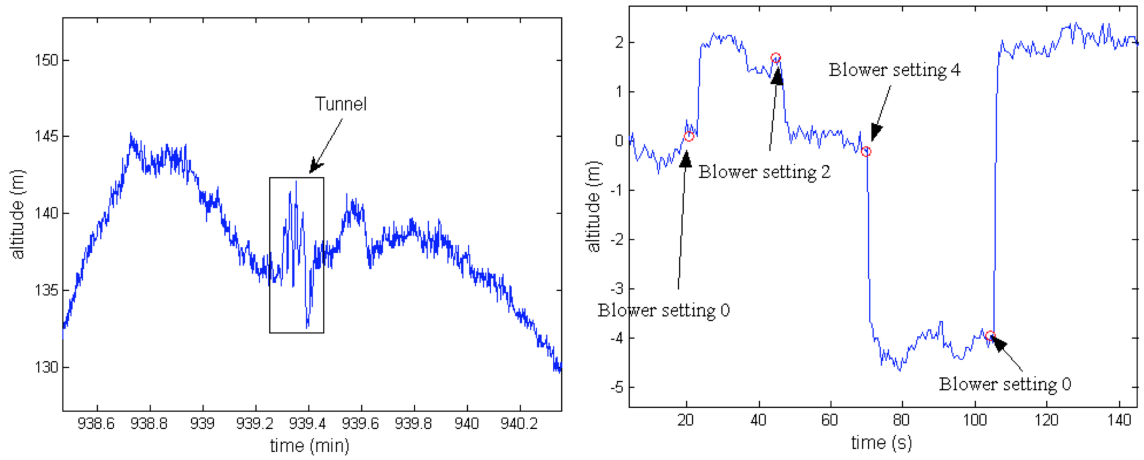

Figure 3.2 a) Influences on air pressure when passing tunnels while driving a car, from [21]. b) Influence on air pressure of different blower settings in the car, from [21].

Zhu et. al. state that barometer accuracy is heavily affected by temperature cross-interference. This interference is even able to affect the linearity of a barometer. To overcome this issue they propose to perform an offset correction of the barometer values. Moreover, they apply a quadratic compensation (second order polynomial approximation) based on the correction values found in the barometer sensor's registers.

In order to compensate the barometer results during a current temperature change, they fit the air pressure error curve during heating and cooling using the least squares method.

Digital filtering is their last measure to improve the barometer values. They use median and arithmetic mean of sliding windows in order to flatten noise while keeping the original signal.

Together with the accuracy evaluation from *Zhu* et. al. we can conclude that air pressure and the resulting barometric altitude are suitable to be used in our distributed system for privacy-preserving creation of elevation profiles.

4

Problem Statement

When looking at current map services like OSM or Google Maps, they give us a two dimensional representation of the world. We are able to navigate from point A to point B, but can we do it in the most efficient way for any case? Obviously, this depends on the means of travel, i.e. driving by car, by bicycle or if we go by foot. For instance, when planning a bike tour different routes will yield also different levels of difficulty for the driver depending on the occurring altitude differences. The same holds for pedestrian navigation. Unfortunately, the common mentioned map services lack precise information about elevation changes [8] such that useful applications - like difficulty dependent navigation for pedestrians or wheelchair users - depending on precise elevation profiles are not realizable at the moment.

Further, investigating the lack of precise altitude information shows that public and open sources for elevation profiles are rare. The approaches currently being used to collect elevation profiles either suffer from a inadequate resolution as seen in SRTM [11] or request significant amounts of money for access, e.g. communal LIDAR data where elevation profiles are created from the reflections of a laser pointed from a plane to the ground in the example of the region Aachen, Germany.

Fortunately, with rising distribution of air pressure sensors in current smartphones we have an instrument to create elevation profiles based on air pressure differences that is available to a broad community of users. These users could contribute to the collection of air pressure based elevation profiles.

Location traces from single users allow for a reidentification of the contributing user as shown in [12]. Hence, users contributing to such a system may worry about their privacy e.g. if in doubt about the trustworthiness of the data collector or about third parties intercepting personal positional data. As soon as a reidentification succeeds for a user it is possible to extract movement profiles of her from the contributed data which heavily violate the user privacy. Especially in current times users ask themselves what happens with data they contribute and what conclusions can be drawn from them about their person.

Establishing anonymity on a given set of data usually requires a trusted third party to perform the anonymizing operations on the data. This trusted party gets in possession of data that origins from single users and thus is able to trace the data back to these users. Unfortunately, it is difficult to find a mutually trusted party in a system that expects the contribution of a large amount of users.

4.1 System Requirements

To solve the mentioned problems our system must thus meet the following requirements:

Anonymity To guarantee user privacy the system should provide a facility to anonymize data. To anonymize data we have to consider the anonymization of the content as well as the anonymization of the data source. The anonymized data should not be vulnerable to de-anonymization attacks while still containing the essential data about altitude differences. In particular, we expect from anonymized data that the original data sources are untraceable i.e., we cannot tell who contributed the data of an anonymized set of data. Additionally, we should avoid a single point of trust as it is complicated for a large set of loose contributors to agree on such a mutually trusted party.

Utility After anonymity is established on the input data, the data should still contain the information required to create elevation profiles. Moreover, the anonymization process should not make collected data less accurate. Further, the system must yield better results than those from data that is currently publicly and cheaply available e.g. in comparison with SRTM data. The accuracy should suffice to enable useful application of the data in navigation applications for e.g., pedestrians, bicycle drivers or wheelchair users.

Scalability The system must be scalable in the number of contributors, i.e. the collection of data should happen in distributed fashion and data handling must be computationally feasible for large amounts of contributors.

Coverage To achieve a reasonable coverage with elevation profiles, the system should be easily deployable i.e., including additional contributors should not require any changes in the running system.

4.2 Adversary Model

We consider all participants to follow the honest-but-curious adversary model [17], meaning they follow the protocol but may try to draw conclusions from the protocol transcript in order to de-anonymize single data sources. We assume this adversary model as the processed information would usually not legitimate the costly dedication of means to yield de-anonymized data. Thus, we need to avoid in first place that a system participant is able to get in possession of input data that allows to be traced back to a certain IP. Access to data from a single source alleviates the de-anonymization process by facilitating attacks like *home identification*, *data matching* [16] and *record linkage* [12].

5

System Design

In this chapter, we introduce the parts our system is composed of and explain their function and interaction with each other. The overall design is best explained by first considering which parties are involved in the privacy-preserving data collection of altitude information. Those are in the order of data flow:

1. The data sources (Input peers)
2. A cryptographically emulated trusted third party (Privacy peers)
3. The data sink (Collector peer)

As shown in Figure 5.1 the desired data is gathered by many *input peers (IP)* that each want to stay *unlinkable* and *untraceable*, as explained in the anonymity requirements in Chapter 4, in the final data set at the *collector peer (CP)*. To achieve this we consider a cryptographically emulated *trusted third party (TTP)* to perform anonymizing operations on the data of the IPs before it is forwarded to the CP. As a single mutually trusted party is usually hard to agree about, the TTP is represented by a set of *privacy peers (PP)* that can collaboratively perform computations on shares of the input data without actually knowing the plain input data. After the computations are done a majority of PPs collaborates and reconstructs the results of the computations. This way the PPs can only see the anonymized result of the computation which is released to the CP.

In our case, the data transported between IPs and CP contains location information and altitude differences between neighboring OpenStreetMap (OSM) nodes. Thus, it possibly reveals the identity of the person behind the IP [5, 14, 15]. To mitigate the identification of single data contributors and their roaming profiles, our TTP creates trace bunches of k intersecting traces yielding *k-Anonymity* (see Chapter 2) for the data that is finally released to the CP.

The two main parts *privacy-preserving data collection* and *extraction of altitude information from air pressure data* are explained in detail in the following sections.

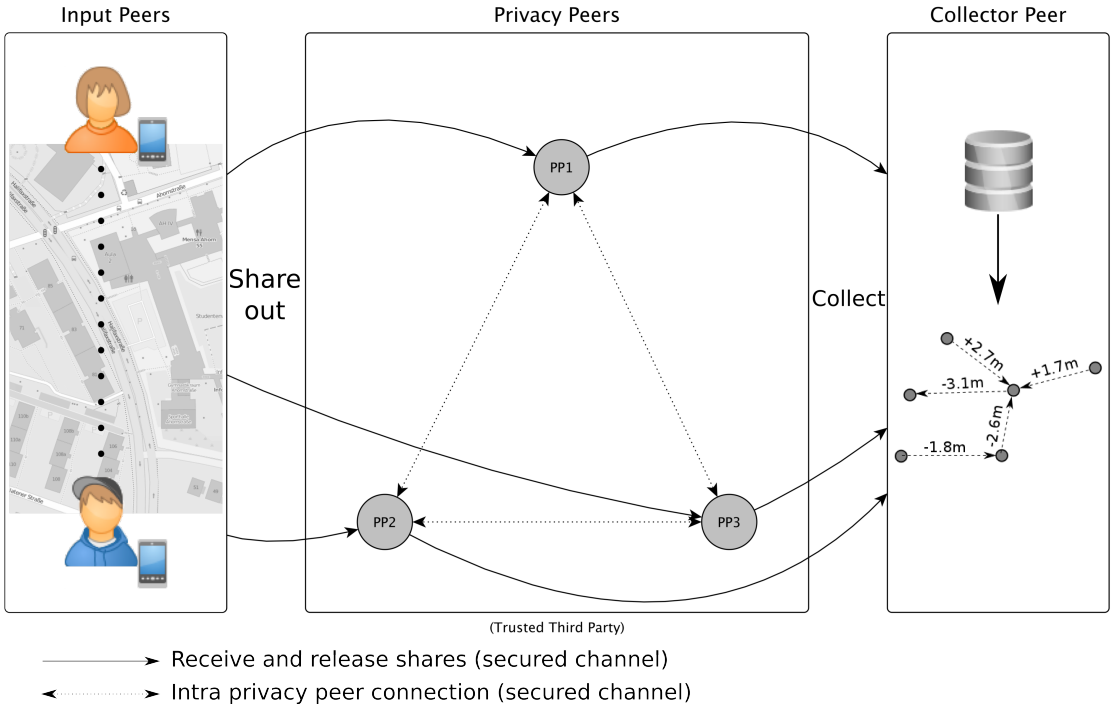

Figure 5.1 Coarse system overview showing input peers, privacy peers and collector peer.

5.1 Privacy-Preserving Data Collection

To realize privacy-preserving data collection we use *Secure Multi Party Computation* (SMPC). SMPC enables us to emulate a TTP that is able to anonymize the data contributed by the users/IPs. Our protocol anonymizes contributed data in the following two ways as required in Chapter 4:

- **Unlinkability:** Data is reported anonymously such that the CP cannot trace back which data got reported by whom.
- **Untraceability:** Data gets anonymized such that the CP cannot draw conclusions from the data i.e., personal information about the users that created the traces.

In this section, we introduce the process how we ensure unlinkability as well as untraceability and the calculations done on each kind of peer i.e., those on IPs, PPs and the CP.

5.1.1 Data Collection and Preparation

IPs are those players in our system that gather environmental data, preprocess it and then forward it to the PPs. In particular, the collected environmental data is a tuple of the current location l determined by GPS together with the current air pressure value p in millibar (mBar) for that location. Where l is itself a tuple containing *latitude* and *longitude*. We call tuples of locations that have been consecutively

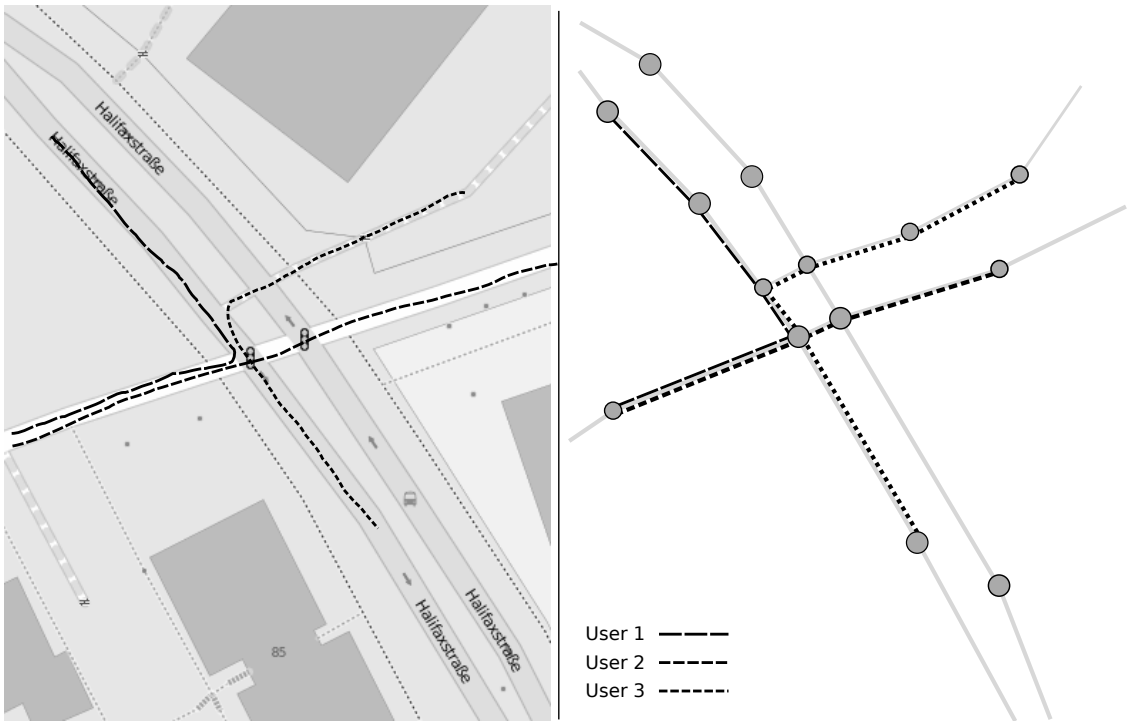

Figure 5.2 The left hand side shows raw traces where users meet at the same crossing but do not necessarily intersect with each other. The right hand side shows discretized traces based on OSM nodes where all users passing a crossroad intersect with each other and thus are indistinguishable at this point regarding their location.

passed by the user a raw trace T_{raw}, such that a raw trace of length n is defined as follows

$$T_{raw} = ((l_1, p_1), (l_2, p_2), \ldots, (l_i, p_i), \ldots, (l_n, p_n)) \tag{5.1}$$

where $l_i = (l_{i_{latitude}}, l_{i_{longitude}})$. We require the traces to fulfill the following requirements in order to establish k-anonymity:

1. They must have determined points where they can intersect e.g., on crossroads.
2. They must be indistinguishable at the intersection point i.e., we should not be able to determine from where a user comes (approaching the intersection) and where she goes (leaving the intersection)

Having determined points on a discretized map enables us to let traces intersect which would normally have no intersections according to their GPS coordinates. This is especially useful on crossroads as shown in Figure 5.2. We discretize the map to a graph of highways, paths and crossings extracted from the OSM database. Within this graph intersections can only occur at OSM nodes, e.g. user traces passing the same crossing are considered to intersect with each other even though their GPS locations may have no intersection. Figure 5.2 illustrates the benefits of a discretized graph in comparison to raw traces. It is now computationally easy to find intersections of two traces as we only need to check if two traces contain the

same OSM node - which is represented by an integer ID. For the sake of establishing k-anonymity a discretized map hence determines the amount of nodes where intersections can occur as well as simplifies the representation of a trace to a list integers.

The next requirement states that traces must be indistinguishable in any manner at their intersection points. This means that all properties of the two traces must be identical at this point otherwise these properties could be used to distinguish them from one another. In particular, the air pressure value is the problematic property at intersections, i.e. it differentiates the intersecting traces. As intersecting traces have most probably been recorded at different times the air pressure value is likely to differ at the intersection point. An attacker could use this circumstance to determine which user went where after the intersection, hence, the intersection is not an anonymizing property anymore. We solve this issue by further processing the raw trace on the input peer into a result trace T_{res} looking as follows:

$$T_{res} = ((n_1, d_{1,2}, n_2), \ldots, (n_i, d_{i,i+1}, n_{i+1}), \ldots, (n_{m-1}, d_{m-1,m}, n_m)) \tag{5.2}$$

T_{res} holds $m-1$ 3-tuples where m is the number of visited OSM nodes. Each tuple is the relation between a source node n_i and a destination node n_{i+1} that have been visited consecutively. The tuple also contains the altitude difference $d_{i,i+1}$ between these nodes. The OSM nodes in T_{res} can easily be found by determining the nearest OSM node for each location from T_{raw} and afterwards compressing multiple consecutive occurrences of the same OSM node to one occurrence as consecutive locations likely yield the same OSM node. These operations are entirely processed on the user's device such that the GPS locations never need to leave the device.

