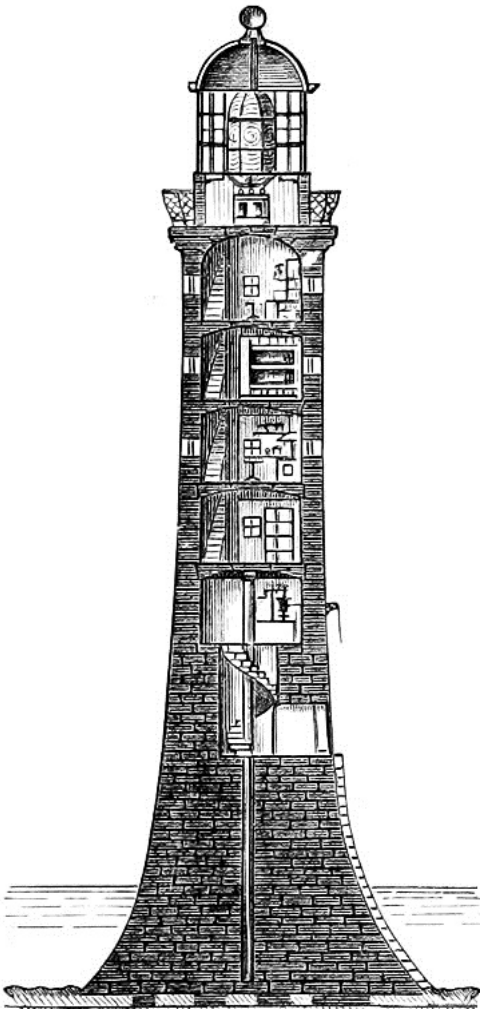
Augustin Zidek



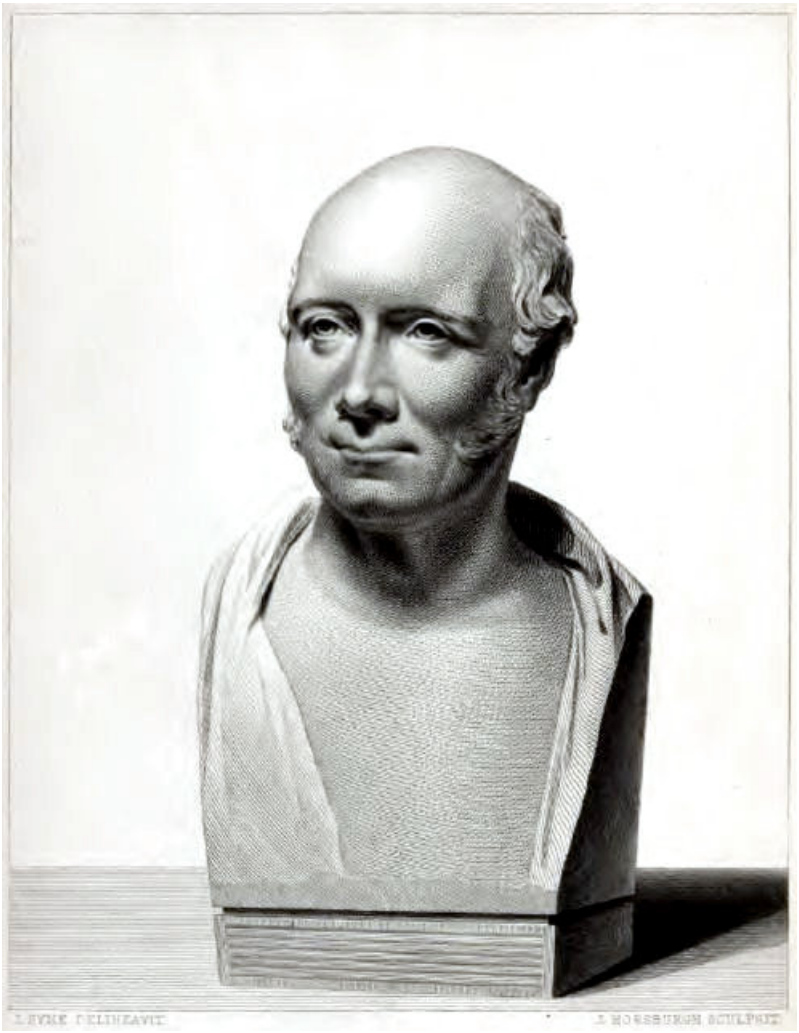
Bellrock – your phone as an anonymous BLE beacon

Computer Science Tripos, Part III

Gonville and Caius College

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for Computer Science Tripos Part III

June, 2016



Robert Stevenson was a Scottish civil engineer who built the Bell Rock Lighthouse between 1807 and 1810. The Bell Rock Lighthouse, pictured on the cover page, is the world's oldest surviving sea-washed lighthouse. [1]

Proforma

Name:	**Augustin Zidek**
College:	Gonville and Caius College
Project Title:	**Bellrock – your phone as an anonymous BLE beacon**
Examination:	Computer Science Tripos, June 2016
Word Count:	**11,916**[1]
Project Originator:	Robert Harle, Augustin Zidek
Supervisor:	Dr. Robert Harle

Declaration

I, Augustin Zidek of Gonville and Caius College, being a candidate for Part III of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

[1] Computed using `texcount`, proforma and anstract excluded

Abstract

This dissertation describes the Bellrock system which consists of Bellrock clients and the Bellrock server. The Bellrock clients serve as dynamic BLE beacons, that is BLE beacons on smartphones which move with their users. Because broadcasting static UUIDs of the clients would breach user privacy and anonymity, Bellrock introduces Anonymous IDs (AID), which are encrypted UUIDs. Such AIDs give away no information about the client to a random observer, but the Bellrock server can decrypt them and provide back to the clients the same kind of information as if UUIDs were broadcasted. This enables position context inference and proximity of friends. Moreover, this scheme offers offline AID changes, i.e. the clients do not need to be connected to the server to change their AID. Although symmetric key cryptography is used for AID encryption for privacy and technical reason, the server enhances the AID brute-force decryption performance using spatial and temporal heuristics and achieves amortised performance independent on the total number of users. The feasibility of the Bellrock approach was shown in office-lie environments using data about user movements in AT&T laboratory.

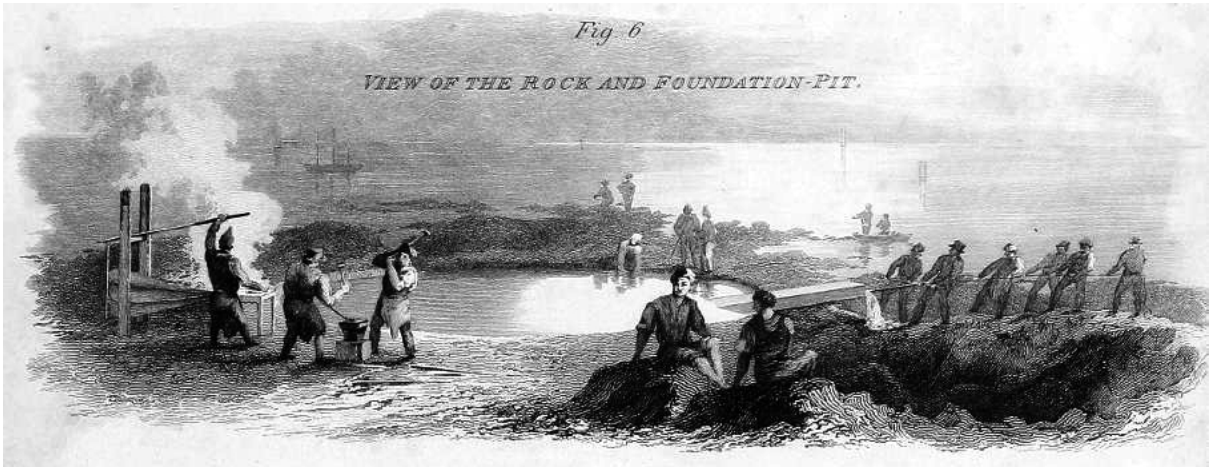
Contents

Acknowledgements

I would like to thank my supervisor Rob Harle for supervising this project, for asking good questions that usually revealed Bellrock's weaknesses and for excellent overall guidance.