We now have a representation for the collected data that fulfills the mentioned requirements of determined intersection points and indistinguishable intersection points. Moreover, we determined the points on the map where altitude differences can be meaningfully calculated i.e., between connected nodes on the OSM graph.

The final step done by the IP is to share out the traces T_{res} among the PPs for further handling. We use two kinds of shares each holding different information from T_{res}:

- **Shamir's Secret Sharing** is used to share:
  1. A bloomfilter containing all OSM node IDs of the trace
  2. m bloomfilters each containing exactly one OSM node ID from the trace
- **Paillier Threshold Encryption** (PTE) is used to share:
  1. All values from T_{res} as a list

Given a trace T_{res} as shown above our Shamir shared information T_{res}^{Shamir} for PP i looks as follows:

$$T_{res,i}^{Shamir} = ([BF(n_1, \ldots, n_m)]_i, [BF(n_1)]_i, \ldots, [BF(n_j)]_i, \ldots, [BF(n_m)]_i) \tag{5.3}$$

We see that $T_{res,i}^{Shamir}$ contains a bloomfilter $BF(n_1, \ldots, n_m)$ holding all node IDs and multiple bloomfilters $BF(n_i)$ each holding the single node ID of the ith node in the trace. The bloomfilter representation supports the finding of intersections between traces. Taking a bloomfilter carrying all node IDs from trace t_1 we can check against the m bloomfilters of another trace t_2 and yield if one or more of t_2's node IDs is contained in t_1. If that is the case we know that t_1 and t_2 have most probably an intersection. Nevertheless, we have to care of false-positives because bloomfilters are probabilistic datastructures as stated in Chapter 2. We use Shamir's Secret Sharing to share the bloomfilters because checking if a bloomfilter-represented ID is contained in another bloomfilter is computationally cheap and only involves calculating the intersection (the bitwise AND) of the two bloomfilters, i.e. calculating the bitwise multiplication, and counting if the number of 1s in the intersection equals the number of hash functions. The bitwise AND used in bloomfilter intersections and the bitwise OR used in bloomfilter unions are realized as shown in Equation 5.4 and Equation 5.5.

$$a \wedge b = a \cdot b \ \mid a, b \in \{0, 1\} \tag{5.4}$$

$$a \vee b = a + b - (a \cdot b) \ \mid a, b \in \{0, 1\} \tag{5.5}$$

The complete process is shown in detail in Listing 5.1.

The sharing of traces only using Shamir and bloomfilters is not sufficient for our system because of the following reasons:

1. The values contained in a bloomfilter are not reconstructable as bloomfilters only hold hashes of the values. However, to collect real data the CP must be able to reconstruct the anonymized data.

2. The shared information from $T_{res,i}^{Shamir}$ serves only for the purpose of efficiently finding intersections between traces, it does not contain collectable data. We need a sharing scheme that allows firstly the reconstruction of plain data and secondly a shuffling of the tuples from T_{res} such that the contributors of the traces stay untraceable for the CP.

To satisfy these two requirements, we additionally share the tuples from T_{res} using PTE. As stated in Chapter 2, PTE allows for a threshold encryption (t, m) where t out of m PPs are required to reconstruct the secret. To encrypt the secret, the IP received the PTE public key K_{pub} from a trusted key dealer. We create the PTE shares $T_{res}^{Paillier}$ for PP i as follows:

$$T_{res,i}^{Paillier} = \{([n_1]_i, [d_{1,2}]_i, [n_2]_i), \ldots, ([n_j]_i, [d_{j,j+1}]_i, [n_{j+1}]_i), \ldots, ([n_{m-1}]_i, [d_{m-1,m}]_i, [n_m]_i)\} \tag{5.6}$$

Finally, the IPs total shared information are those shown in equations 5.3 and 5.6. In the following we refer to the total shared data of one trace as

$$T_{shared} = T_{res}^{Shamir} || T_{res}^{Paillier} \tag{5.7}$$

All the data is distributed among the PPs as shares and thus never leaves the user's device in plain. Assuming the PPs to follow the *honest-but-curious* adversary model they follow the protocol and hence non of them gets in possession of plain data. Even assuming single evil players among the PPs they are not able to reconstruct the shared secrets as long as they do not build the majority of PPs. In particular, regarding our system both sharing schemes are set that two out of three PPs are required to collaborate in order to reconstruct a secret. However, these variables can be easily modified depending on the required security and performance of the system. In Chapter 8 we benchmark multiple different settings of the sharing schemes and point out how performance trades off with security.

5.1.2 Data Anonymization

The PPs are responsible to anonymize the received traces before they are released. Anonymizing location traces is done by establishing *k-Anonymity* [24], i.e. merging multiple traces that have intersections with each other. Hence, the PPs first collect the received trace shares and then find intersections between the available traces. Those traces with intersections get merged and shuffled such that:

1. We increase the anonymity level by one for each merged trace
2. The PPs cannot distinguish anymore which tuples belong to which trace

An overview over our anonymization algorithm is shown in Listing 5.2.

A single trace t_1 as received by the PPs is considered to be *1-anonymous* because there are no other traces besides itself from which the trace could be indistinguishable. As soon as we can merge t_1 with an intersecting trace t_2 they are indistinguishable at their intersection point, i.e. we cannot tell which trace continues where. We then set t_1 to be the merged trace $t_1 \cup t_2$. Trace t_1 is now considered to be *2-anonymous* and is ready to be merged with further incoming intersecting traces. Depending on the anonymity requirements we can choose a k such that traces are held back at the PPs before they are released until they exhibit *k-anonymity*. As soon as we merged k intersecting traces the union of these traces features at least k intersections by construction. The resulting trace is composed of traces originated by k contributors and as we cannot tell which contributor took which direction at the intersection points, we call the resulting trace *k-anonymous*. However, choosing k is a trade-off between data anonymity and collection speed. Choosing a high value for k raises the risk of merged traces with less than k intersections being stuck at the PPs because we lack further intersecting traces to meet our anonymity requirement. Merging the traces t_1 and t_2 is done by appending the tuples of t_2 to t_1. The same holds for the bloomfilters with the exception that the bloomfilter holding all node

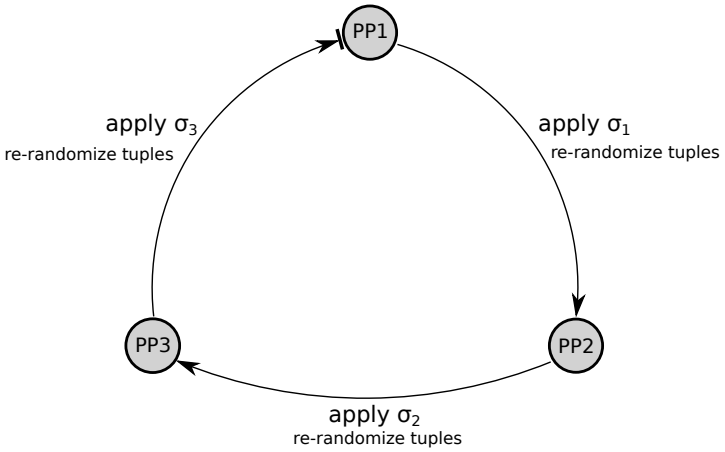

Figure 5.3 Shuffling: Each PP i once applies its own random permutation σ_i to the tuples and re-randomizes them by adding an encrypted value of zero to each part of the tuple. By adding the encrypted zero we ensure that the other PPs cannot recognize the tuples and are able to reconstruct the original order.

IDs must be the union of the bloomfilters from t_1 and t_2 holding their respective nodes.

At this point however, the PPs know which portion of the merged trace belongs to the original first trace - namely the first part - and which to the original second trace - namely the second part. With this knowledge honest-but-curious PPs are able to reverse the merging process and again yield *1-anonymous* traces. Moreover, the CP gets in possession of plain traces that are strung together and hence are easily decomposable by searching for not connected consecutive tuples. We prevent this situation by *shuffling* the tuples.

After the merging process, we let each PP i forward the tuples to the next PP $i+1$ while applying a random shuffling σ_i to them as shown in Figure 5.3. While shuffling and forwarding we have to keep in mind that the tuples are shared using Paillier Threshold Encryption. This means that the shares are identical for all PPs. Hence, PP $i+1$ is able to reconstruct the unshuffled order by recognizing the tuples. To avoid PPs from recognizing tuples we re-randomize them by adding [0] to each value of the tuples. We thus re-randomize the values of a tuple tup_i as follows:

$$tup_i = ([n_i], [d_{i,i+1}], [n_{i+1}]) \stackrel{re-rand}{=} ([n_i] + [0], [d_{i,i+1}] + [0], [n_{i+1}] + [0]) \tag{5.8}$$

Doing this, the real value held by the shares stays identical, but the share itself changes and hence cannot be recognized by the next PP in the round. The shuffling and re-randomization process is repeated m times where m is the number of PPs. After m repetitions the merged trace completed one round of shuffling among the PPs. We finally synchronize the final shuffled order of tuples from PP 1 across the remaining PPs. Now, none of the PPs is able to reverse the original order.

We have now merged two traces and randomized the order of tuples such that none of the PPs is able to establish a connection between the current order and the original order. Hence, PPs are not able to distinguish which tuple belongs to which trace.

Further, if the merged trace meets the anonymity requirements and gets released to the CP, the CP only receives a set of shuffled tuples determining the altitude differences between some neighboring OSM nodes.

Assuming some trace t_1 approaches the anonymity requirement of containing k intersecting traces, then this trace is released to the CP. Each PP i holds the PTE shares of t_1. These shares were created by the IPs using a (t, m) threshold encryption as explained in Chapter 2, where t out of m parts of K_{priv} are needed to reconstruct the secret. Hence, each PP performs a partial decryption of its trace-shares and sends them to the CP. In order to prevent an adversary who intercepts all partial decryptions on their way to the CP from reconstructing the result, all connections between any peers are secured using the *Secure Socket Layer (SSL)* protocol. Only the CP, who is in possession of an arbitrary part of K_{priv} and all partial decryptions, is able to remove the last layer of encryption from the partial encryptions. The final reconstruction of the secret at the CP only requires the following parts of K_{priv} [19]:

- t the number of required parts of K_{priv}
- *delta* the factorial of m
- c the combine constant calculated as the inverse of $4 \cdot delta \cdot delta$ modulo $p \cdot q$ where p and q are two primes randomly chosen during the key generation process

Hence, it is not important which part of K_{priv} the CP uses for reconstruction.

5.1.3 Data Collection

The CP is responsible for collecting the anonymized traces. It is considered to be an *untrusted* player in the system. Thus, it must not play a role in the anonymization process and it must only be able to reconstruct secrets of anonymized data. This means that the traces received by the CP must already fulfill our requirement to be *k-anonymous*. We have seen in the last section that both of these points apply in our system. The CP uses its part of K_{priv} to combine the partial decryptions that it received by the PPs. Thus, it yields a shuffled list of tuples determining the altitude differences between a set of OSM nodes. Though this list can be sorted such that tuples with common node IDs are considered to stem from the same trace, we cannot assume this on intersections anymore as traces are indistinguishable at their common nodes.

While collecting traces, at some time the CP will receive altitude differences for two neighboring nodes that are already known to have a certain difference according to an earlier report. It is likely that the new value differs slightly from the old one. However, in order to keep the elevation profile recent, our system treats the average of the five most recent reports to be the current valid result.

5.2 Extracting Altitude Differences from Air Pressure Traces

Air pressure is a commonly used metric to determine altitude in altimeters [2]. The atmospheric pressure decreases with increasing elevation such that we can use atmospheric pressure values of two locations to calculate the altitude difference between them. We refer to Chapter 2 for a detailed explanation of the barometric altitude formula. In this section, we concentrate on how we mitigate the influence of weather effects on our measurements.

Atmospheric pressure is subject to permanent change conditioned by weather. Where a good-weather area usually involves a higher air pressure, a bad-weather area involves lower air-pressure. This way, measurements taken at the same spot but at a different time yield different results. In order to make our system work properly we need to make sure that only air pressure differences induced by changed altitude are used in the calculation. Due to the permanent change we are not able to calculate absolute altitudes from air pressure without a recent calibration point. However, considering applications for pedestrian navigation it suffices to know altitude differences between the nodes in order to rate the difficulty of the path. We use the barometric altitude formula as shown in [10] and described in Chapter 2 to calculate an absolute altitude, biased by the current weather conditions for two locations as follows:

$$h(p) = -\frac{\left(\left(\frac{p}{1013{,}25mBar}\right)^{\frac{1}{5{,}255}} - 1\right) \cdot 288{,}15K}{0{,}0065K} \tag{5.9}$$

The constants used in this formula are explained in Chapter 2. We use this formula for each location l_i where p_i is the air pressure measured for this location. The altitude difference between two locations l_1 and l_2 is now $h(p_2) - h(p_1)$. Both absolute altitudes are obviously biased by weather effects, otherwise the absolute values were already accurate estimations for elevation. However, as we are only interested in altitude differences, we argue that the calculated difference between two points whose air pressure measurements have been taken in a timely close frame of up to two minutes, is exact because air pressure does not change quick enough to cause noticeable errors in short time frames as we will show in Chapter 8. Even though the weather itself can change within two minutes, this change is preceded by a slow change in air pressure. We will show in Chapter 8 that in a series of measurements recording the air pressure at a fixed location over several days, even the fastest change in a window of two minutes has no noticeable impact on the calculation of altitudes. Hence, we can use recorded traces of air pressure to calculate exact altitude differences if the time between two nodes does not exceed two minutes. The longer the time between two measurements, the higher is the influence that weather takes on the result.

Data: Trace t has:
t.BFcomplete $\leftarrow$ BF$(n_1, \ldots, n_m)$ where n_i are the nodes from t
t.BFsingle[] $\leftarrow (BF(n_1), \ldots, BF(n_m))$

Input: Traces t_1 and t_2
Output: *True*, if traces have common node (intersection), else *False*

```
// Bloomfilter length
BFl ← 10;
// Amount of hashes
BFhashes ← 2;

// Check for intersections between t1 and t2:
foreach BF_i in t2.BFsingle do
    // Create intersection bloomfilter
    for A_j and B_j being the jth bit from (BF_i) and (t1.BFcomplete) do
        // As the real values of the shares are either 1 or 0
        // we can emulate the bitwise AND by
        // multiplying the shares.
        IntersectionBF[j] ← A_j · B_j;
    end
    // Count the number of 1s in the intersection BF
    sum ← 0;
    for Bit b in IntersectionBF do
        sum += b;
    end
    // As we only summed up the bloomfilter shares, we still
    // need to reconstruct the real result.
    numberOfOnes ← reconstruct(sum);
    if numberOfOnes == BFhashes then
        // If all 1-bits of the single bloomfilter BFi are also
        // 1-bits in t2, then BFi is contained in t2 hence,
        // we have an intersection.
        return True;
    end
end
return False;
```

Listing 5.1 Using bloomfilters we can check if a certain node is with high probability included in a list of nodes.

Input: 1. Incoming traces from input peers: $t_1, t_2, \ldots$
Input: 2. Anonymity requirement: k
Result: Sets of alti. difference tuples $(n_{src}, d_{src,dst}, n_{dst})$ from k intersecting traces

```
traces ← [];
while True do
    if new trace t_i available && traces == [] then
        t_i.anonymity ← 1;
        traces.append(t_i);
        continue;
    else if new trace t_i available then
        for Trace t_o in traces do
            if t_i has intersection with t_o then
                t_i ← t_i ∪ t_o;
                t_i.anonymitylevel ← t_i.anonymitylevel + 1;
                traces.remove(t_o);
            end
            traces.append(t_i);
        end
    end
    for Trace t_o with t_o.anonymitylevel == k do
        releaseToCollector(t_o);
        traces.remove(t_o);
    end
end
```

Listing 5.2 Abstract overview over the anonymization process that creates k-anonymous traces from single input traces.

6

Implementation

Our system is composed of multiple loosely coupled parts as shown in Figure 5.1. The input peer (IP) is realized as an Android application that can be run on a large set of current smartphones with integrated barometer e.g., Google's Nexus 5. Hence, the code for the IP is written in Java. The privacy peers (PPs) as well as the collector peer (CP) are realized entirely in Python. To let IP, PPs and CP communicate with each other we use the *MessagePack*[1] serialization library that provides means of communication for a wide variety of languages. The remainder of this chapter delivers a closer look at the implementation of the input peer Android application, the privacy peers and the collector peer. We also introduce how existing libraries for SMPC like *SEPIA*[2] and the *Paillier Threshold Encryption Toolbox*[3] are used to create shares for Shamir's Secret Sharing and Paillier Threshold Encryption.