I would also like to thank Petra Vlachynská not only for proof-reading and spotting the tons of missing definite and indefinite articles. My years at the university would not be what they were without you.

Last but not least, I would like to thank my family and my friends, Fruzsina in particular, for providing useful comments and listening to me as I was trying to explain what Bellrock is actually about.



The beginning of the works at Bell Rock. [2]

Chapter 1

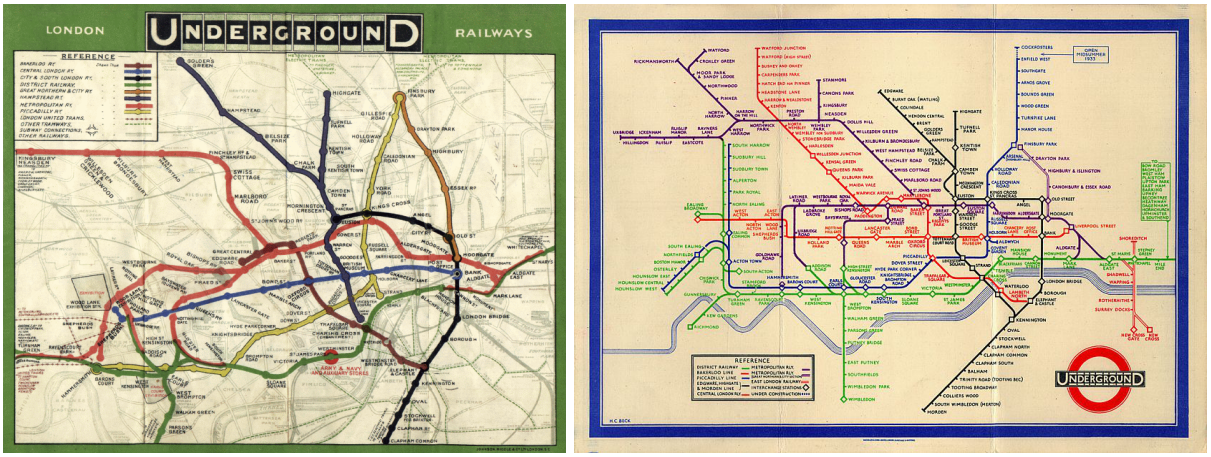
Introduction

Since 2007, when Apple introduced the first iPhone, smartphones have become ubiquitous in developed countries, reaching over 68% market penetration in the UK [3]. Smartphones provide not only ubiquitous internet access, but also access to the Global Navigation Satellite Systems (such as GPS), hence enabling reliable outdoor navigation capabilities.

However, it is still a hard problem to provide precise and reliable indoor navigation. This would be useful for instance for navigation in shopping centres, underground commuting systems or in emergency situations.

The key thought behind this project is the following observation: Human users are usually not interested in their exact location coordinates. Rather, they are interested in the *context* in which they currently are.

This is well illustrated for instance by the 1933 Tube map by Henry Charles Beck [4]. Rather than capturing the exact shape of the Tube, Beck focused on the *structure* of the Tube. His map (diagram, better said) preserves basically the only important aspect of the Tube – the adjacency of its stations.



(a) The original Tube map from 1908 [5] (b) Topological Tube map from 1933 [4]

Figure 1.1: The evolution of the London Tube map

Similarly, people are not interested in where they are, but rather in what is next to them. While computers prefer (52.2111071, 0.0915418), humans would gain more by hearing "in front of WGB, Cambridge, UK".

Moreover, most people usually meet more or less the same set of people every day. People meet their family at home, their fellow commuters on their way to work and their colleagues at their workplace. They meet the same shop assistants at the supermarket and they meet the same people in church on Sundays.

The Bellrock system turns everyone's smartphone into a beacon without compromising users' privacy and anonymity. Each device listens to the beacons of the others and based on the IDs of the devices it hears, it can perform certain actions, such as notify you about the proximity of your friends or provide location-based services.

Rather than focusing on various use-case scenarios, Bellrock provides the platform (i.e. client, server and a set of protocols) to achieve the aforementioned. The most useful use-case for Bellrock would probably be indoor positioning, though. The main focus of the Bellrock project is on the privacy of its users which is achieved via an anonymised encrypted ID scheme and a trusted server used for the anonymous ID resolution.

Since Bellrock is loosely coupled with indoor positioning, the following section provides the background on indoor positioning. It also explains the motivations behind dynamic beacons.

1.1 Indoor positioning

Many indoor positioning systems have been proposed, based on Wi-Fi, Bluetooth, ultrasound, visual information, GPS or GSM fingerprinting [6, 7]. These systems achieve up to 1 cm accuracy. Since Bellrock uses Bluetooth Low Energy beacons, these will be discussed in detail.

Positioning using Bluetooth beacons is *not* done using lateration, i.e. by estimating the distances to multiple beacons with known position using the received signal strength (RSS). This is mainly due to the multi-path problem which is illustrated in Figure 1.2.

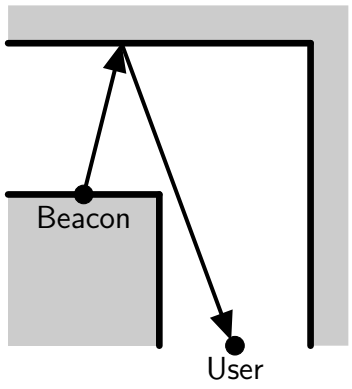


Figure 1.2: Multi-path problem with indoor beacons: the user's phone estimates (using the RSS) that it is much further from the beacon than it really is

Hence, iBeacon [8] and AltBeacon [9] architectures deploy beacons with low power and estimate the location only based on the nearest beacon. This implies lower precision – the position can be anywhere in a sphere around the beacon – but the main purpose of iBeacons

and AltBeacons is to provide location-based services, such as information about pictures in an art gallery or advertisement for a nearby restaurant.

It is possible that in the future every smart device (light bulbs, fridges, door locks, etc.) will have an incorporated beacon that will broadcast their unique ID. Once this happens, the accuracy of indoor positioning using beacons will improve as there will be much more data to infer the position from: with thousands or more beacons in a building, every place will have a unique set of observable beacons identifying the place.

1.1.1 Problems of static beacons

Static beacons have multiple disadvantages. The main two are the following:

1. **Lack of authority.** There is no guarantee that beacons will be at the locations they advertise. They can be turned off, lose power or they can be easily spoofed. Moreover, each beacon has to have an owner who maintains it.
2. **Lack of deployment.** If one wishes to use beacons for indoor positioning, a lot of such beacons have to be deployed inside a building. While Global Navigation Satellite Systems are present basically everywhere outdoors, this is not the case with beacons. Rarely any building is equipped with Bluetooth beacons and there is no centralised database of beacon locations.

It is therefore very challenging to deploy a large-scale indoor positioning system based on static beacons. This is why Bellrock takes the approach of dynamic beacons.

1.2 This project

The motivation for Bellrock comes from the fact that static (BLE) beacons are not ubiquitous yet. Bellrock is hence a trade-off between the available technology and the goal it tries to achieve. The trade-off comes in the form of *dynamic* beacons – beacons that are not static and move around with their users. While this limits the possible information that can be inferred, there is no need to deploy the beacon infrastructure. By using smartphones as BLE beacons, there are already millions of devices that could act as Bellrock beacons.

However, even with dynamic beacons, it is possible to infer a lot of information, such as proximity of friends, or even position, based on periodic user behaviour analysis.

The Bellrock project is based on the following key ideas:

1. Every smartphone acts as a beacon transmitting an Anonymous ID (AID) which is effectively random to observers, but the trusted Bellrock server can resolve it to a Unique User ID (UUID).
2. To prevent tracking, smartphones broadcast only when stationary. When the user moves, the broadcast is stopped. When user becomes stationary again, a new AID is broadcasted.
3. The trusted server gathers pieces of information from the clients and returns inferred information back to them.
4. The system is able to infer who and what is near rather than the exact location.

In the future, the Bellrock system could serve as a basis of indoor positioning system based on spatial and temporal space fingerprinting, it could be used to infer proximity of people, buildings or even animals. Moreover, in a world where there are thousands of (BLE) beacons in every house, Bellrock could unite static and dynamic beacons into a single gigantic dynamic BLE grid. The grid would have its own routing protocols and it would enable communication of offline BLE devices through nearby BLE device that have internet connection.

1.2.1 Bellrock overview

Bellrock consists of two main components, the client and the server. This section gives a broad overview of both and Chapter 3 gives more detail.

Client

The client is written in Java and runs on Android phones. When the device is stationary, it periodically broadcasts AID using Bluetooth Low Energy (BLE). The client also listens for AIDs broadcasted by nearby phones and it records the observed AIDs along with the observation times in the internal database.

Each time a user starts moving, AID broadcast is stopped to prevent user tracking. When the user is stationary again, a new AID is generated and the broadcast is resumed.

The client periodically establishes a secure connection with the server and sends it the list of its AID observations. The client also sends the list of the Cell IDs it observed, thus imposing a rough estimate on its location. This makes the AID decryption on the server side faster, as discussed in Chapter 4.

Server

The server is also written in Java. Its main task is to receive observations of AIDs from clients, resolve each observation into a UUID and based on this information, infer proximity of friends or possibly other information.

Typically, each client observes thousands of AIDs per day (as shown in the Evaluation). The decryption performance is therefore crucial. The decryption module is the most sophisticated part of the server and it employs multiple heuristics, caching and implementation techniques to increase the performance. Even using modest hardware, the server is able to decrypt millions of AIDs per second.

1.2.2 Dissertation overview

The dissertation is organised into 7 chapters:

1. **Introduction** gives an overview of the Bellrock project.
2. **Background** gives an introduction into the technologies that are used in Bellrock.
3. **The Bellrock system** chapter gives an in-depth description of the Bellrock system, namely the Bellrock client and the Bellrock server.

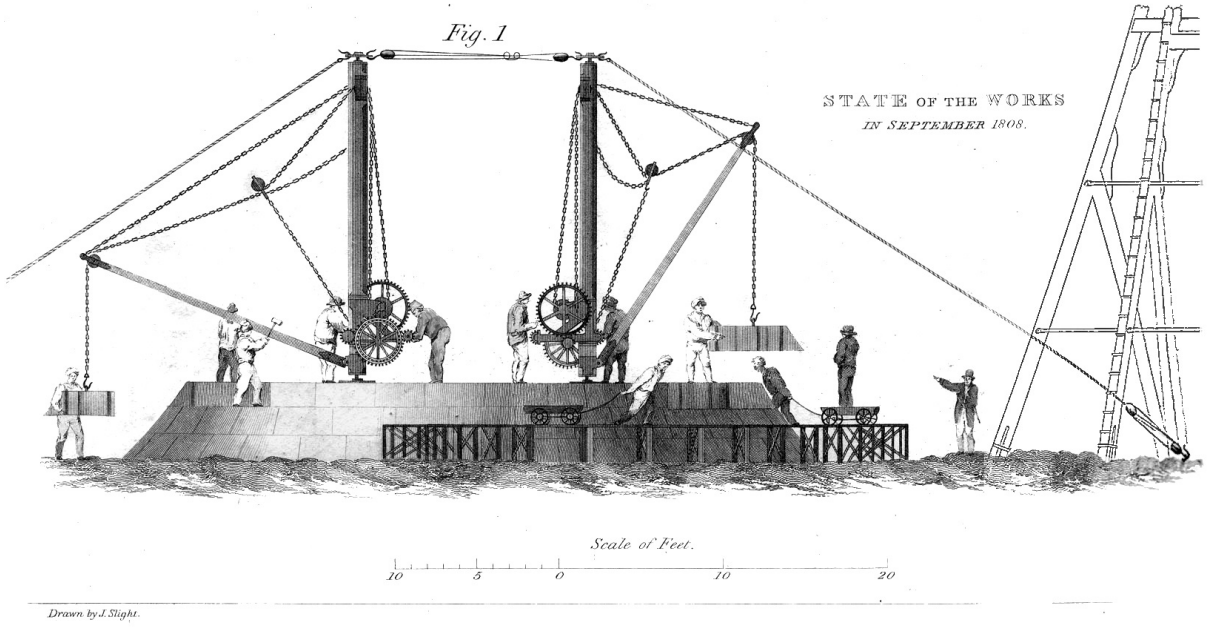
4. The **Achieving anonymity** chapter explains the technical arguments that lead to using symmetric cryptography rather than asymmetric cryptography and explains the techniques Bellrock uses to make this approach feasible and scalable.
5. The **Evaluation system** chapter serves as a background chapter for the Evaluation chapter. This chapter explains the details of the simulation that was used to evaluate the feasibility of the Bellrock system.
6. **Evaluation** shows that the approach taken by Bellrock is feasible and then gives detailed analysis of the Bellrock server performance. Possible attacks on Bellrock are discussed at the end of this chapter.
7. **Conclusion** closes the dissertation with an overview of what has been done and discusses possible extensions.

1.2.3 Why Bellrock

As a tribute to the great engineer Robert Stevenson, this project is named after the Bell Rock Lighthouse which Stevenson built between 1807 and 1810. The Bell Rock Lighthouse is the world's oldest surviving sea-washed lighthouse and it is still in service at the coast of Scotland. Lighthouses, or beacons, are a symbol of hope or certainty. Similarly, this project aims to give smartphones more certainty about the context they are in.

This dissertation is accompanied by original drawings and Bell Rock Lighthouse plans taken from Robert Stevenson's book *An Account of the Bell Rock Light-house* [2] which he wrote fourteen years after completing the Bell Rock Lighthouse.

Robert Stevenson is known for the construction of 16 lighthouses, 5 bridges and numerous lighthouse innovations, such as new light sources, mountings and reflector designs, the use of Fresnel lenses and for shutter systems that provided each lighthouse with a unique signature that could be identified by ships.



Progress of the Bell Rock Works, engraving by William Miller. [2]

Chapter 2

Background

Bellrock uses multiple technologies that will be explained first in order to let the reader understand the technical decisions made later. In particular, Bluetooth Low Energy, Cell Tower ID and Android Movement Activity API are discussed in this chapter.