6.1 Android Application

The feasible operability of our system relies on a strong participation of users recording traces. Hence, we have chosen to create an Android application called *Elevation Logger* that enables a broad set of smartphone users to contribute. An overview of the application is depicted in Figure 6.1. The application consists of three *activities* which display the user the current status of the application, the currently measured values as well as a list of traces. In Android, *Activities* are classes that display content to the user and allow for interaction with the application. However, activities are only active if the application runs in foreground. Hence, the application also has a part running in background, the *SensorService*. The SensorService is an Android Service. Services have the ability to run in background and to be automatically started after the smartphone boots. We intent to keep the application working

[1] http://www.msgpack.org
[2] http://www.sepia.ee.ethz.ch
[3] https://www.utdallas.edu/ mxk093120/paillier/

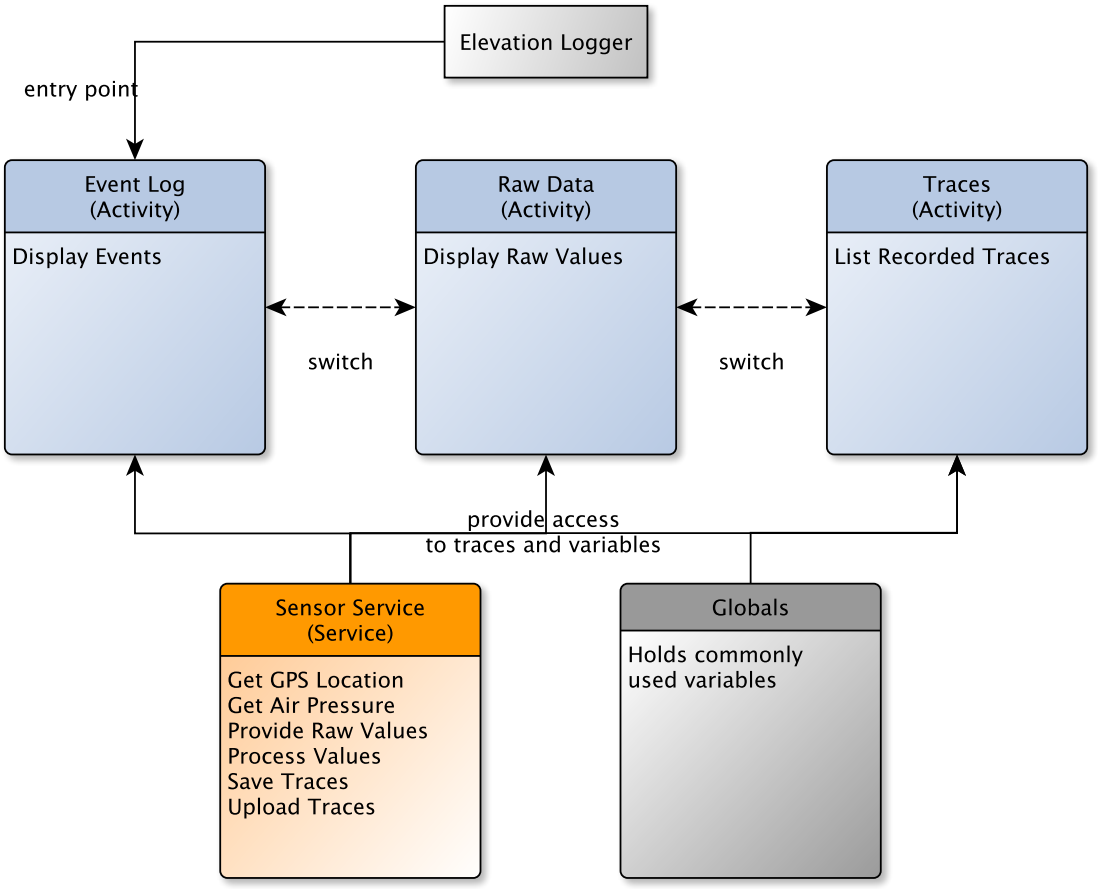

Figure 6.1 Elevation Logger is composed of three activities: a) The event log b) the raw value screen and c) the trace list. Background work is done using a service called SensorService.

with a minimum of attention required from the user, so services help us to keep the application working over smartphone reboots.

6.1.1 Application GUI

We think that users generally want to contribute, but will quickly quit contributing if the application demands too much attention. We focus on an application that interferes as little as possible with the regular user experience on Android i.e., the application only shows the measured values and uploaded traces, the user does not need to setup anything. Further, all work is done in the background such that once the application has started the user can forget about it. However, if no other applications using GPS are running, Android's icon for used location services indicates that the application is running. ElevationLogger features a simple interface shown in Figure 6.2 that provides the user with the following information:

1. The event log, showing when traces are recorded or uploaded
2. An overview of all sensed data
3. A list of traces that already have been recorded

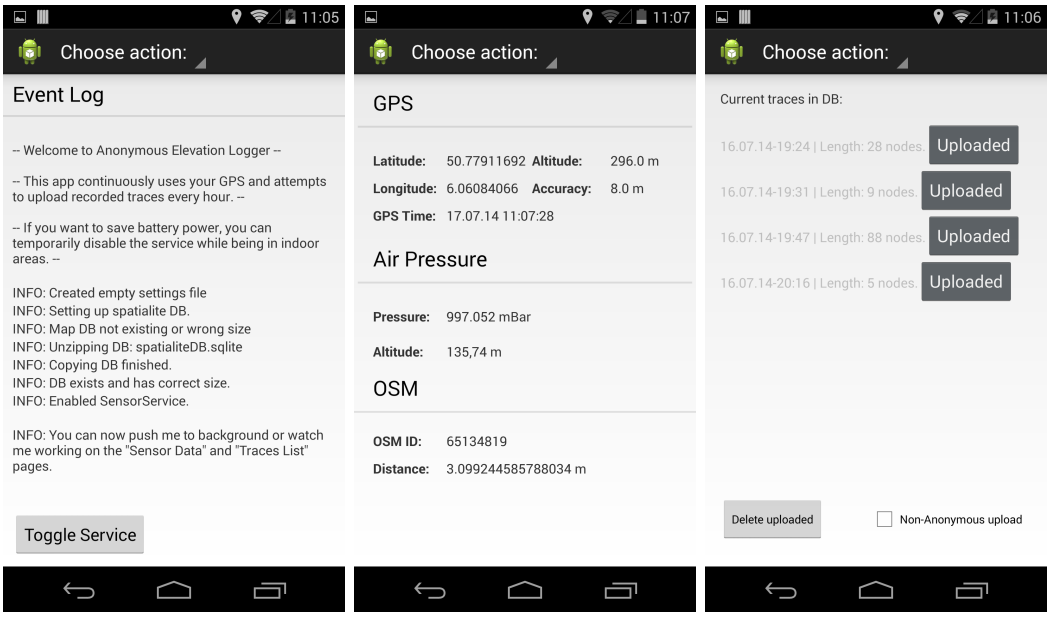

Figure 6.2 *ElevationLogger* consists of the three activities *Event Log* - showing a list of actions the application performs -, *Raw Data* - where the user can see the currently measured values - and *Traces List* - showing a list of completed traces together with their upload status.

As depicted in Figure 6.1, the user can switch between these three activities in order to read the desired information.

The *Event Log* Activity is a list of all events occurring during the application's lifecycle. Thus, it is the most important activity to inform the user about when a trace is being recorded, when a finished trace is being uploaded and when during one of these actions an error occurred. On start of the application the user gets also informed about the increase in battery drain due to the use of GPS.

The *Raw Data* Activity firstly gives an overview over the measured values. These include *latitude*, *longitude*, *altitude* and *accuracy* from the GPS module which are the raw values as received from the Android API. Further, we show the air pressure in *millibar* (equals hPa) and the altitude yielded by the barometric altitude formula for this air pressure. Note that this value is affected by weather effects and hence does not show an exact altitude. It is internally used to calculate the altitude differences between the nodes.

The *Trace List* Activity lists all successfully recorded traces by the date their recording started, their final amount of nodes and their upload status. This activity also allows for uploading traces whose upload process has previously failed. We further implement a function to perform an *non-anonymous* upload. This function serves the purpose of a faster contribution of data as released traces are directly sent to the CP. Hence, we give an opt-in option for users that want to skip the anonymization process of their traces in favor of a faster collection of altitude information. This option is reset if the application gets restarted. Otherwise, the user needs to actively opt-out if she wants further traces to get anonymized again.

6.1.2 Application Backend

To provide the functionality behind the activities, we implement a class *SensorService* as the application's backend. The SensorService runs in background and is responsible for initialization, reading and proper shutdown of the GPS module and air pressure sensor. Moreover, for each location reading we directly use a spatialite database query to efficiently find the closest OSM node. All in all, the SensorService has the following main responsibilities on service startup:

- Initialize the wake lock in order to keep the CPU running for a proper air pressure reading while the smartphone is in stand-by mode.
- Check the existence of the *traces.db* file that contains a serialized instance of the TraceDB class we implemented to manage recorded traces. If the file exists, it gets deserialized and we add new traces to the existing database.
- Initialize the Android LocationManager to use GPS as the location provider and yield new positions on a regular basis.
- Initialize the air pressure sensor such that we get a new sensor reading every 500 milliseconds.
- Initialize the Spatialite database containing the network of OSM nodes.

Using the smartphone sensors requires CPU time but unfortunately, Android prevents applications from using the CPU if the smartphone is in stand-by mode. As the smartphone rests most of the time in this mode while users travel, we must ensure that the applications get access to the CPU by setting a wake lock. With the class *PowerManager* Android provides access to the system power management where developers can set a *partial* or a *full* wake lock where the first keeps the CPU running if in stand-by and the latter also keeps the display activated. Hence, we implement a partial wake lock to keep the sensor readings working.

To manage and handle recorded traces we implement a simple datastructure called *TraceDB*. The TraceDB consists of a list of *LocationTraces* which themselves hold a list of *LocationNodes*. The TraceDB is a *serializable* Java object and hence can be saved to the smartphone storage if the SensorService is stopped, and can be deserialized if it is started. Moreover, the TraceDB provides functionality to add new sensor readings to the currently active trace and to export traces as GPX files for a later evaluation.

In order to get a recent GPS location the Android *LocationManager* needs to be set to use the GPS module as the location provider. To process the new location the SensorService extends the class *LocationListener* that provides the callback function *onLocationChanged(location)* which is called as soon as a new GPS location is available. Using the GPS provider we achieve a location update rate of approximately one location per second.

The air pressure sensor is accessed using Android's *SensorManager*. We use the SensorManager to firstly check for the existence of an air pressure sensor and secondly for initializing the sensor to yield new readings every 500 milliseconds such that a

```
SELECT osm_id, ST_Distance(geometry, MakePoint(6.4123, 50.4123), 0)
    AS distance
FROM 'regbez-koeln-highways_nodes'
WHERE ROWID IN
    (SELECT ROWID FROM SpatialIndex
        WHERE f_table_name='regbez-koeln-highways_nodes'
        AND search_frame=BuildCircleMbr(6.4123, 50.4123, 0.001))
AND distance < 30 ORDER BY distance LIMIT 5;
```

Listing 6.1 A spatialite query to yield a list of OSM nodes being nearer than 30 meters to the given GPS coordinates sorted by ascending distance.

recent reading is available for each new GPS location. Similar to the location, the sensor reading is received by a callback function *onSensorChanged(SensorEvent)* which is provided by the Android *SensorEventListener*.

While receiving new GPS locations we are also interested in the nearest OSM node for each of the GPS readings. Hence, the application contains a Spatialite[4] database of all OSM nodes dedicated to paths and streets. To keep the application at a handy size we restrict this database to the *Regierungsbezirk Koeln* in *Nordrhein-Westfahlen*, Germany. The uncompressed OSM network of streets and paths of this area requires approximately 115MB storage on the device. Spatialite supports us with the advantage that we can generate a *SpatialIndex* of all OSM nodes and relation. The SpatialIndex is a R-Tree datastructure that allows for very efficient searches for specific regions [3]. For instance, we can efficiently search for all OSM nodes in a given radius around some given GPS coordinates. We show an exemplary query for such a search in Listing 6.1. The result of this query is a list of OSM nodes around the given GPS coordinates sorted by ascending distance. We take the nearest OSM node and save it together with the GPS location and air pressure as next entry of our currently recorded trace. Finally, all traces that have not been uploaded yet, are uploaded on an hourly basis. We summarize the SensorService behavior as a flowchart in Figure 6.3.

6.1.3 Communication between GUI and Backend

In order to show the user which data is currently measured and recorded we need to establish an event-based connection between the SensorService and the activities. Such a connection serves the purpose of delivering sensor readings or information that is not directly accessible from the activity classes to the current activity in the moment they occur. We can thus directly show these information in the application GUI. The key information transported between the SensorService and the activities are:

- Current sensor readings
- A list of previously recorded traces

[4]https://www.gaia-gis.it/fossil/libspatialite/index

```
double sensorReading = 42.0;

Intent intentA = new Intent("sensor-data");
intentA.putExtra("sensorsAnswer", sensorReading);

LocalBroadcastManager.getInstance(getBaseContext()).sendBroadcast(intentA);
```

Listing 6.2 Using Intents and the LocalBroadcastManager to broadcast data within an application.

```
BroadcastReceiver recv = new SensorDataReceiver();

onCreate() {
...
LocalBroadcastManager.getInstance(this).registerReceiver(recv);
...
}

class SensorDataReceiver extends BroadcastReceiver {
    onReceive(Context con, Intent intentA) {
        double sensorReading = intentA.getDoubleExtra("sensorsAnswer");
        ...
    }
}
```

Listing 6.3 Using Intents and the LocalBroadcastManager to receive broadcasted data within an application.

Both of these information are only accessible by the SensorService class so we need to be able to submit them to the GUI. Fortunately, Android provides tools empowering us to send *Intents* via the *LocalBroadcastManager* within our application. *Intents* are empty frames that can be filled with information to be delivered or with actions to be performed. When sending data within our application we create an Intent *intentA* giving it a name e.g. *sensor-data* and fill the data into it by using `intentA.putExtra("varname", variable)`. The next step is to broadcast *intentA* using the *LocalBroadcastManager* class. The complete sending process is shown in Listing 6.2.

For the activity, or any other arbitrary class, to be able to receive the information from *intentA*, this class needs to implement a subclass that extends the Android provided *BroadcastReceiver*. This way, the subclass implements the `onReceive(Context,Intent)` method. On activity start we register our BroadcastReceiver with the LocalBroadcastManager with an IntentFilter that must have the same name as the Intents we want to receive. As soon as an Intent with this name is sent within the application, our `onReceive(Context,Intent)` is invoked and handles the received data. An exemplary receiving process is shown in Listing 6.3.

6.1.4 SMPC and Communication

Our Android application *ElevationLogger* serves as an input peer in a Secure Multi Party Computation system that anonymizes the recorded traces and finally releases them to a potentially untrusted data collector. As shown in the previous subsections

we collect recorded traces in a local trace database. On an hourly basis we check if new traces have been recorded and then attempt to create shares from these traces as explained in Chapter 5 in order to distribute the shares among the privacy peers.

Creating shares from traces is computationally expensive and thus should be done in background. Otherwise, the GUI would freeze for the time of the computations. The duration of computation depends on the trace length, the chosen bloomfilter size, the key size for Paillier's threshold encryption and of course the processor speed. On average, we expect a duration around 5 seconds. Hence, we implement an *AsyncTask* class called *UploadTrace*. Running UploadTrace's `run(traceID)` method is always performed in background using the smartphones multithreading capabilities. When invoking an instance of UploadTrace it gets supplied with the ID of the trace that should be shared and uploaded. The responsibilities of our UploadTrace class are:

- Get a *minimal* copy of the trace determined by the supplied trace ID
- Create a list of $(n_i, d_{i,i+1}, n_{i+1})$ tuples from the trace
- Receive the public key used for Paillier's threshold encryption
- Encrypt the created tuples with the public key
- Create bloomfilters for each node in the trace, plus one bloomfilter containing all nodes
- Create Shamir shares of the bloomfilters
- Release all shares together with a random trace ID among the privacy peers

The traces as stored in the applications trace database contain more information than just those that we want to release to the privacy peers. They contain all recorded GPS locations and air pressure values. Moreover, they have GPS and pressure information for locations between the real OSM nodes. Hence, we create a list of locations that contains only those which are the nearest to the next OSM node. This way, we get to know which air pressure measurements have been taken nearest to which OSM node and can attach them. The result is a list of OSM nodes with their respective air pressure measurements. We call this list a *minimal* trace.

As shown in Chapter 5 we need to process the air pressure values in order to create a list of tuples stating the altitude difference between consecutive OSM nodes. Therefor, we use the barometric altitude formula as described in Chapter 2. Altitude differences are usually not *integer* but *double* values which cannot be handled by PTE as it is based on the discrete logarithm problem where encryption and decryption is only applicable to positive integers. Hence, we scale them by a factor of 10000 in order to maintain four decimal places which is enough regarding the Nexus 5 barometer accuracy of 0.12mBar which translates to roughly 90cm. We further have to take into consideration that altitude differences can be negative in value. However, PTE does not support encrypting negative values. Hence, we circumvent this issue by parting the positive range of `0 ... MAX_LONG` into the range `0 ... MAX_LONG/2` for positive values and `(MAX_LONG/2)+1 ... MAX_LONG` for

negative values. For instance, we map the value -1 to 4611686018427387904 with $MAX\_LONG = 9223372036854775807$.

In order to create the PTE shares of our altitude difference tuples we use the PTE toolbox[5]. The toolbox is a PTE Java library, hence, it is suitable for use in Android applications. It provides simple means to handle the required public key with the *PaillierKey* class. Once the public key is received from the key dealer, we can initialize an *PaillierKey* instance and use it to create an instance of *PaillierThreshold*. The latter is used to encrypt data using the public key. This way, we encrypt every node and every altitude difference using PTE. The resulting encrypted data is of class *BigInteger* which is used to represent integers larger than supported by native types. Unfortunately, the MsgPack library used to send messages over the network does not support numbers of this size. Thus, we convert the numbers into strings before sending. We consider the optimization of sending numbers over the network with less overhead as future work.

For the creation of BFs we implement an own class *MMH3BloomFilter* that allows for adding elements to a BF and looking up elements if contained in a BF. A BF is an array of bits where adding an element to the BF sets those bits to 1 that are at the position `hash(seed, elem) mod m` where m is the BF length. Each added element is hashed h_{count} times with different seeds such that after adding one element at most h_{count} bits are set to 1. As the hash function we use a public domain implementation of MurmurHash3[6]. Finally, we create one BF for each node in our trace plus one BF containing all nodes. Adding and looking up elements is done as shown in Listing 6.4.

```
1  int bfSize = 40;   // Bloomfilter will have 40 bits
2  int bfHashes = 7;  // Elements will be hashes with 7 different seeds
3                     // setting at most 7 different bits in the BF
4
5  String elem = "Add me!";
6
7  MMH3BloomFilter bf1 = new BloomFilter(bfSize,bfHashes);
8  bf1.add(elem);
9
10 boolean contains;
11 contains = bf1.lookup("Add me!"); // True!!
12 contains = bf1.lookup("42");      // False!!
```

Listing 6.4 Adding and looking up an element to and from a BF.

Creating the Shamir shares of the BFs is done using the SEPIA library as presented in [7]. SEPIA facilitates the simple creation of shares from integer inputs. Settings that need to be set before are the *fieldsize* i.e., the residue class ring used in Shamir's calculations, the degree t of polynomials, and the number of privacy peers. Creating Shamir shares for an array of values is shown in 6.5. We do this for all BFs of the current trace.

At this point we created PTE shares for the altitude difference tuples and Shamir shares for the BFs. These shares are ready to be sent to the privacy peers. Using

[5]https://www.utdallas.edu/ mxk093120/paillier/
[6]http://github.com/yonik/java_util

```
long[] inputs = {0,1,0,0,1,1,0,0,1,0}; // BF of size 10
int degree = 1;    // Requires 2 PPs for reconstruction
int peers  = 3;    // Number of supporting points of the polynomial
long fieldsize = 92233720368547757831; // Largest prime
                                       // smaller than MAX_LONG

ShamirSharing shamir = new ShamirSharing(peers);
shamir.init();                         // Initialize sharing matrix
shamir.setRandomAlgorithm("SHA1PRNG");
shamir.setFieldSize(fieldSize);
shamir.setDegreeT(deg);

long[][] res = shamir.generateShares(inputs);
```

Listing 6.5 Creating Shamir shares from a given array of inputs.

MessagePack we can serialize the shares in the Android application to a byte-stream ready to be sent over the network. The counterpart at the privacy peers then deserializes the byte-stream to a Python object. A Java *HashMap* is thus deserialized to a Python dictionary. Finally, we use TCP *Socket* connections to release the shares to the privacy peers. As stated in Chapter 5 we need to secure the connections between all participating peers to prevent adversaries from collecting shares and ultimately from reconstructing secrets. This is considered future work but can be easily achieved by using TLS for all connections.

6.2 Privacy Peers

The PPs are responsible to anonymize the data they receive from the IPs. Hence, they should provide functionality to process shares from Shamir's Secret Sharing and from Paillier Threshold Encryption. We implemented the privacy peers in Python for a fast prototyping progress. While we use an own Python implementation of PTE, the library handling Shamir shares stems from VIFF, the Virtual Ideal Functionality Framework[7]. We depict the PP behavior in the flowchart in Figure 6.4. In this section, talking about a *trace* always refers to the shares of this trace located at the PPs.

On startup the PPs request their respective part of the PTE private key from the key dealer. In our prototype we decide which part each PP gets by their ID. However, as IDs can easily be spoofed, a productive implementation would have to use peer authentication at this place. As an alternative, the key generation for PTE could be done in distributed fashion using SMPC by the PPs. This would render a dedicated dealer unnecessary.

In order to communicate with each other, the PPs establish a TCP connection to one another which lasts until one PP is shut down or the connection breaks. As soon as all connections are established, the PPs start listening on an extra port for incoming trace shares.

In the PPs main loop we regularly check if new trace shares have been received. Assume that IP_1 releases a trace t_1 to the PPs where the shares are $[t_1]_{PP_1}, [t_1]_{PP_2}, [t_1]_{PP_3}$

[7] http://viff.dk

```
// a) --- Direct intersection of plain BFs ---
    bf1 = [0,1,1,0,0,1,1,1,0,1]
    bf2 = [1,1,0,1,0,0,1,1,0,0]
    intersectionBF = [bit1 & bit2 for bit1,bit2 in zip(bf1,bf2)]
    // intersectionBF == [0,1,0,0,0,0,1,1,0,0]

// b) --- Intersection of BF shares ---
    bf1 = [...] // List contains shares of bf1's bits
    bf2 = [...] // List contains shares of bf2's bits
    // Logical AND gets emulated by multiplication of shares
    // where * is the multiplication of Shamir shares
    intersectionBF = [bit1 * bit2 for bit1,bit2 in zip(bf1,bf2)]
```

Listing 6.6 Calculating the intersection of a) two plain BFs and b) two shared BFs

for a set of three PPs. Network delay can cause these shares to not arrive simultaneously at all PPs such that PP_1 receives $[t_1]_{PP_1}$ but PP_2 and PP_3 still have no knowledge of t_1 because they did not receive $[t_1]_{PP_2}$ and $[t_1]_{PP_3}$ yet. To solve this issue the PP that receives its shares initializes a synchronization round by requesting the IDs of the traces that the other PPs have recently received. The next processed trace is then the commonly available trace with the lowest ID. If the PPs have not yet saved any other traces the new trace is saved and the PPs determine the next trace to be processed.

If the PPs agreed on the next trace t_{new} and they have a set of known traces $t_{old_1}, \ldots, t_{old_2}$ we create the intersection BFs for each known trace i.e., we take $BF(n_1, \ldots, n_m | n \in t_{new})$ and $BF(n_1, \ldots, n_m | n \in t_{old_i})$ and calculate their intersection BF_{int} for each known trace. In Listing 6.6 we show how the intersection BF of two BFs is calculated in Python. Under SMPC the BF bits are Shamir shares hence we emulate the logical AND between two bits by the distributed multiplication of the shares as stated in Chapter 5. We can now check each BF of the single nodes in t_{new} against BF_{int} if they are contained. A node n is probably contained in both traces if $BF(n)$ is contained in BF_{int}. Refer to Chapter 2 for an explanation of the probabilistic properties of bloomfilters. Listing 6.7 shows how we check this in Python. We first intersect $BF(n)$ with BF_{int} followed by counting the 1s in the intersection. If the number of 1s equals the amount of hashes used to add node n to the BF, node n is most probably contained in both traces. However, we have to keep in mind that we can yield false-positives as stated in Chapter 2.

If the intersection check yields a positive result for a node stating it to be contained in both traces t_{new} and t_{old_i}, these traces get merged. We merge two traces by creating a new trace containing both traces' altitude difference tuples and single-node BFs and a union of their complete BFs. We create the union of two BFs by calculating the bitwise OR. Under SMPC regarding Shamir shares we emulate the bitwise OR using Equation 5.5 as stated in Chapter 5. The complete calculation is shown in Listing 6.8. Remember, the calculation is performed on Shamir shares, thus, we have to use the Shamir specific multiplication as stated in Chapter 2.

Once merged, the PPs still know which part of the trace belongs to which original trace. Thus, if sent with a non-anonymous IP address, the PPs could still identify a contributor in a merged trace. To circumvent this issue we shuffle the PTE tuples holding the altitude differences between the nodes. The shuffling process is described

```
bfHashes = 2 // Number of hashes used for adding elements to BF

// a) --- Direct check using plain BFs ---
    bfN   = [0,0,1,0,0,0,1,0,0,0] // BF of single node n
    bfINT = [1,0,1,1,0,0,1,1,0,0] // BF of all common nodes
    intersectionBF = [bit1 & bit2 for bit1,bit2 in zip(bf1,bf2)]
    // intersectionBF == [0,0,1,0,0,0,1,0,0,0]

    // If number of 1s in intersectionBF equals bfHashes
    if sum(intersectionBF) == bfHashes:
        return True // Node n is contained in both traces!

// b) --- Check using BF shares ---
    bfN   = [...] // List contains shares of bfN's bits
    bfINT = [...] // List contains shares of bfINT's bits
    // Logical AND gets emulated by multiplication of shares
    // where * is the multiplication of Shamir shares
    intersectionBF = [bit1 * bit2 for bit1,bit2 in zip(bf1,bf2)]

    // If number of 1s in intersectionBF equals bfHashes
    if reconstruct(sum(intersectionBF)) == bfHashes:
        return True // Node n is contained in both traces!
```

Listing 6.7 Checking if node n is contained in t_{new} and t_{old_i} using BFs.

```
bf1 = [...] // List contains shares of bf1's bits
bf2 = [...] // List contains shares of bf2's bits
// A or B -> A + B - A * B
unionBF = [(bit1 + bit2 - bit1 * bit2) for bit1,bit2 in zip(bf1,bf2)]
```

Listing 6.8 Building the union over two shared BFs.

in Chapter 5 and is implemented straight forward. However, shuffling incorporates a re-randomization by adding [0] to each shuffled value. For performance optimization we use the idle time between new incoming traces to pre-compute PTE encrypted zeros [0] for direct availability. Finally, the merged and shuffled trace is added to the list of known traces where the original traces are removed.

As a last step we check the list of known traces for candidates fulfilling the anonymity requirement k. For each candidate each PP decrypts the altitude difference tuples with its part of the PTE private key and releases them to the CP which is now able to reconstruct the shuffled and k-anonymous trace.

6.3 Collector Peer

The CP is implemented in Python such that it can use the same libraries for network connections and share handling as the PPs. Its responsibilities are the reception and storage of anonymized traces. It therefor listens for incoming connections from the PPs releasing traces. The CP receives partial decryptions of trace tuples from each PP. Hence, we need to secure the connections from the PPs such that an adversary can not intercept the partial decryptions. After full reception of a trace the CP reconstructs the tuples and stores them for later processing. We choose the JSON

format for storing traces as it is a universally processable and readable format. The complete CP behavior is shown in Figure 6.5. We consider the CP to only reconstruct and store the anonymized traces. Further handling for differing altitude information between the same pairs of nodes is easily implementable and together with releasing the information to OSM considered as future work.

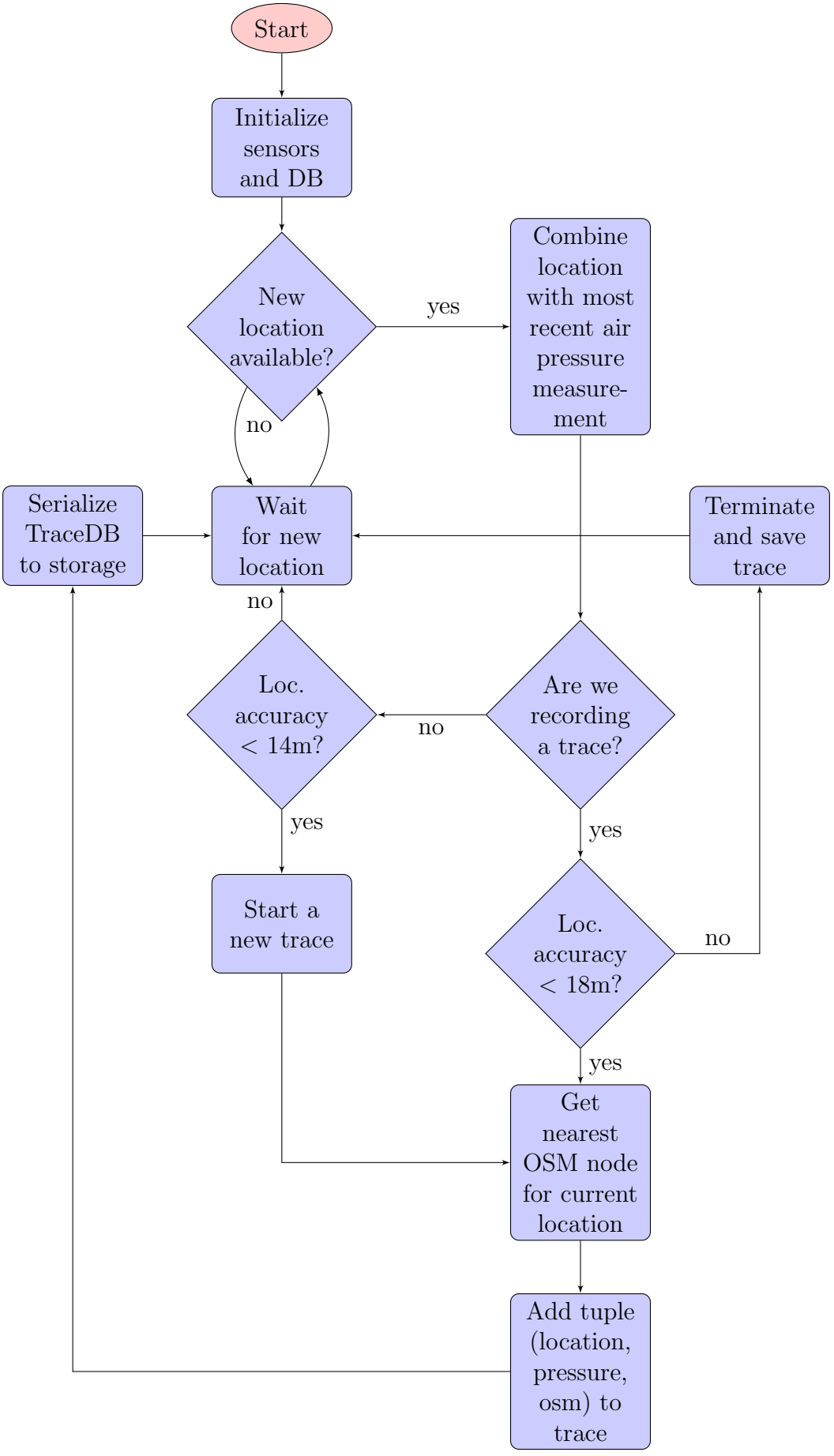

Figure 6.3 Flowchart showing how SensorService behaves and under which conditions traces are recorded.