2.1 Related work

Currently, there is a range of positioning systems available. The following table compares the main ones together with Bellrock.

System	Advantages	Disadvantages	Accuracy
GNSS (e.g. GPS)	Works everywhere outside, cheap clients	Only outside, power intensive	Position on a street
Cellular	Low power, always available	Low accuracy	Area in a city
Wi-Fi	High availability in cities	Low accuracy, power intensive	Area within a city block
BLE beacons	Low power, high precision	Low availability, difficult deployment	Area within a room or floor
Bellrock	Low power, high precision, available when other Bellrock devices around	Requires a server, location resolution not guaranteed	Building or a room within a building

It is important to emphasise that Bellrock is not primarily an indoor positioning system. Bellrock is a device proximity framework which is not limited to location only, it can also provide other information (friend proximity). Moreover, it can be combined with other technologies (such as Wi-Fi fingerprinting or static BLE beacons) to provide more reliable information.

Multiple systems using BLE beacons have been proposed. Feldmann et al. [10] proposed a system based on BLE beacons where lateration based on received signal strength is used to obtain a position approximation. They report functionality of the system only when the distance between the phone and the beacon is less than 8 meters with a precision of 2 meters.

Such precision is expected because signal interference and multi-path skew the measurements.

Inoue et al. [11] improve on Feldmann by first scanning the entire area, thus obtaininga "fingerprint" for each point which they later use to estimate the location. They report better precision of 0.5–2 meters. However, their system is dependent on the invariance of the area, e.g. when the furniture is moved, the precision of the system degrades. Moreover, their solution requires a server, hence requiring the clients to be online which decreases the power efficiency.

Choi et al. [12] describe a system based only on the proximity to beacons, which is the same approach as Bellrock has taken. Their use-case is not positioning, but rather turning off lights and PCs in an office environment when the user gets out of their range. Cheung et al. [13] give details of an inexpensive trick which limits the range of BLE beacons to make them more suitable for proximity-based services so that it is better defined which of the beacons is the nearest one. The trick they describe consists of wrapping the BLE beacon in two layers of aluminium cooking foil, thus limiting the range of the BLE beacon.

2.2 Bluetooth Low Energy

Bluetooth Low Energy (BLE) is a radio communications protocol designed to maximise battery lifetime and support short Internet-of-Things-like interactions between devices. It was introduced in the Bluetooth 4.0 standard [14] and operates in the 2.4 GHz ISM band at 2400–2483.5 MHz. Forty channels are used with centre frequencies at $2402 + 2k$ MHz for $k \in \{0, 1, \ldots, 39\}$. BLE has the notion of *advertisements*, which are short packets of data periodically broadcasted at a configurable rate. The radio channels 0, 12 and 39 are dedicated advertising channels and the remaining channels are data channels.

The advantage of using BLE in Bellrock is the negligible battery-life impact. When testing Bellrock, negligible battery impact was observed after 16 hours of broadcast at rate of approximately 1 advertisement per 5 seconds. Moreover, it seems likely that BLE will be the protocol powering the Internet of Things and it is becoming a ubiquitous protocol in new smartphones and other wearable devices.

However, there are two disadvantages of BLE:

1. **BLE packets are *very* short**. This is a vital part of the BLE protocol to enable its low power consumption. Thus, the length of packets is up to 31 bytes. However, there are further restrictions, since the data has to follow the Generic Attribute Profile (GATT) specification [14]. At best, this requires the following fields: `length` (1 byte), `data type` (1 byte) and `manufacturer UID` (2 bytes), leaving only 27 bytes for the user data. Moreover, if one is not a registered manufacturer, the manufacturer UID has to be 16 bytes long. In this project we assume the ownership of a two-byte manufacturer UID and therefore we have up to 27 bytes for the data.
2. **BLE is a young protocol.** While scanning and receiving BLE advertisements works on all devices with BLE support, only the most recent (or older, high-end) devices support sending BLE (advertisement) packets. While this is not a problem in the long-term, it imposed severe limits on the deployment of Bellrock and its testing with real users.

BLE has two modes of operation: advertising mode and connection mode. In the advertising mode, a device periodically sends short packets which anyone can listen to. It is therefore stateless and hence energy efficient. In the connection mode, a connection has to be estab-

lished first between two clients and then they can communicate. However, this mode is not used in Bellrock, as it requires time (and hence energy) to establish the connection. Moreover, the number of connections would be n^2 with n clients at a typical BLE coverage distance. All Bellrock clients therefore advertise and listen simultaneously.

2.3 Cell tower information

Thanks to the huge spread of mobile phones in the last 20 years, most of the inhabited parts of the world are now covered by mobile network(s). The last segment of the mobile network that communicates with the phones is the cell tower. Cell towers have ranges from 10 m (femtocell) to 40 km (large cell tower).

The Bellrock server uses cell tower information to determine an approximate location of its clients to assist the AID decryption. This is discussed in more detail in Section 4.3. Each cell tower (GSM, CDMA, UMTS or LTE) broadcasts the following meta-information:

- **Mobile Country Code (MCC):** a 3 decimal digit code defining the country where the cell tower is located.
- **Mobile Network Code (MNC):** a 3 decimal digit code defining the operator that operates the cell tower.
- **Location Area Code (LAC):** a 16 bit number defining the area *within the operator's network* where the cell tower is located.
- **Cell ID (CID):** a 28 bit number with the unique ID of the cell tower within the country, network and area.

This information can be packed into a `long` variable which enables efficient database storage:

```
1  public long pack(short mcc, short mnc, int lac, int cellID) {
2      long packed = 0;
3      packed += mcc;
4      packed <<= 10;
5      packed += mnc;
6      packed <<= 16;
7      packed += lac;
8      packed <<= 28;
9      packed += cellID;
10     return packed;
11 }
```

Similarly, unpacking can be done in the following way:

```
1 public CellTower(final long ctInfo) {
2     cellID = (int)    (ctInfo & 0x0000_0000_0FFF_FFFFL);
3     lac    = (int)   ((ctInfo & 0x0000_0FFF_F000_0000L) >>> 28);
4     mnc    = (short) ((ctInfo & 0x003F_F000_0000_0000L) >>> 44);
5     mcc    = (short) ((ctInfo & 0xFFC0_0000_0000_0000L) >>> 54);
6 }
```

2.3.1 OpenCellID

Bellrock uses the OpenCellID database [15] which collects crowd-sourced data of cell tower location measurements. The OpenCellID database, given cell tower information, returns the approximate location of the tower.

Since using the online version of the OpenCellID accessible through a web API was too slow (120 ms latency), the database was downloaded and a local version was incorporated into the server. The local version has access times smaller by 3 orders of magnitude, achieving less than 0.5 ms lookups. Bellrock uses the database in such a way that it is not important to update the database very frequently. This is assuming the cell tower operators don't periodically randomly change the Cell IDs.

The OpenCellID database (distributed in CSV format) was preprocessed with an `awk` script that got rid of the information redundant for the purposes of Bellrock. The filtered file was processed further and saved as a Java `HashMap` where the packed cell tower information was used as a key, enabling $O(1)$ lookups.