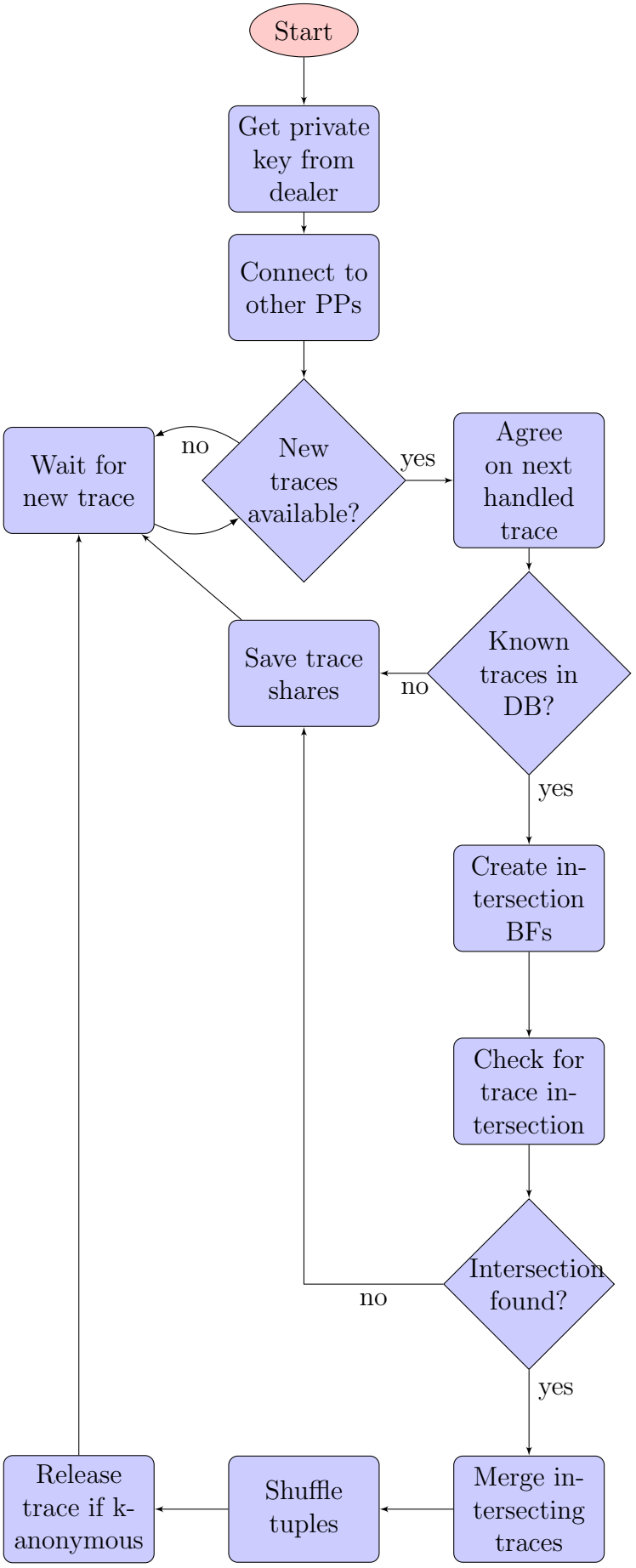

Figure 6.4 Flowchart showing the privacy peer behavior.

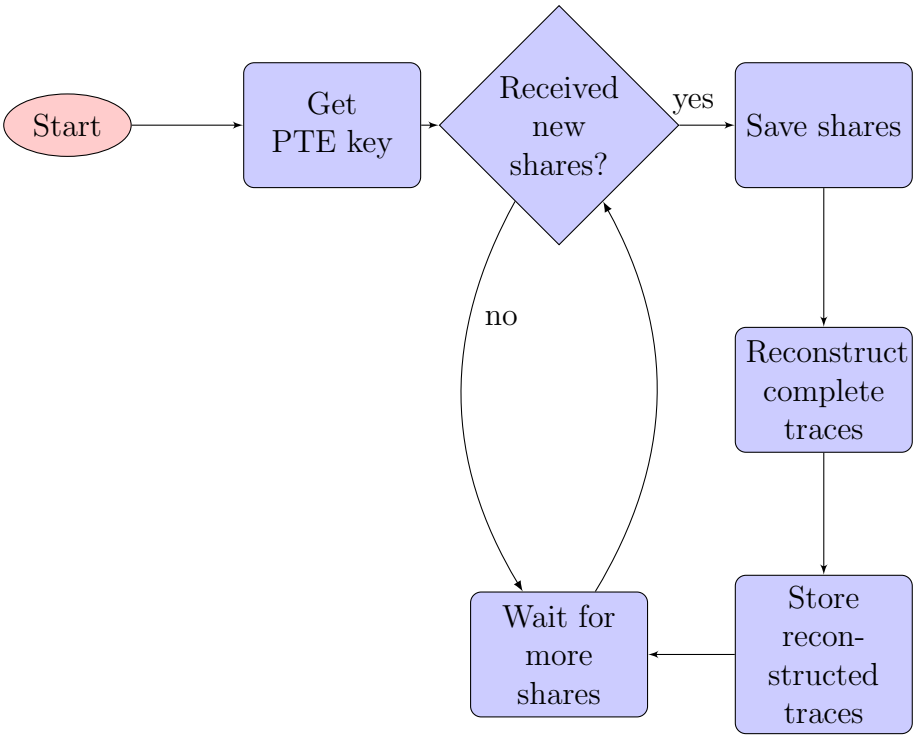

Figure 6.5 Flowchart showing the collector peer behavior.

7

Security Discussion

As stated in Chapter 4, we make demands on the user-privacy and security. We ultimately want to achieve that a contributor can neither be identified as a data source nor as subject to a recorded trace. To fulfill this demand, we presented an anonymization process based on a TTP emulated by SMPC. In this chapter, we explain the different parameters of SMPC and their influence on system security together with the chosen adversary model. Moreover, we explain user *unlinkablity* and show that our system fulfills this requirement. We further discuss home identification attacks on anonymized traces and how they can be mitigated. We finally note that our system meets the requirements on anonymity as stated in Chapter 4.

When regarding possible attacks on our system we have to choose between several adversarial models. Especially, we distinguish between the *malicious* adversary and the *honest-but-curious* adversary. Where the *malicious* adversary may actively misbehave by diverging from the system protocol, the *honest-but-curious* adversary must comply with the protocol but tries to gain as many information from the protocol transcript [17]. In our system, we consider the adversaries to be *honest-but-curious*. The processed data in our system has no monetary or life-harming influence such that we expect that an adversary would not incur the effort of actively disturbing our system.

We distinguish between two types of user reidentification as stated in our anonymity requirements in Chapter 4:

1. Reidentification by data source linkability.
2. Reidentification by data content traceability.

When a contributor releases the shares of a trace to the PPs the PPs know that these shares belong to the same trace and thus to the same contributor. Hence, we need to eliminate the link between user and shares before they get reconstructed at the CP. We do this by shuffling and re-randomizing the shares at the PPs. For instance,

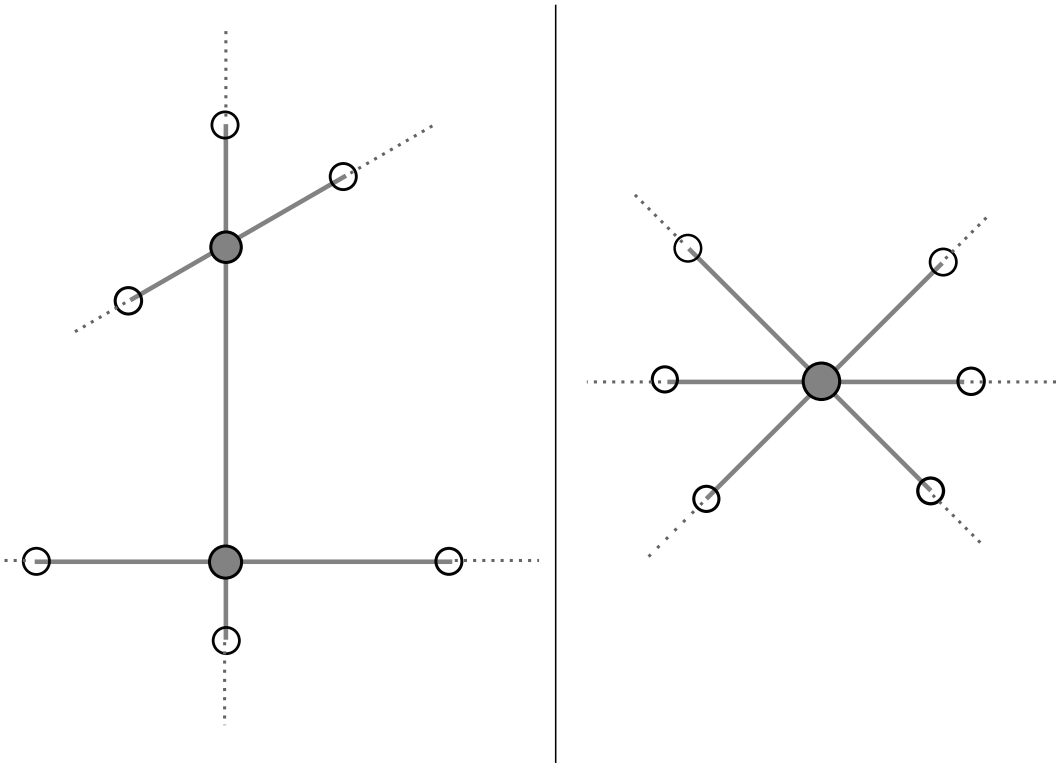

Figure 7.1 Left side: k=3 anonymous trace with *transitive intersections*. Right side: k=3 anonymous trace with *single intersection*.

directly after merging two traces t_1 and t_2 into $t_{1\cup2}$ we still know which shares belong to which original trace as $t_{1\cup2}$ essentially is a concatenation of t_1 and t_2. Thus, the anonymity level of the content increased reasoned by the intersection of t_1 and t_2 but we can still link the single shares to a source. We eliminate this link by shuffling the order and re-randomizing by adding an encrypted zero to the PTE shares at each PP. A visualization is shown in Figure 5.3. Re-randomization prevents the next PP in the round from recognizing the shares and possibly reordering the shares. After one complete round none of the PPs can link the shares to a source anymore.

The CP receives all its messages from the PPs and hence can not link the traces to the original source. However, the CP eventually reconstructs the content of the trace and is assumed to draw conclusions about the persons who created the trace. These conclusions hence are solely based on the content of the trace. We prevent the application of home identification, data matching [16] or record linkage attacks [12] by merging traces at the PPs until they fulfill an anonymity level k. We call a trace k-anonymous if one contributor of the trace is indistinguishable from k-1 other contributors of the same trace. Indistinguishability is given if multiple traces intersect and we are not able to tell where the contributors entered and left the intersections. The intersections created by our system are categorized in *transitive intersections* and *single intersections*. As depicted in Figure 7.1, transitive intersections let multiple traces intersect in different points where single intersections are intersections of traces in the same point. Undoubtedly, choosing a higher k improves the contributor anonymity but it is also a trade-off to data collection speed as traces that do not reach k-anonymity will never be released to the CP.

Although exhibiting k-anonymity, at least parts of traces are attackable by home identification. This first or last part of a trace before and after an intersection is a potential candidate for home identification. Though we can not draw conclusions about the further course after the intersection, we can possibly identify the person departing or approaching her home. To circumvent the identification at trace endpoints we suggest the use of geofencing when recording traces. Using geofencing the user is able to white- or blacklist areas such as her home and vicinity for trace recording ultimately preventing home identification attacks. However, the implementation is considered future work.

Our system uses SMPC to emulate a TTP for anonymity establishing calculations. The TTP can consist of an arbitrary number of PPs where the PP amount is a trade-off between computational effort and security. Most importantly, a higher amount of PPs makes it harder for an adversary to gain control over a majority of PPs which would empower him to jam the system or reconstruct non-anonymous traces. On the other side computational effort to perform calculations under SMPC increases with rising number of PPs. For a detailed evaluation of this point we refer to Chapter 8.

Both, using PTE and Shamir's secret sharing involves the decision for a suitable threshold value $share_{thres}$. This threshold determines for PTE how many parts of the private key are needed in order to decrypt the secret. As each PP holds one part of the private key, the threshold also determines how many PPs have to collaborate to decrypt the secret. Similarly, we set the threshold $share_{thres}$ when creating Shamir shares such that these shares are based on polynomials of degree $share_{thres} - 1$. Hence, we need $share_{thres}$ collaborating PPs to reconstruct a Shamir share. An adversary would have to gain control over at least $share_{thres}$ PPs which then are able to reconstruct the shared PTE or Shamir secret. At this point, the system operator must decide the trade-off between security (more PPs) and computational effort (less PPs). We show how computational effort behaves with rising number of PPs in Chapter 8.

Altitude difference tuples are shares using PTE where PTE allows us to choose between different key sizes. Like Diffie-Hellman, PTE bases on the *discrete logarithm* problem [19]. As for other public key algorithms like RSA key sizes of 1024 bits may suffice for the near future but are likely to get insufficient [23]. Hence, we recommend the use of primes of length 2048 bits or more for PTE.

Over the complete process of data collection we can attack the contributor identity through data source linkability and data content traceability. We have shown that using shuffling and PTE for trace tuples eliminates a linkability to the data source successfully. Moreover, we have shown that k-anonymity is a suitable measure to circumvent contributor traceability from the data content.

8

Evaluation

In this chapter we firstly evaluate the accuracy of collected barometric altitude differences in comparison to a reference map and secondly we evaluate the performance of our SMPC based trace anonymization system.

8.1 Barometric Altitude Differences

All self recorded measurements in this section are captured using a Nexus 5 smartphone containing the BMP280 [4] piezo-based air pressure sensor. According to its datasheet it features a typical relative pressure accuracy of $0.12mBar$ while the data resolution is $0.01mBar$. However, in our evaluation we will see that accuracy is usually better in comparison to $0.12mBar$.

8.1.1 Feasibility

In Chapter 2 we stated that barometric altitude is subject to weather effects like high- and low-pressure areas. We begin this section by showing that these weather effects have no significant influence on our measurements as the speed of weather based air pressure changes is not high enough to influence the measurements between neighboring OSM nodes.

To show how significant weather impacts our system, we consider the average distance to neighboring nodes in the OSM network of *Regierungsbezirk Köln*, Germany, and an average traveling speed for pedestrians to calculate the average time needed to travel from one node to the next. Then, we show that air pressure changes within this time frame have a magnitude that is below the accuracy of the barometer used in our test device. To be exact, the average distance to neighboring nodes in the street and path network used in this thesis is $\approx 125m$. Where - given an average walking speed of 4km/h - a user needs on average $112,5s$ from one node to the

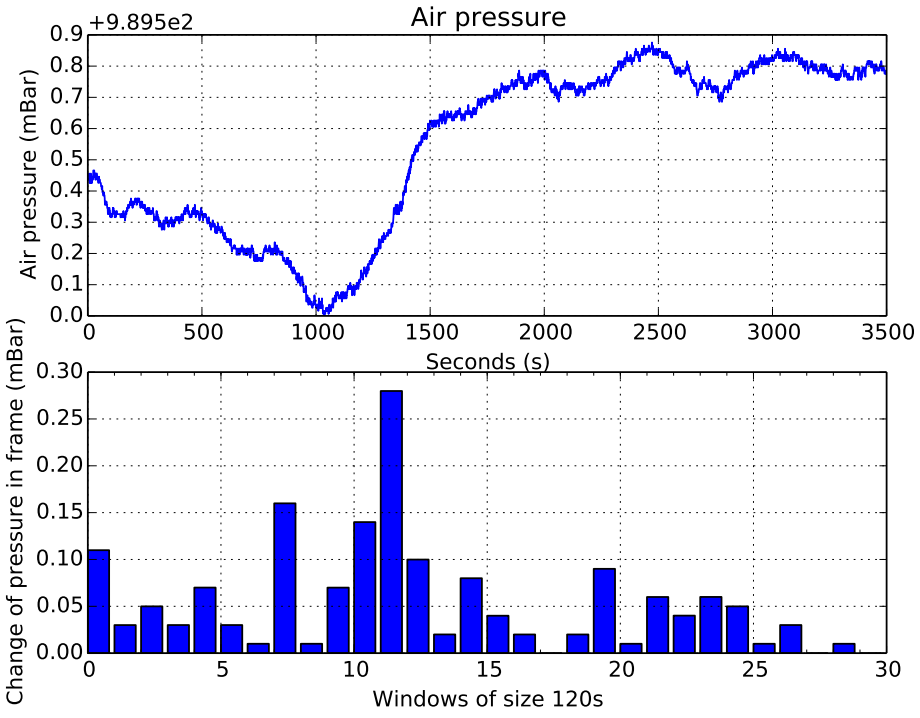

Figure 8.1 On the top we see the real air pressure values measured within the hour of fastest change on that day. On the bottom we depict the change of air pressure for two-minute frames from that hour i.e., the pressure difference between the first and the last measurement in that frame.

next. Hence, we are interested in how much weather affects the air pressure in time frames around $112,5s$. However, we have to note that neighboring nodes with higher distance such as highway nodes are usually passed with higher speed. Hence, the practical time between OSM nodes is probably lower than $112,5s$.