2.4 Android Movement Activity API

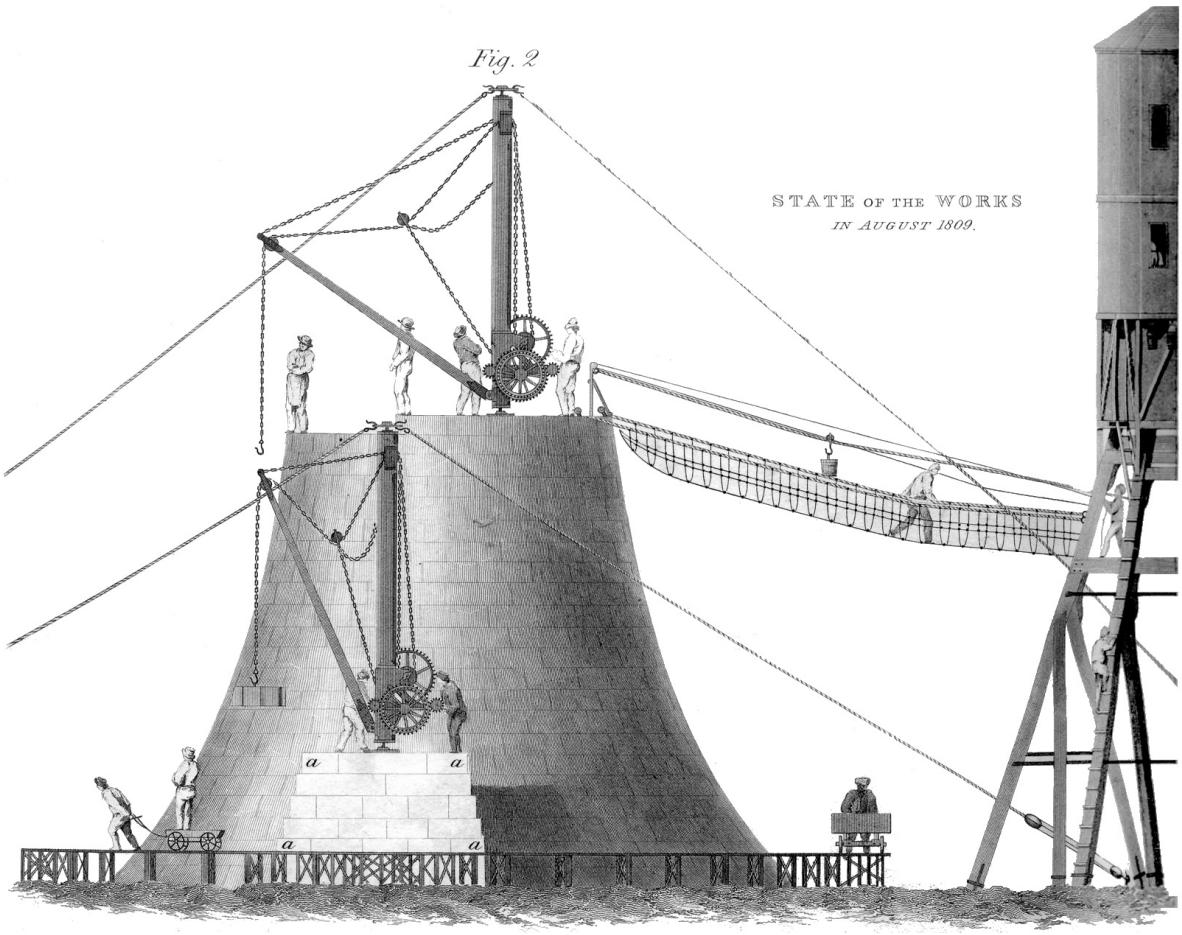
The Bellrock client needs to determine the movement activity (still, running, walking, etc.) of its user. This is done to prevent the Bellrock client from broadcasting an AID while the user is moving.

The Bellrock client uses Android Movement Activity API[1] to determine the movement activity of the user. Once Bellrock is registered as a listener of this service, Android notifies it every time it detects there was a movement activity change and gives Bellrock a list of detected activities, each with a probability. The following activities are distinguished:

- `IN_VEHICLE` – The device is in a vehicle, such as a car.
- `ON_BICYCLE` – The device is on a bicycle.
- `ON_FOOT` – The device is on a user who is walking or running.
- `RUNNING` – The device is on a user who is running.
- `WALKING` – The device is on a user who is walking.
- `UNKNOWN` – Unable to detect the current activity.
- `TILTING` – The angle of the device relative to gravity has changed significantly.
- `STILL` – The device is still (not moving).

Bellrock processes the obtained activities further, more detail is given in Section 3.1.3.

[1] https://developers.google.com/android/reference/com/google/android/gms/location/DetectedActivity



Progress of the Bell Rock Works, engraving by William Miller. [2]

Chapter 3

The Bellrock system

This chapter describes the Bellrock system in detail. In particular, the design of the client and the server are discussed.

3.1 Bellrock client

The client is written in Java and it is designed to run as a background application on Android phones. The screenshot below shows the GUI of the Bellrock client.

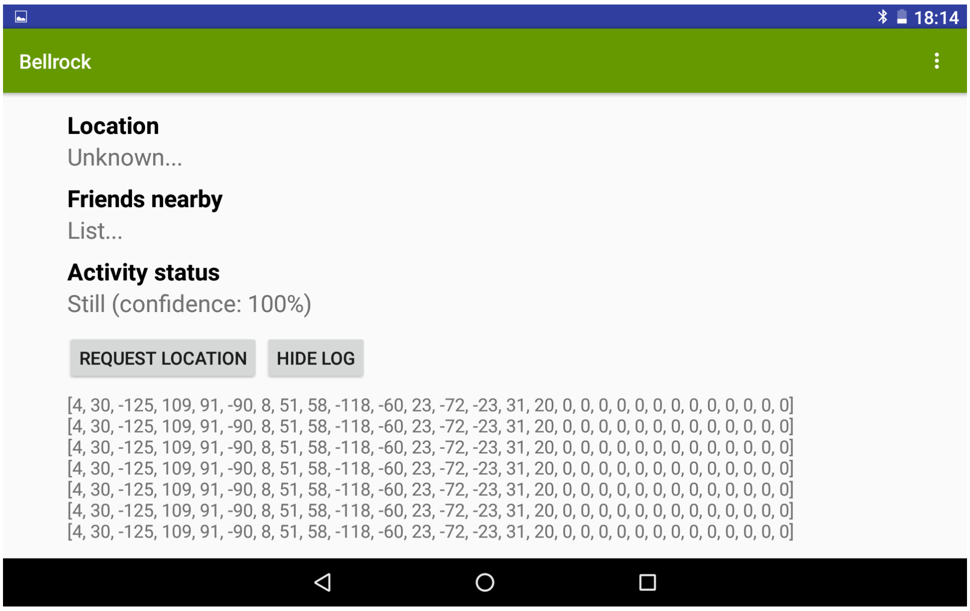


Figure 3.1: The GUI of the Bellrock client. The log in the lower part of the screen shows a list of the observed 16-byte AIDs.

The diagram below provides an overview of the main components of the client.

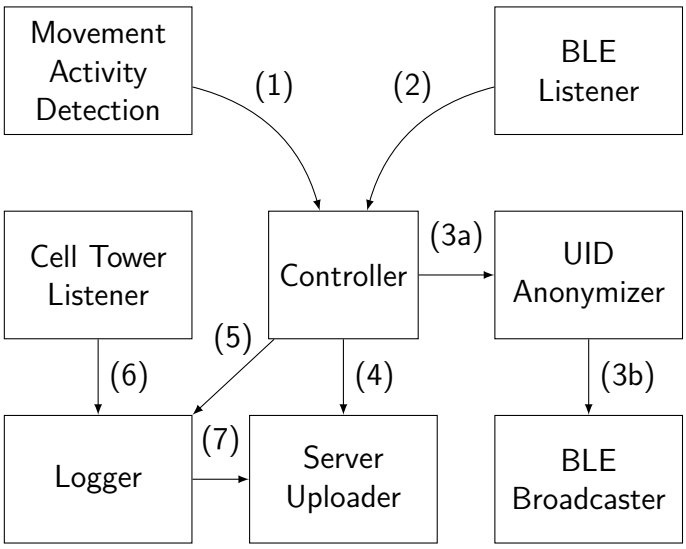


Figure 3.2: Diagram of client components

3.1.1 UID Anonymizer

The UID Anonymizer takes the Unique User ID (UUID) and produces the Anonymous ID (AID). This AID is then broadcasted instead of the UUID to prevent user tracking. The AID is produced by encrypting the 8-byte UUID together with an 8-byte random nonce. This makes the AID look random from a random observer's point of view, but Bellrock is still able to resolve it.

Since the encryption and decryption of AIDs is rather involved, a rigorous explanation and a discussion of the scheme can be found in Chapter 4.

3.1.2 BLE Broadcaster

The BLE Broadcaster broadcasts periodically the given AID using a BLE advertising packet of the following format:

Len	**Type**	**GAP**		**AID**															
1	2	3	4	5															20

Table 3.1: The Bellrock AID packet format with byte numbers in the second row

The one-byte *Len* field contains the length of the packet (20 bytes). The two-byte *Type* field contains the information about the packet payload required by the BLE advertisement packet specification. The *Type* is set to `DATA`. The *GAP* contains the unique manufacturer ID, which is set to `0x0000` for the purposes of this project. If the project were to become public, a manufacturer ID would have to be registered. The remaining 16 bytes contain the AID. Since BLE packets can be up 31 bytes long, there are 11 bytes free for possible further enhancements.

Since the BLE broadcast mode is not yet supported on the majority of Android devices, this component is optional, making Bellrock operate in passive mode on such devices. In such cases the device only listens for BLE packets and does not broadcast.

The BLE Broadcaster can be turned on and off according to the device's moving/stationary status obtained from the Android Movement API which notifies the controller (1). The controller then decides whether the broadcast should be turned on or off. If the broadcast is turned on, the controller commands the UID Anonymizer (3a) to produce an AID which is then broadcasted (3b).

3.1.3 Movement Activity Detection

Bellrock is registered as a listener of the Android Movement API, which notifies it about the movement activity changes. Each notification consists of a list of activities with a probability assigned to each of them.

Bellrock takes the activity with the largest probability and checks if it is `STILL`, `TILTING` or `UNKNOWN`. These states are considered to be the *stationary* (non-moving) states. All the remaining states are classified as *moving* states. Bellrock keeps a window of the last 3 detected activities. If most of them (i.e. at least two) are *moving* activities, Bellrock stops broadcasting until most of the activities are *stationary* activities again.

This is done to smooth out the local errors in activity detection, as illustrated in Figure 3.3. The window size of 3 proved to be optimal empirically, as 5 or greater caused a behaviour that was too "stiff".

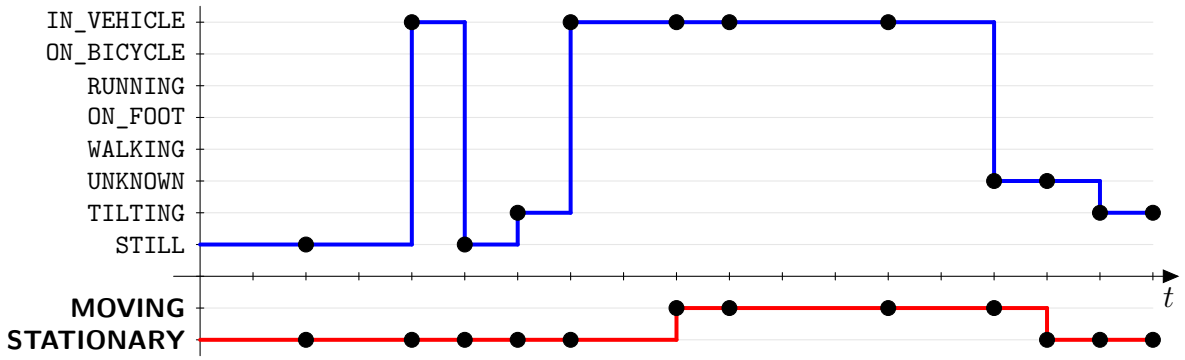


Figure 3.3: The blue line denotes the most probable observed state, the red line the state determined from the window of size 3. Black points are the measurements.

The Movement Activity Detection component notifies the Controller about the status of the device and the Controller controls the BLE broadcast accordingly.

3.1.4 BLE Listener

The BLE Listener listens for BLE advertisement packets sent by other Bellrock devices. Upon capture, the packet is parsed and the AID received in the packet is stored in a log. That is, the BLE Listener notifies the Controller (2) about the new packet and the Controller uses the Logger (5) to save the observation.

3.1.5 Logger

The Logger logs two things: Firstly, it logs the observations of AIDs received from the BLE Listener. The time of observation is stored together with the AID. Table 3.2 shows an example of such an observation.

Field	Value
AID	`0xA0B3_11F2_0042_513F`
Time	2016-05-09T13:15:23+00:00

Table 3.2: An example of an AID observation

Secondly, cell IDs of cell towers the user was registered with are logged. A typical record for a user that came to the Computer Laboratory at 9 am and left at 5 pm would look like this:

Field	Value
MCC	234
MNC	20
LAC	37
CellID	10874448
Begin	2016-05-09T09:00:00+00:00
End	2016-05-09T17:00:00+00:00

Table 3.3: An example of a record in the log of user cell tower usage

As described in Section 2.3, the cell tower information is stored in a more compact packed format. Moreover, the beginning and ending times are stored in epoch format, resulting in a very compact representation. The example above would be hence stored as: `(4215721104882658896, 1462784400, 1462813200)`.

The information about the user's cell tower is used to obtain user's coarse location on the server side and that is used to make the AID decryption faster, as discussed in Section 4.3.

3.1.6 Server Uploader

If the user is connected to the internet using Wi-Fi connection, the data from the log are uploaded to the server. Once data are uploaded, the log is emptied. The data are uploaded using a standard encrypted stream, as they are potentially sensitive. Each client has a secret key given to it by the server upon registration so that it can prove its identity to the server. This is done to prevent attackers from pretending that they are a Bellrock user. This part of the client is not actually implemented as it is only a technical detail.

If the user is offline, the data are uploaded once the user goes online. This introduces unpredictable behaviour into the system, as on the server side it is not certain when user observations will be received. Section 3.2 explains in detail how the server deals with this.

3.2 Bellrock server

The server is also written in Java, since Java provides convenient out-of-box cryptography, networking and database support. The diagram below provides an overview of the main components of the Bellrock server.