Figure 8.1 shows an one hour extract from a twelve hour measurement of air pressure changes on a fixed location. This particular extract was chosen because it contains the fastest change in air pressure within the twelve hour measurement including the rise of a bad weather front. We can read from the graph that during the fastest change within the twelve hour measurement the pressure changes by $0.28mbar$ within that timeframe where the used barometer features a typical relative accuracy of $0.12mBar$ according to [4]. Further, we note an average change over all time frames of $0.06mBar$ i.e., the air pressure difference between the first and the last measurement in a two minute frame is on average $0.06mBar$ in this case. Hence, we assume that meteorological influences do not have a relevant influence while moving between OSM nodes.

8.1.2 Accuracy

The product of our data collection system is a set of air pressure based altitude differences between neighboring OSM nodes. In this section we compare these traces with a reference map of the region of Aachen, Germany. The reference map is provided by the local land-registry and features an average grid resolution of 20cm by 20cm and an altitude resolution of 10cm. The reference data was collected using LiDAR technology i.e., by laser scanning the ground from an air plane. We have two

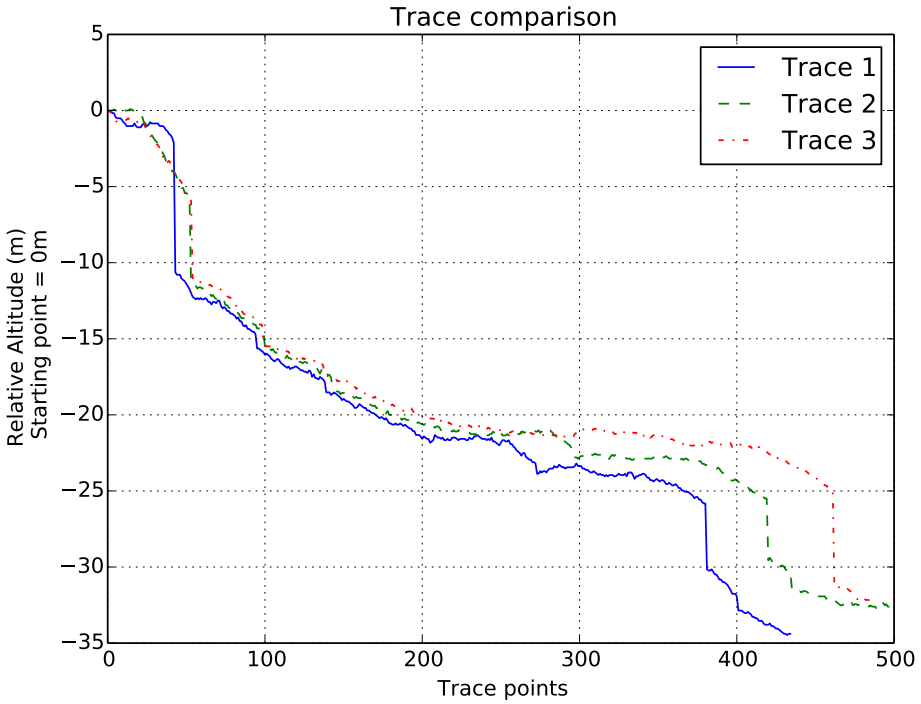

Figure 8.2 This figure shows three traces of the same path recorded on different days with different air pressure. Trace points directly convert to seconds.

variants of this map where the first has the mentioned exact resolution but contains obstacles like trees and rooftops, and the second is a 25m by 25m grid of the same data but cleaned from obstacles.

8.1.2.1 Self Recorded Traces Evaluation

We start the evaluation with manually recorded traces as these contain exact GPS coordinates in contrast to system-collected traces which only feature OSM nodes. Exact GPS coordinates feature a better position accuracy as the mapping to OSM nodes is omitted. Later, we evaluate the anonymized traces collected through our system over a time of approximately one month. In the first two weeks of collection we registered 10 active installations of our application.

Recording the same trace in different weather environments can yield slightly different results for altitude differences. Using our manual recordings we show how strong multiple measurements of the same trace differ. We consider three different traces of the same path recorded on different days with different air pressures and show how their air pressure values and altitude differences variate in Figure 8.2.

At the end of the traces we can see a deferral caused by different travel speed. The common slopes around trace points 50 and 400 were caused by a gap in the GPS recordings. We see that the general characteristics of the paths are identical besides some minor deviations.

Figure 8.3 depicts an exemplary comparison between a barometric elevation profile, the reference profile and geoid corrected GPS profile for a single trace. Looking at this Figure, we have to note the following:

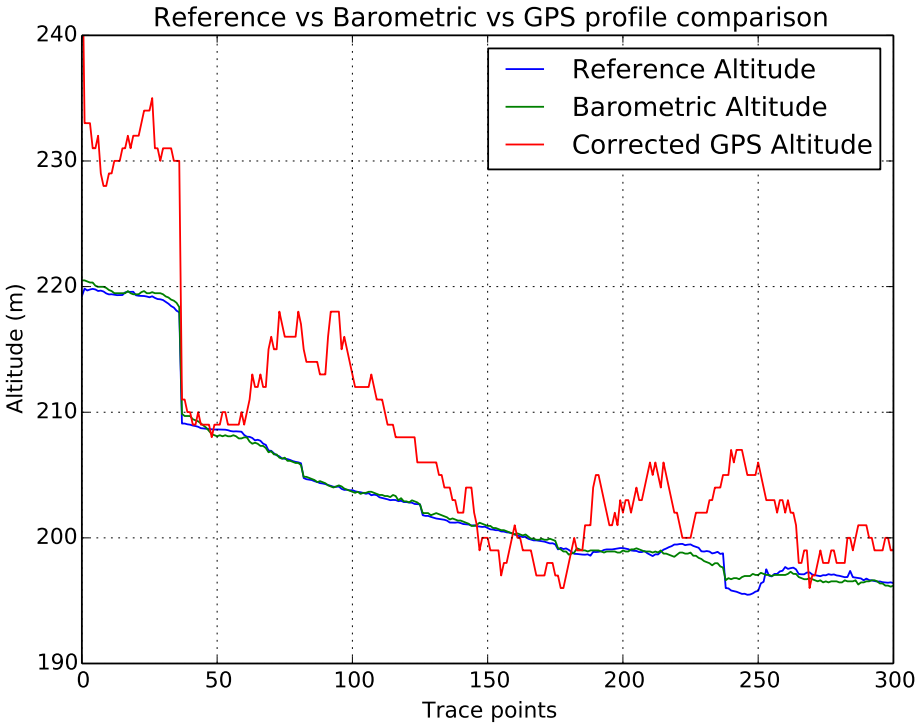

Figure 8.3 In this figure we show the reference elevation profile and our barometric profile of the same trace. As our barometric trace only features altitude differences, we aligned it to the reference trace such that the error between them is minimal. This figure visualizes that we yield very accurate altitude differences using our system for this trace.

1. Our barometric altitude only consists of relative altitude differences, hence, in Figure 8.3 we align it to the reference profile to yield a minimal error. This way, we can directly identify the relative deviations from the reference.

2. The GPS altitude is geoid corrected but is also very erratic. In this case, the mean error in comparison to the reference altitude is 5.05m and the maximal error is 30.78m. If we compare the altitude differences of consecutive trace points (first derivation) between GPS and reference profile, GPS deviates on average *0.81m*.

3. Because our barometric trace is manually aligned to the reference profile we cannot compare the absolute altitudes. Regarding altitude differences of consecutive trace points (first derivation) between the barometric and reference profile, the barometric profile deviates on average *0.12m*.

To finish the evaluation of manually recorded traces, we compare barometer and GPS based altitude differences of five traces with the reference differences in Figure 8.4. We have to remember that we compare the accuracy of altitude differences between consecutive trace points, not the absolute altitude. This is also the reason why GPS performs comparably well. Further, we see that the results are very similar if compared against the exact reference map (left diagram) or the obstacle cleaned reference map (right diagram). We finally see, that the barometer based altitude differences of consecutive trace points have an average error of *0.19m* compared to the exact reference map, and *0.12m* compared to the obstacle cleaned reference map. GPS performs comparably bad with results of *0.88m* and *0.80m*.

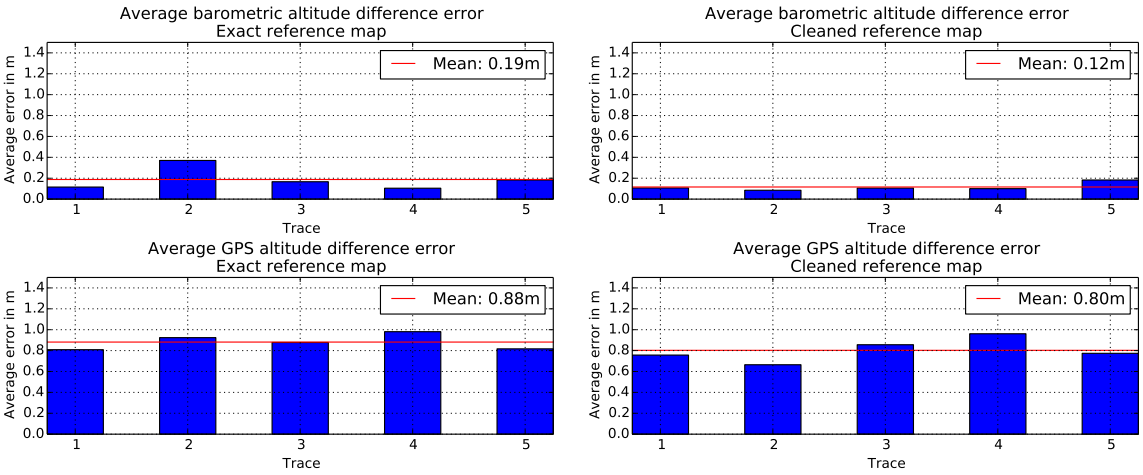

Figure 8.4 This Figure compares five traces with the exact reference map (left side) and the obstacle cleaned reference map (right side). For each diagram the upper part shows the average error of altitude differences between the barometric elevation profile and the reference profile, while the lower part shows the same for the GPS based elevation profile. The red lines depict the average error over all traces.

The previous evaluations have all been done by comparing to the exact reference map of the land-registry. However, this map represents an elevation profile that contains also trees, buildings and other obstacles. Hence, this map is no representation of the pure ground elevation. Unfortunately, this causes errors in the evaluations if the evaluated traces pass these obstacles. For instance, inaccurate GPS measurements can lead a trace over a building even though the user actually walks along the street next to the building. This case also applies to our manually recorded traces. At certain points the GPS trace overlays a building or tree such that the reference altitude at this point is the altitude of the obstacles top. Clearly, we should not compare this reference altitude to the one we measured by air pressure or GPS. Ultimately, we masked our self-recorded traces such that these points are ignored for the evaluation. As for the evaluation of all anonymously collected traces we cannot mask them due to the following reasons:

1. The anonymous traces have considerably less trace points i.e., only the OSM nodes.

2. The bulk of traces makes the effort of masking infeasible.

Thus, the following evaluations of our anonymously collected traces is entirely based on the obstacle cleaned reference map with a resolution of 25m in the plane and 0.1m in altitude. We can further see from Figure 8.4 that the results from both references are comparable.

8.1.2.2 Collected Traces Evaluation

We start our trace evaluations with on overview over all collected traces. Figure 8.5 shows all traces that have been collected by our anonymization system over the time of one month. We collected these traces from voluntary users of our Android

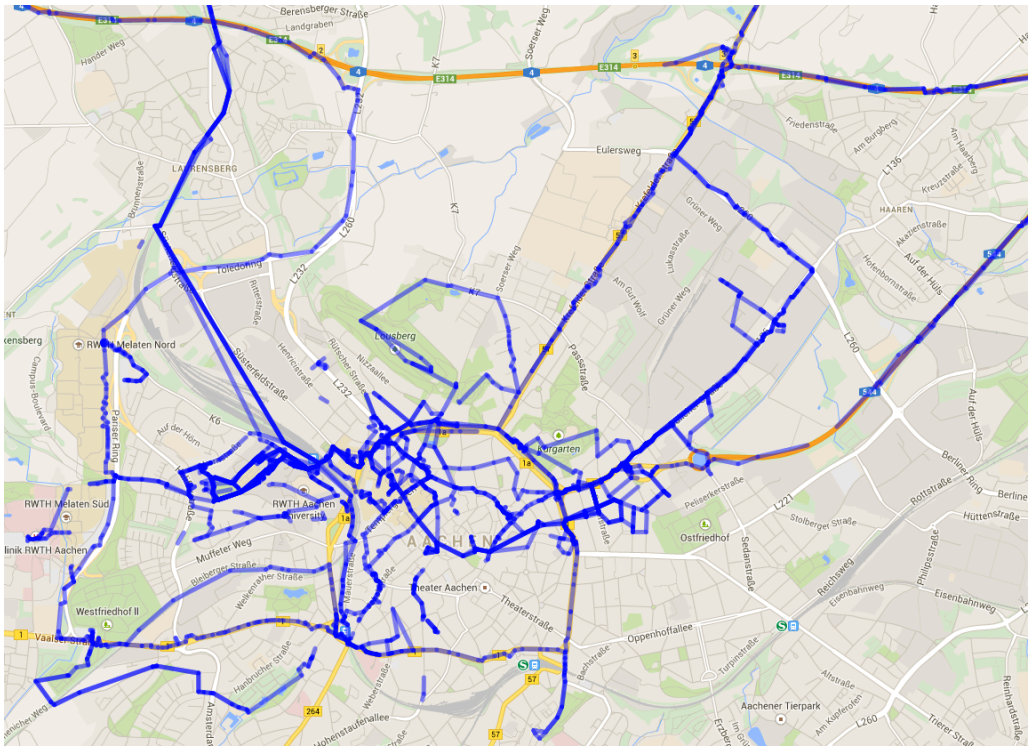

Figure 8.5 This map shows all traces that have been collected by our anonymization system over the time of one month.

application. We can consider this map as a directed graph where each edge holds the altitude difference between its source and destination vertex. As done in the GPS trace evaluations, we will compare the collected altitude differences with those calculated from the obstacle cleaned reference elevation profile used previously.

All in all, we have a total amount of 95 valid traces out of 135 that we evaluate. The remaining traces were either recorded using the first version of our Android application that did not feature the wake lock that allows for proper air pressure measurements when the application works in background and the screen is locked, or they have not been recorded in the area of Aachen, Germany. In the latter case, we have no reference material to compare with. Note, each of the 95 traces contains two original traces as they have been collected with an anonymity requirement of k=*2*. Table 8.1 shows a summary of our evaluation results. The evaluation process for each trace is structured as follows:

1. Lookup the GPS coordinates for each occurring OSM node in the spatialite database.
2. Lookup the absolute altitudes of the GPS coordinates in the reference map.
3. Calculate the reference altitude differences for neighboring nodes.
4. Compare the reference altitude differences with those from the trace.

In Figure 8.6 we show the average altitude difference error for each of the 95 traces sorted by ascending error. We see that 25 percent of traces have an average error below *0.70m* and 75 percent are still below *1.19m*. All in all, on the error averages

No. of Traces	Error	Std. Dev.	Min. Error	Max. Error
95	0.99m	0.91m	0.09m	3.58m

Table 8.1 Summary of evaluation results for altitude differences from anonymized traces. All values are the averages over all 95 traces.

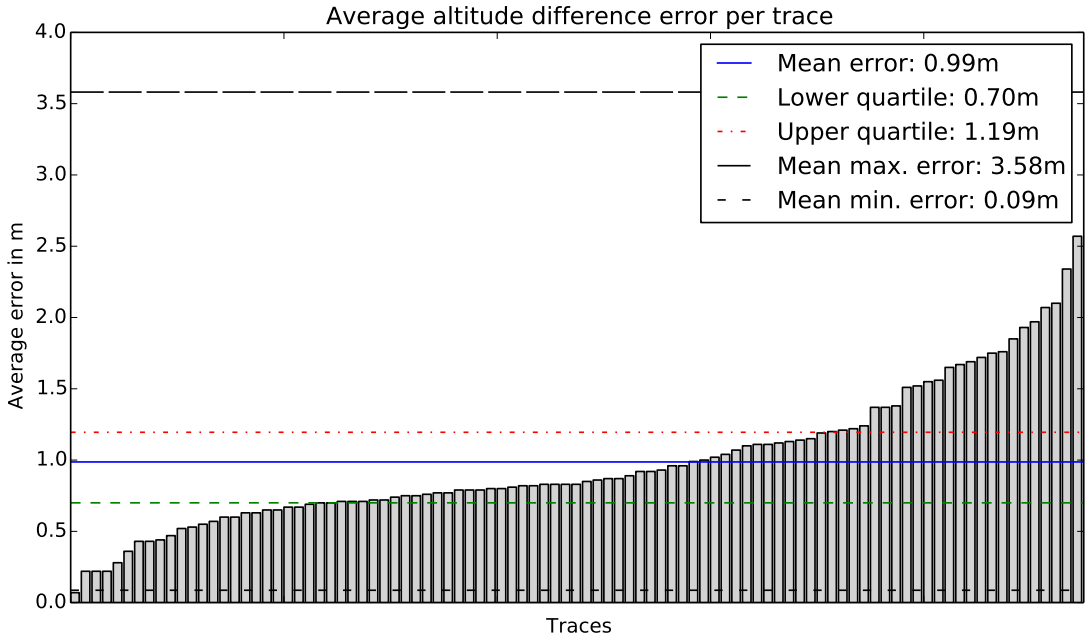

Figure 8.6 This figure gives an overview over the average altitude difference error for each trace. We additionally see, the average minimal and maximal error over all traces. Moreover, 25 percent of the traces have an average error below *0.70m* while 75 percent of the traces have an average error below *1.19m*. The error averages to *0.99m*.

to *0.99m* for all traces. We also calculate the minimal and maximal error for each trace. On average these account for *0.09m* and *3.58m*. The collected traces have lengths between 2 and 283 OSM nodes with an average length of 42 nodes.

Compared with our manually recorded traces, the average error is significantly higher. The reason is found in the higher distance between trace points. Where our GPS traces have a trace point for each second, we stated in 2 that in the OSM path and street network a pedestrian needs on average *112,5s* from one node to the next. Hence, on average *112,5s* pass while a user moves between two consecutive nodes in the anonymized traces. As our anonymized traces do not contain GPS elevations, we cannot compare our barometric altitude against those from GPS for the 95 traces. However, we have to note that in our self recorded traces GPS performed approximately 5-8 times worse in comparison.

8.2 SMPC Based Trace Anonymization

The traces recorded by our contributors pass an anonymization process which is realized under SMPC as explained in Chapter 5. The performance of this process depends on the following variables:

	EU	East US	West US
EU	(1.202ms , 0.252ms)	(76.266ms , 0.177ms)	(170.091ms , 5.009ms)
East US	(76.303ms , 0.301ms)	(1.146ms , 0.173ms)	(59.967ms , 0.364ms)
West US	(169.595ms , 5.952ms)	(60.006ms , 0.406ms)	(1.089ms , 0.254ms)

Table 8.2 Ping-test between Amazon EC2 instances located in Ireland (EU), North Virginia (East US) and Oregon (West US). The tuples represent *mean* and *standard deviation* over 30 echo requests. Column title is the source and row title is the destination.

1. The amount of privacy peers
2. The Paillier Threshold Encryption key size
3. The bloomfilter size
4. The length of the traces

We evaluate the system's performance by varying one of the variables while keeping the others fixed. Thus, we get an overview of each variables impact on the system performance. Afterwards, we state which variable values give a practical starting point for a productive system usage. Using these values we show how many traces the system can handle per minute, how much communication overhead is caused, and how much time is required for the input peers to create the shares.

8.2.1 Evaluation Environment

For our evaluations the privacy peers are set up on the smallest Amazon EC2 instances featuring one virtual CPU and 1GB of system memory. The host systems all feature *Xeon E5-2670 v2 @ 2.50GHz* CPUs. To give a realistic network environment with differently behaving connections between the privacy peers, the privacy peers are distributed over three locations worldwide i.e., Ireland (Europe), North Virginia (East US) and Oregon (West US). The PTE key dealer and collector peer are run together on a machine at ComSys in Aachen, Germany. As stated in Chapter 7 these roles must be separated in a productive rollout, otherwise the untrusted trace collector is in possession of the complete Paillier Threshold Encryption key and hence is able to reconstruct single not-anonymized traces if in collaboration with one of the privacy peers. Finally, the evaluation of required processing time on input peers for creating shares is performed on a late 2013 MacBook Pro with 2.4Ghz Dual-Core Intel Core i5 processor and 8GB system memory.

We start by giving an overview over the network connectivity between the three locations Ireland, North Virginia and Oregon, in Table 8.2. The table clearly states that the connection between EU and West US is the most prone to delays. It has a round trip time around *170ms* in both directions and a standard deviation of *5-6ms*. The remaining connections between different locations show a delay between *60-76ms* with constantly low standard deviation around *0.2-0.4ms*.

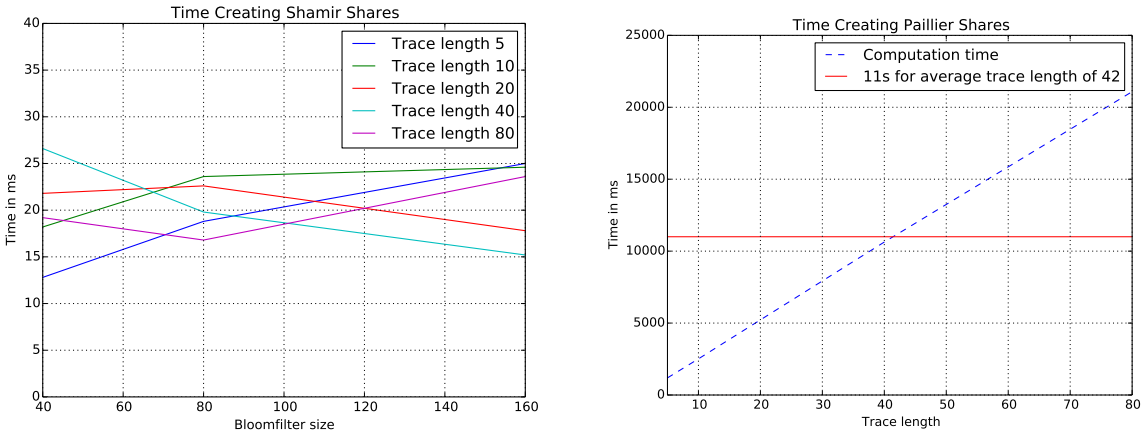

Figure 8.7 The left diagram shows how calculation time of Shamir shares behaves with increasing bloomfilter size. The right diagram shows how Paillier share creation time increases with the trace length. Here, we used a fixed key size of 2048bit for PTE.

8.2.2 Input Peer Evaluation

In Figure 8.7 we see how the computation time on the input peer behaves. As stated in Chapter 5, the input peers firstly share the trace bloomfilters using Shamir's Secret Sharing and secondly share the OSM nodes and altitude differences using Paillier Threshold Encryption. Hence, the amount of created Shamir shares depends on the trace length - one bloomfilter for each node - and on the chosen bloomfilter size. As depicted in Figure 8.7 on the left side, the time required for creating the Shamir shares does not considerably differ for the chosen trace lengths and bloomfilter sizes. Moreover, with a range between *13-26ms* for Shamir computations their timely influence is negligible. The time consumption of creating PTE shares is directly dependent from the trace length - we encrypt every altitude difference together with their source and destination OSM node. On the right side of Figure 8.7 we see, the share creation time consumption in dependence to the trace length. Where the time consumption develops linear between *1190ms* for *5* node traces and *21086ms* for *80* node traces. The red line shows the time consumption of *11000ms* for the average trace length of *42* nodes stated in the first part of this chapter. Note, that all computations are usually performed on smartphones. As the currently used Java implementation of PTE is single-threaded and smartphone CPUs still not reach the computation power of desktop CPUs it is likely that the computations take more time on the smartphone. Hence, we perform the computations in a background thread as stated in Chapter 6.

8.2.3 Privacy Peer Evaluation

We continue by evaluating the performance and scalability of our anonymization system realized by the privacy peers. To give a first overview over the system performance, we show in Figure 8.8 on the left, how the overall computation time per privacy peer scales with trace length. For this, and the following experiments we prepared pairs of traces where only the last node has the same ID. Thus, finding intersections involves checking *all* bloomfilters for the two traces. The total computation time includes the following processes:

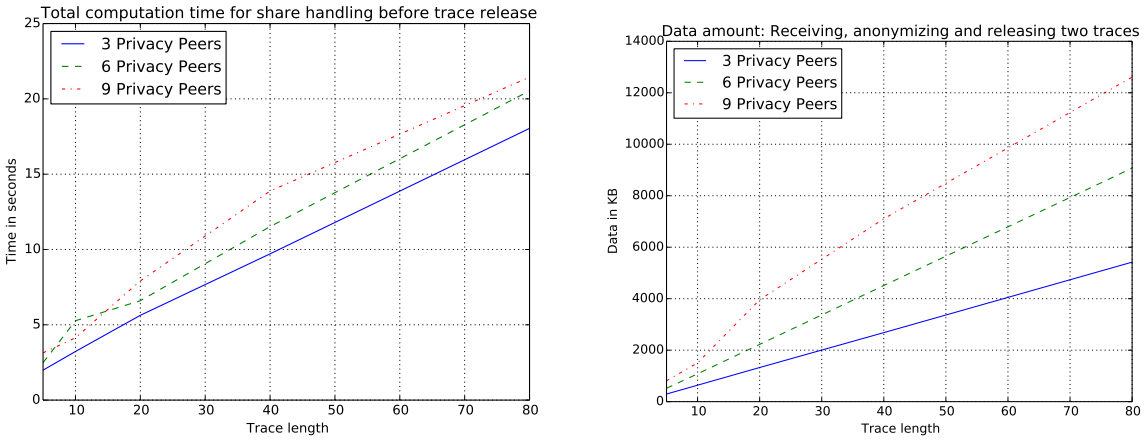

Figure 8.8 The left figure shows the complete computation time in seconds per privacy peer in dependence of trace length for 3, 6 and 9 privacy peers. On the right side, we see the traffic caused per privacy peer depending on the trace length.

- Calculating the intersection bloomfilter of the two traces
- Checking the single nodes bloomfilters against the intersection bloomfilter
- Shuffling the PTE encrypted tuples
- Doing a partial decryption of the tuples before releasing them to the collector peer

For each test we keep the bloomfilter size at 40 bits and the PTE key size at 2048 bits. All tests are repeated five times and the average time is taken as result. We see in the figure that the computational effort rises almost linearly for all constellations with 3, 6 or 9 privacy peers. Computation time starts with *2-4s* for traces with 5 nodes. The test with a trace length of 10 shows an outlier as the 9 privacy peer scenario yields a shorter time compared to the 6 privacy peer scenario. The complete test has been repeated several times where each attempt showed such outliers for a random trace length. As we use the smallest instances on Amazon EC2 we have no guarantees on their constant network performance, hence, these outliers probably originate from other load at the host systems network. Finally, we see that increasing the number of privacy peers adds on average a constant *0.8s* of computation time per added privacy peer. Based on the numbers from this figure, we can see that our system is able to anonymize *4-6* traces of length 40 per minute, depending on the amount of privacy peers.

The right side of Figure 8.8 shows the amount of data traffic used at each privacy peer during the reception, anonymization and release of two input traces. Again, we see that network traffic increases linearly with trace length but with higher increase for more privacy peers. As stated in Chapter 6 we decided to send numbers larger than the usual primitives as strings. This accounts for a majority of the network traffic. In future work, this is a point for significant improvements.

The three major time consuming actions at the privacy peers are:

- Finding intersections between contributed traces.

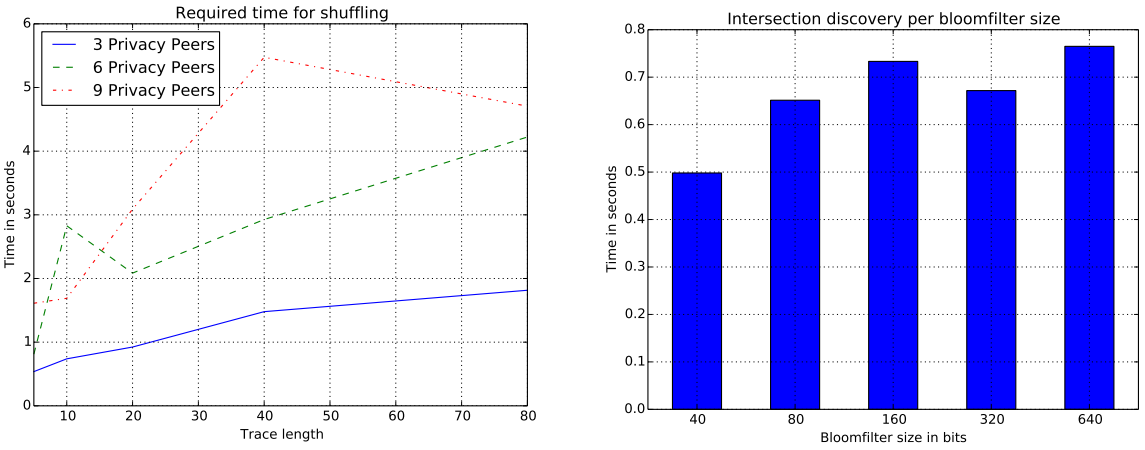

Figure 8.9 The left diagram depicts the average time per peer required for shuffling depending on the trace length. The right diagram shows the average intersection time with increasing bloomfilter size.

- Shuffling and re-randomizing the PTE encrypted altitude difference tuples.
- Performing the partial PTE decryption of these tuples before they get released to the collector peer.

The first privacy peer action i.e., finding intersections of traces using bloomfilters, turns out to be almost independent from the trace length and the intersection points. We evaluate the intersection time for traces of lengths 5, 10, 20, 40 and 80 nodes. For all lengths finding intersections took between *0.65s* and *1.05s*. Interestingly, the number of privacy peers has no noticeable impact even though finding intersections involves several distributed multiplications of Shamir shares.

Further, we show on the right side of Figure 8.9 that the intersection time generally increases along with the bloomfilter size, but also has outliers. Reasons for outliers are one of the following:

- The common node of the intersecting traces is one of the first nodes.
- Network delay has a higher impact on time than computation time.

While comparing single node bloomfilters of the second trace with the intersected bloomfilter of both traces, we start with the bloomfilter of the first node. As soon as a match is found we abort the search as we already found an intersection. Hence, the intersection time for traces with common nodes in the beginning is short. As the second point, Shamir share calculations are not computationally intensive. A delay prone network setup between the privacy peers as used in our evaluation has a higher impact on the overall intersection time compared to the pure calculations.

When two traces have been detected to have an intersection, we merge these traces and, in order to establish unlinkability as stated in Chapter 4, we shuffle the order of their shares. We evaluate for 3, 6 and 9 privacy peers with fixed bloomfilter size of 40 bits and fixed PTE key size of 2048 bits how the shuffling and re-randomization process scales with the trace length. The results are shown in Figure 8.9. Shuffling

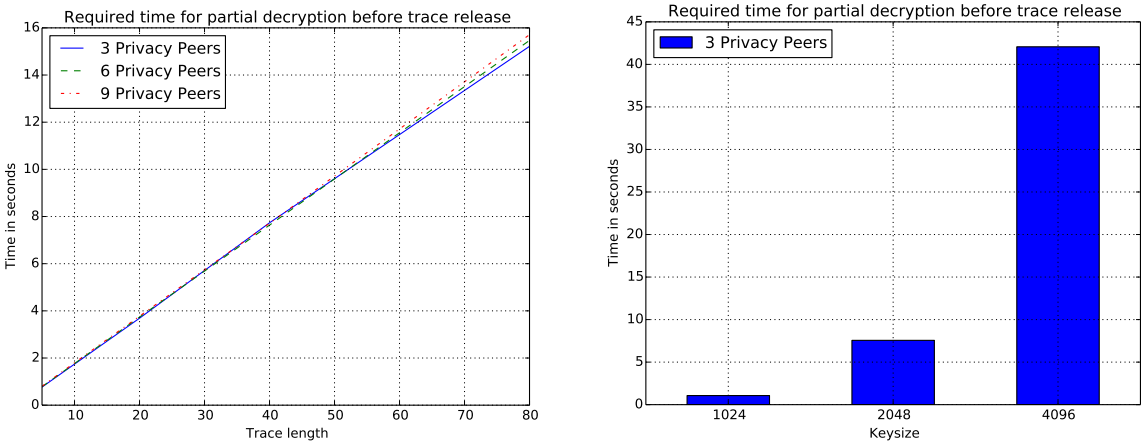

Figure 8.10 On the left side we show how the time for partial decryptions for PTE encrypted tuples behaves with trace length. On the right side we show the increase in computation time for growing PTE key sizes where each bar shows the average decryption time over 3 privacy peers.

does strongly depend on the network connection, hence, jitter and throughput shortages directly affect the shuffling time. This is also the reason for the 6 peer scenario taking more time than the 9 peer scenario for traces of length 10. The remaining trace length do not show this anomaly. Again, we can say that time increases linearly with the trace length but with higher increase the more privacy peers are used.