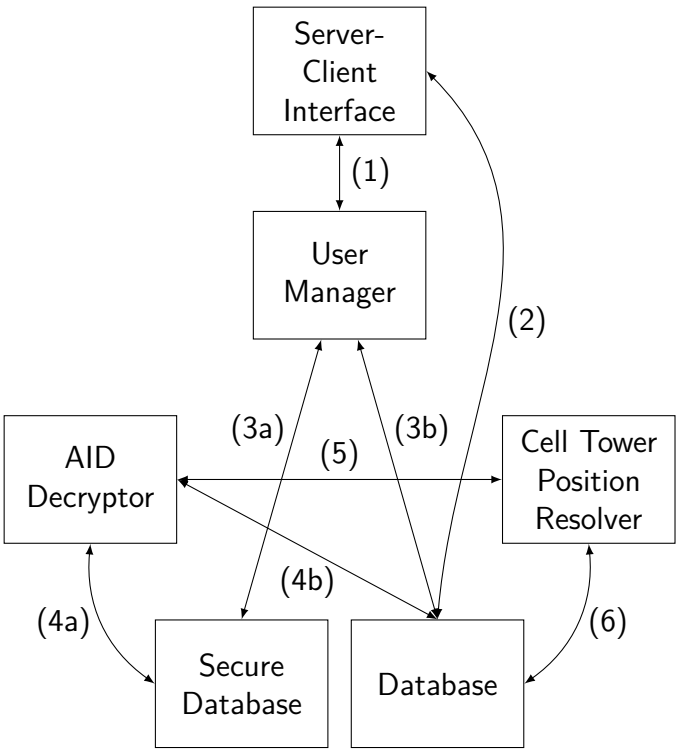


Figure 3.4: Server component diagram

3.2.1 Server-Client Interface

The Server-Client Interface serves only as a translator between abstraction layers – between the TCP/IP in which packets are transferred from and to clients and between the internal Java methods exposed by the server. This component takes care of the client authentication and then processes the client requests. A typical use case is uploading the client observations and sending back the information inferred by the server.

3.2.2 User Manager

Each time a new client is created, it needs to get a Unique User ID and also it's Secret Client Key (SCK) which is *known* to the server. The User Manager takes care of these things as well as taking care of SCK renewal. User Manager also takes cares of peers – pairs of users that want to be notified when their friend is nearby.

3.2.3 Databases

The Bellrock server has 2 HSQLDB databases: One unencrypted that stores data such as lists of peers, AID observations and user positions and one encrypted, which stores (UUID, SCK) pairs. This is a compromise between performance and security, based on the assumption that the unencrypted database is protected enough by the server hosting infrastructure. The secure database is encrypted using a 128-bit AES key.

However, the system is created so that turning the encryption on in the unencrypted database is just a matter of encrypting the existing database and changing a single line of code.

Unencrypted database

The unencrypted database contains the following tables:

Table	Attributes
Uid	`Uid`
Peers	`Uid, Peer`
Observations	`Uid, Aid, DecryptedUid, Timestamp, Lat, Lon, PosStr`
Locations	`Uid, StartTime, EndTime, Lat, Lon, CellTowerInfo`

Table 3.4: Tables in the unencrypted database

The Uid table contains UUIDs of all Bellrock users. The Peers table contains all peers, i.e. all pairs of users that want to be notified by the system if they get into a proximity of their friend. The Observations table contains the AID observations made by users. The field `DecryptedUid` is filled in by the AID Decryptor and the `Lat`, `Lon` and `PosStr` fields are filled in using the Cell Tower Position Resolver.

Encrypted database

The encrypted database contains the following table:

Table	Attributes
KeyStore	`Uid, Key`

Table 3.5: Tables in the encrypted database

The KeyStore table contains pairs of (UUID, CSK). The client key is used to produce an AID from an UUID, details of which are discussed in Chapter 4.

3.2.4 AID Decryptor

The AID Decryptor is used when a client uploads its observations to resolve each of the observed AID into a UUID. If this is not possible at the upload time due to lack of information, the user's observation is saved in a database and the database is later processed and the system attempts to resolve all the unresolved AIDs. The entire Chapter 4 deals with details of the AID decryption.

3.2.5 Cell Tower Position Resolver

The Cell Tower Position Resolver uses the local version of the OpenCellID database to make look-ups faster. The OpenCellID database was downloaded and converted from an CSV format to a Java `HashMap` which was serialized to the disk. The module, once started, loads the `HashMap` (which has about 200 MB) and then provides fast (< 0.5 ms) in-memory look-ups. This is used to aid the AID decryption: The coarse location of Bellrock clients is used as a spatial heuristic decreasing the number of potential Bellrock user's whose UUID could have been encrypted as that AID.

The approximate location of the client is obtained by rounding the location of the cell tower the user is registered with (latitude, longitude in radians) to two decimal places, i.e. the precision is reduced to approximately ± 0.5 km^2 as shown in Figure 3.5.

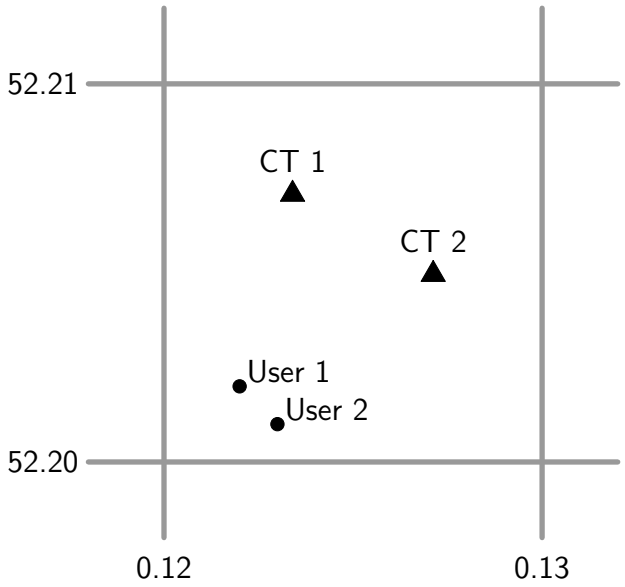
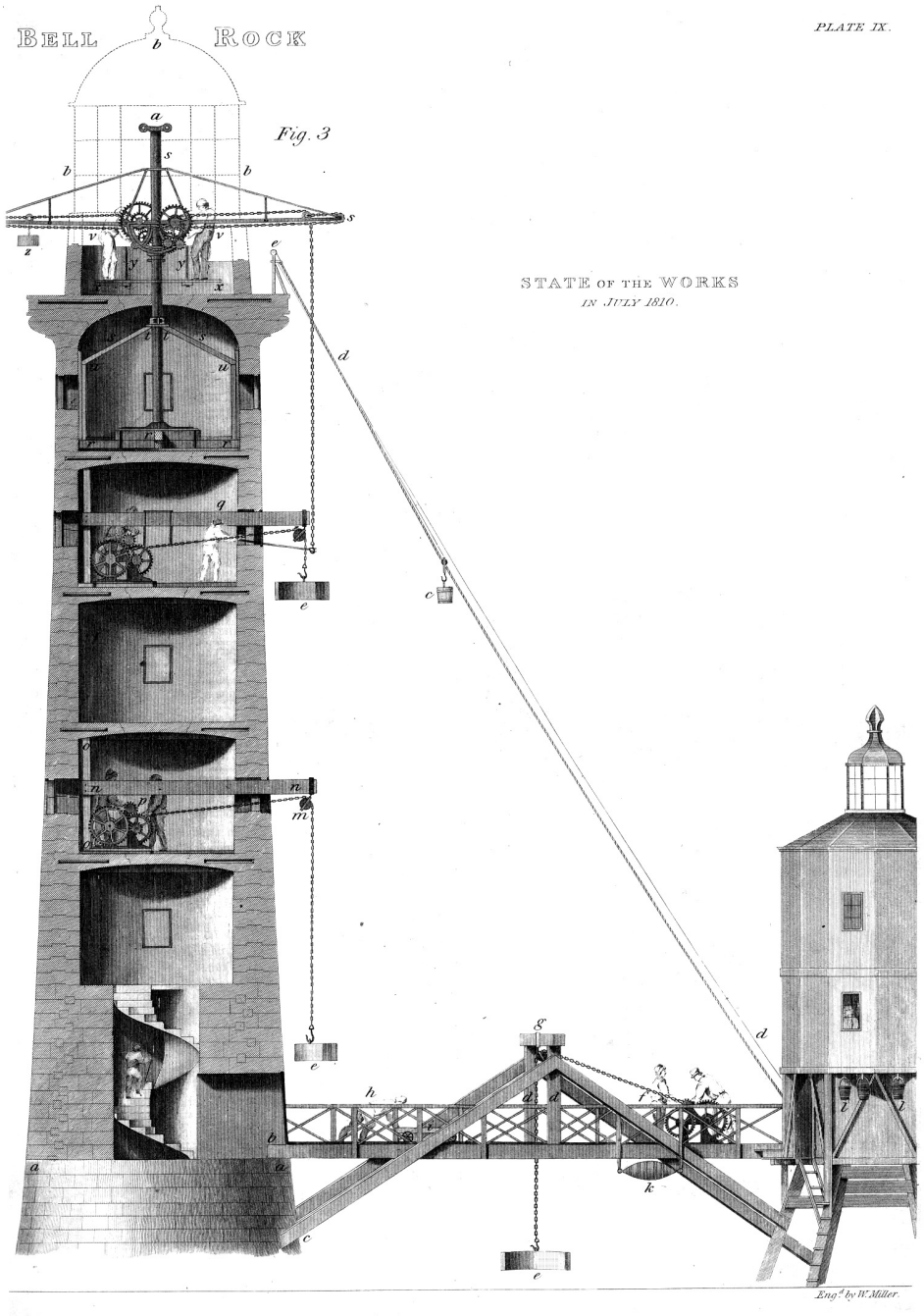