As soon as a merged and shuffled trace reaches the anonymity requirement k, each of the privacy peers performs a partial decryption of the PTE encrypted altitude tuples to release the partial decryptions to the collector peer. The decryption does not require peer interaction such that the process can be done locally. However, the decryption time is the most significant part of the total computation time shown in 8.8. The decryption time depends on the following variables:

- The trace length
- The Paillier Threshold Encryption key size

With increasing trace length, also the number of PTE encrypted tuples increases. We depict in Figure 8.10 for 3, 6 and 9 privacy peers how the decryption time per peer develops for traces with 5, 10, 20, 40 and 80 nodes. The figure clearly shows that the partial decryptions do not rely on the number of privacy peers. For all three scenarios, the time needed for partial decryptions is almost identical and increases linearly with the trace length.

The key size we use for PTE has an influence on the computation time needed to de- and encrypt a single PTE value. In the diagram on the right side of Figure 8.10, we show how the time for partial decryptions increases for a 40 node trace with growing key size. We tested the decryption time for key sizes of *1024, 2048* and *4096* bits. The results show that while we double the PTE key size, the computation time increases by a factor of approximately six. Including this into the total time per merged trace from the left diagram in Figure 8.8 decreases the per minute throughput of processed traces from *4-6* to *0-1* for a key size of 4096 bits. Hence, choosing 2048 bits is a feasible tradeoff between security and utility.

8.2.4 Collector Peer Evaluation

Finally, we examine the collector peer. The collector peer receives the partial decryptions from the privacy peers and uses them to reconstruct the secret. In our case, the OSM node IDs and altitude differences. We evaluate the time needed to reconstruct the tuples from two merged traces each one having a length of 5, 10, 20, 40 and 80 nodes. The chosen PTE key size is 2048 bits.

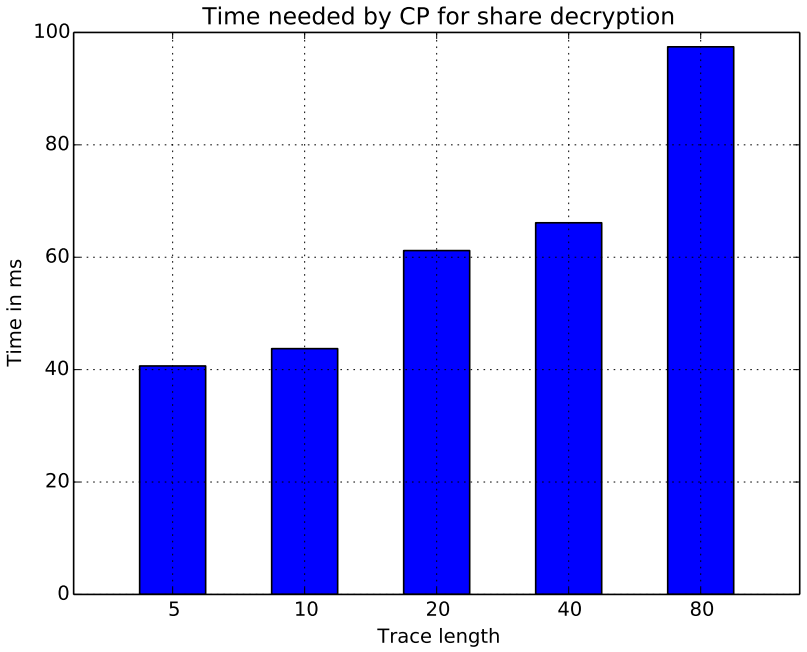

Figure 8.11 Average time needed for the collector peer to reconstruct a trace from the partial PTE decryptions of 9 privacy peers.

Figure 8.11 states that the reconstruction of traces ranges from approximately *40ms* up to *98ms*. However, the time for reconstructing traces only increases significantly for the traces with 20 and 80 nodes. The reason is that the collector peer uses multiple threads for the decryption. We use a machine with 16 cores and 32 threads where one core decrypts one PTE secret. Hence, the computation time for 5 and 10 node traces is almost identical. The same holds for the 20 and 40 node traces. We can further see, that the reconstruction of traces is much faster than the anonymization process at the privacy peers. Hence, the trace reconstruction is not a bottleneck.

8.2.5 Evaluation Summary

In our evaluation we investigated both, the accuracy of current smartphone barometers and the feasibility of a SMPC based anonymization system for trace collection. We have shown that our measurement technique as described in Chapter 5 is feasible to gather useful altitude differences between neighboring OSM nodes. Moreover, air pressure based altitude differences yield a better result compared to GPS by a factor of approximately six, see Figure 8.4. The overall accuracy for anonymized traces is lower compared to our self recorded traces as the distance between consecutive measurements is higher. OSM nodes have an average distance of approximately 125m where consecutive GPS trace points from manual recordings account for a distance of

1.11m assuming a travel speed of 4 km/h. We yield an average accuracy in altitude difference between neighboring nodes of 0.99m over 95 traces.

Regarding the SMPC based anonymization we evaluated a scenario of world wide distributed privacy peers. According to our figures from Section 8.2 our system properly scales with increasing trace length, where we achieve a throughput of 4 to 6 anonymized traces per minute for traces of length 40. This value can be easily increased by assigning more CPU power and using less delay prone network connections.

9

Conclusion

In this thesis we proposed a system for crowd-sourced privacy-preserving collection of user created elevation profiles. Therefor, we approached the problem of privacy-preserving collection for location dependent data and the problem of creating elevation profiles based on air pressure based altitude differences. The system is split into three parts. Firstly, the Android application that gathers air pressure data for the user's positions. Secondly, the privacy peers that receive shares of contributed traces and establish *k-anonymity* on the data under secure multi party computation before releasing *k-anonymous* chunks of altitude differences. Lastly, the potentially untrusted collector peer receives the anonymized chunks and is not able to draw conclusions on who contributed the data (see *unlinkability* in Chapter 4) and who is represented in the data (see *untraceability* in Chapter 4).

9.1 Evaluation Discussion

In the evaluation of air pressure based altitude differences we saw that our barometric approach performed better compared to GPS by a factor of six for trace points with an average distance of 1.11m. We were able to yield an average error of 0.12m for altitude differences compared with a reference map for the region of Aachen, Germany. This value is remarkable as the typical accuracy of the Nexus 5 barometer is 0.12mBar according to [4]. An air pressure difference of 0.12mBar translates to an altitude difference between 0.8m and 1m (depending on the absolute air pressure). Hence, we can conclude that gathered air pressure values are usually more accurate than stated by the typical accuracy in [4]. Further, for short distances between trace points we yield an accuracy of 0.12m which is approximately 6.67 times better than the GPS based value in the exact same scenario. Regarding the accuracy of anonymized traces we cannot compare them against GPS as we do not collect GPS altitude with our system. However, we notice that the accuracy of altitude differences between neighboring OSM nodes decreases with increasing node distance.

Most probably, weather effects get a higher impact on the accuracy with increasing node distance.

Further, temperature has an impact on air pressure that we did not consider in this thesis. For instance, users taking their smartphones out of their pocket move it in a colder environment effectively measuring lower air pressure values. Moreover, tests in [21] show that blowers in cars account for a noticeable change in air pressure, hence, being a further error source. Finally, we reach an accuracy of 0.99m on average over 95 traces with an average trace length of 42 nodes. All these traces have been collected through our anonymization system. Considering traces with 42 nodes and an average distance of 125m between nodes the accuracy of 0.99m accounts for traces with a length on average 2,635km. Hence, we consider the collection of air pressure based altitude difference feasible for use in pedestrian navigation e.g., finding routes with least elevation for wheelchair users or determining the difficulty of biking tours.

The evaluations of our SMPC based anonymization system have shown that it scales linearly in terms of computation time with the trace length. Looking at Figure 8.8, confirms this behavior and shows that rising amount of privacy peers adds a constant additional computing time of approximately 0.8s per privacy peer. The additional computation time is reasoned by the additional shuffling effort where shares have to travel a complete round across all privacy peers. The biggest portion of the complete computation time is accounted by the partial decryption of PTE shares before they get released to the collector peer. However, this computation only occurs if a traces reaches its anonymity requirement and hence gets released.

The partial decryption time consumption depends firstly on the trace length and secondly on the PTE key size. As we can not influence the trace length, we use the PTE key size to find a wise decision for a tradeoff between computation time and security. As 2048 bits are considered to be sufficient also for future use, looking at Figure 8.10 we can say that 2048 bits is a suitable tradeoff as 4096 bits increases the computation time by a factor of 6. We further see, that the amount of privacy peers does not have a significant influence on the decryption time as decryption is performed locally at every privacy peer.

Using 2048 bits for PTE shares also yields a good performance for trace decryption at the collector peer size. Depending on the trace length the decryption takes up to 97ms for an 80 node trace. Hence, the collector peer is not the bottleneck and allows for processing large amounts of incoming traces.

Finding a suitable value for the bloomfilter size depends on the average trace length. Undersized bloomfilters yield false positives if they get overcrowded as stated in Chapter 5. In Chapter 8 we found an average trace length of 42 nodes. Together with the fact from the right diagram in Figure 8.9 which states that the bloomfilter handling's share on the overall computation time is negligible, we suggest to use bloomfilters of at least 320 bits and 4 hashes per object. Thus, we are able to process even longer traces without the danger of occurring false-positives during the intersection detection.

Possibilities to scale the SMPC anonymization system are either the use of more performant systems, or the deployment of multiple parallel privacy peer systems to spread the load of contributors. Remember, that our evaluations were done on

the smallest Amazon EC2 instances with least guarantees on CPU and network performance.

9.2 Future Work

Our system is in the state of a proof of concept. As the evaluation shows it already performs well, but there are several ideas that still need to be realized for productive use. A list of these ideas looks as follows:

- Implement the aggregation of altitude differences.
- Improve the network communication for big integers.
- Implement a connection to OSM for direct result release.
- Implement geofencing in the Android application.
- Secure the connections between all participating peers using TLS.
- Let the user choose the anonymity requirement k.

In the current state we save all received traces. However, there is no handling for the received data i.e., when receiving multiple information for the same pair of OSM nodes, these are saved but not aggregated. As suggested in Chapter 5 we recommend to implement a sliding window algorithm that calculates that average altitude difference of the last five measurements for each pair of OSM nodes. This way, changes in ground level get incorporated over time.

As the currently used library for serialization and deserialization does not provide primitives for big integers, we send them as strings. This causes a majority of the network traffic and is a good point for improvements.

As already stated, the collected traces are only stored by our proof of concept implementation but do not go through further processing. As we want to contribute the data to the public we need an implementation that connects our system with OSM.

Even though our system establishes k-anonymity on contributed data, a users anonymization is reasoned by intersection points. Hence, that part of the contributed trace up to the first intersection point is not anonymized such that we can possibly see that a user came from a certain address of travelled to this address. As a future feature we can empower the user to determine areas on the map where no traces should be recorded to avoid this problem.

As stated in Chapter 7, all connections must be secured in order to prevent an adversary from intercepting the shares among the privacy peers. This implementation is also considered future work.

As a last possible feature for future work, we should enable contributors to choose the desired anonymity requirement k for their traces. This way, each user can ensure that her contributed traces at least exhibit her desired level of anonymity before collected by the collector peer.

9.3 Problem Revision

In Chapter 4 we stated a list of requirements on our system which are *Anonymity, Utility, Scalability* and *Coverage*. In the following we show to which extent these requirements are fulfilled:

Anonymity Our system addresses both problems user *unlinkability* and *untraceability*. We to hide a traces origin and hence, to establish unlinkability, we shuffle the contributed traces at the privacy peers. After the shuffling process no peer can tell which part of the trace was contributed by which input peer. Further, we establish k-anonymity on the traces by merging those traces that intersect with each other before they get released to the collector peer. Ultimately, both problems get solved by our system.

Utility We must ensure that anonymized data still contains all necessary data to fulfill the use case of creating elevation profiles. The only loss of precision occurs with the discretization of GPS traces to OSM nodes. However, as we want to calculate the altitude differences between OSM nodes, the data has exactly those information that we need i.e., a pair of nodes and the altitude difference between them.

Scalability As shown in Chapter 8 the SMPC trace anonymization scales linearly with the trace length. Adding more privacy peers only adds a constant value to the computation time. As we can process between 4 and 6 traces per minute on the smallest instances on Amazon EC2 we can increase this value by using stronger instances. However, assuming several thousand contributors probably requires some speed optimizations. Data collection via our Android application is performed totally independent. Hence, also the process of collecting data easily scales with the number of users. We have further shown, that also the collector peer computations scale linearly with rising trace length.

Coverage We achieve an easy system deployment by using an Android application as input peer to our system. This way, everyone having an Android smartphone with barometer can join and contribute to the system.

Bibliography

[1] Online Resource. https://www.openstreetmap.org, accessed 2014-08-22.

[2] Online Resource. https://en.wikipedia.org/wiki/Altimeter, accessed 2014-09-02.

[3] Online Resource. https://www.gaia-gis.it/fossil/libspatialite/index, accessed 2014-09-10.

[4] Online Resource. http://www.bosch-sensortec.com/en/homepage/products_3/environmental_sensors_1/bmp280/bmp280, accessed 2014-09-18.

[5] Beresford, A. R., and Stajano, F. Location privacy in pervasive computing. *Pervasive Computing, IEEE 2*, 1 (2003), 46–55.

[6] Brickell, J., and Shmatikov, V. Efficient anonymity-preserving data collection. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining* (2006), ACM, pp. 76–85.

[7] Burkhart, M., Strasser, M., Many, D., and Dimitropoulos, X. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. *Network 1* (2010), 101101.

[8] Community, O. Osm altitude information. Online Resource. http://wiki.openstreetmap.org/wiki/DE:Altitude, accessed 2014-07-28.

[9] Cramer, R., Damgård, I., and Maurer, U. General secure multi-party computation from any linear secret-sharing scheme. In *Advances in Cryptology—EUROCRYPT 2000* (2000), Springer, pp. 316–334.

[10] Diverse. Barometric altitude formula. Online Resource. https://de.wikipedia.org/wiki/Barometrische_Höhenformel, accessed 2014-07-30.

[11] Farr, T. G., Rosen, P. A., Caro, E., Crippen, R., Duren, R., Hensley, S., Kobrick, M., Paller, M., Rodriguez, E., Roth, L., et al. The shuttle radar topography mission. *Reviews of geophysics 45*, 2 (2007).

[12] Fung, B., Wang, K., Chen, R., and Yu, P. S. Privacy-preserving data publishing: A survey of recent developments. *ACM Computing Surveys (CSUR) 42*, 4 (2010), 14.

[13] Graham, M. Gps versus barometric altitude the definitive answer. Online Resource. http://www.xcmag.com/2011/07/gps-versus-barometric-altitude-the-definitive-answer/, accessed 2014-07-30.

[14] Gruteser, M., and Grunwald, D. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proceedings of the 1st international conference on Mobile systems, applications and services* (2003), ACM, pp. 31–42.

[15] Gruteser, M., and Hoh, B. On the anonymity of periodic location samples. In *Security in Pervasive Computing*. Springer, 2005, pp. 179–192.

[16] Hoh, B., Gruteser, M., Xiong, H., and Alrabady, A. Achieving guaranteed anonymity in gps traces via uncertainty-aware path cloaking. *Mobile Computing, IEEE Transactions on 9*, 8 (2010), 1089–1107.

[17] Lindell, Y., and Pinkas, B. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality 1*, 1 (2009), 5.

[18] Nave, C. R. Barometric altitude formula. Online Resource. http://hyperphysics.phy-astr.gsu.edu/hbase/kinetic/barfor.html, accessed 2014-07-30.

[19] Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology—EUROCRYPT'99* (1999), Springer, pp. 223–238.

[20] Paillier, P. Paillier encryption and signature schemes. In *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 902–903.

[21] Parviainen, J., Kantola, J., and Collin, J. Differential barometry in personal navigation. In *Position, Location and Navigation Symposium, 2008 IEEE/ION* (2008), IEEE, pp. 148–152.

[22] Shamir, A. How to share a secret. *Communications of the ACM 22*, 11 (1979), 612–613.

[23] Silverman, R. Has the rsa algorithm been compromised as a result of bernstein's paper? *RSA Laboratories, April 8* (2002).

[24] Sweeney, L. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10*, 05 (2002), 557–570.

[25] Zhenhai, H., and Shengguo, H. Development of high precision barometric altimeter [j]. *Journal of Nanjing University of Aeronautics & Astronautics 1* (2009), 028.

[26] Zhu, W., Dong, Y., Wang, G., Qiao, Z., and Gao, F. High-precision barometric altitude measurement method and technology. In *Information and Automation (ICIA), 2013 IEEE International Conference on* (Aug 2013), pp. 430–435.