Figure 3.5: User 1 is registered with Cell Tower 1 (CT 1) and User 2 is registered with Cell Tower 2 (CT 2), so the positions of their cell towers differs. However, they are next to each other. By rounding the cell tower positions to two decimal places, their position is resolved within the same square (shown in grey).

This scheme compensates for the fact that the clients are not positioned at the cell tower and also that multiple users in the same physical location could be using different cell towers.



Progress of the Bell Rock Works, engraving by William Miller. [2]

Chapter 4

Achieving anonymity

Static beacons transmit the same static ID periodically. This is not possible with dynamic (moving) beacons, as users of such system would be easily trackable using a grid of beacon listeners. It is therefore essential to change user's IDs as they move around.

An initial idea would be to stop broadcasting when the user is moving and resume broadcasting when the user becomes stationary again. Although this hides *how* the user got somewhere, it does not prevent the attacker from seeing where the user actually got to.

A second improvement would be to start broadcasting a new random UUID after movement. This would be feasible only with the assumption that all users can ask the server for a new UUID each time this happens. Although in many cases this would not be a problem, it would not be feasible for other technical reasons:

1. **Power.** Connecting to the server via internet is power-expensive. Each UUID renewal would therefore imply a battery drain, hence the benefit of power efficiency of BLE would be removed.
2. **Uniqueness.** If every user was to renew their UUID multiple times per day, the UUIDs would need to be longer to prevent duplicates. It would also make the inference on the server side more complex, as UUIDs of the same user would need to be translated into some unique and consistent internal server representation.
3. **Online requirement.** Bellrock clients would need to be online in order to change their UUID. Every offline UUID change would require either owning some pre-generated UUIDs or using the old UUID. This either puts more complexity into the client in the first case, or compromises user privacy in the second case.

For the reasons above, it is better to generate a single UUID for every client. Then generate a nonce and encrypt the UUID padded with the nonce using server's public key. The resulting Anonymous ID (AID) would be broadcasted. When the user moves, broadcast would stop and when the user becomes stationary again, a new random nonce would be generated and the UUID padded with this new nonce would be broadcasted as the new AID.

The main advantage of this scheme is that while the AID seems random to a random observer, the server can immediately tell the UUID of the sender once it decrypts the AID. Moreover, the clients can change the AID without contacting the server. While this scheme seems ideal, there are multiple disadvantages which are described in the next section.

4.1 Reasons for not choosing asymmetric cryptography

Bellrock uses symmetric key cryptography for AID en/decryption. When designing Bellrock, there were three decisions against using asymmetric cryptography:

Block length and BLE limitations. Asymmetric cryptography has a fundamental property that the *key length* and the *block length* are the same. Moreover, RSA requires at least 256-byte keys and, for the same level of security, ECC requires at least 32-byte keys [16]. This is the lower bound as of 2016 and it is expected that with the increasing computing power this requirement will grow. As discussed, BLE advertisement packet can accommodate up to 27 bytes of payload, hence it is not feasible to fit the AID encrypted using asymmetric cryptography into a single packet. Since having more than one advertising packets is undesirable due to power and storage considerations, this property makes the asymmetric cryptography undesirable.

Single point of failure. Assume the server private key is compromised, e.g. by using a huge number of collected AIDs. In such situation, all AIDs in the system are compromised and the security of the system is breached. This could be partially mitigated by periodically changing the server's public/private key pair, but distribution to clients would again increase the demand on clients being online. Moreover, on each key change, each AID would need to be decrypted using all historic private keys of the server as there would be no guarantee on what public key was used to generate the AID.

No breach feedback. This problem is loosely coupled with the previous one. Bellrock involves broadcast and the main feature of broadcast is the lack of feedback or privacy. As something is broadcasted, anyone can listen to it and the system has no way of finding out who is doing so. Therefore, if the server's private key gets compromised, the attacker can now decrypt all AIDs and the Bellrock system has no way of detecting that.

4.2 Symmetric key cryptography

This section describes how Bellrock clients use symmetric key cryptography to obtain an AID from their UUID. Then the decryption on the server side is described together with the heuristics it uses to speed up the AID decryption.

4.2.1 Encryption on the client side

Each device i obtains a secret key K_i from the server when first registered. The AID is generated from the client's UUID_i (8-byte) in the following way:

$$\text{AID}_{i,j} = \text{Enc}_{K_i}(\text{Concat}(\text{UUID}_i, n_{i,j})).$$

An 8-byte nonce $n_{i,j}$ is used until the device starts moving. Then the broadcast of this $\text{AID}_{i,j}$ is stopped and when device becomes stationary, a new random nonce $n_{i,j+1}$ is generated and therefore new $\text{AID}_{i,j+1}$.

AES with 128-bit key K_i and 16-byte block in Electronic Code Book (ECB) mode is used as the encryption function Enc. ECB mode is secure in this context, since exactly one block of the data is encrypted at a time.

We should emphasize that thanks to this scheme, the AID changes happen offline, that is without any assistance needed from the server. This is very valuable property, as the privacy of the Bellrock clients is not compromised if they are not online (which is often the case).

4.2.2 Decryption on the server side

Since symmetric key cryptography is used, the server has to use brute force approach to decrypt the AIDs. The AIDs can be decrypted since the server maintains a list of tuples (UUID_i, K_i). The initial naïve implementation looks like this:

Algorithm 1 Naïve AID decryption

```
1:  procedure DecryptAID(AID)
2:      for (K_i, UUID_i) in KeyUUIDMap do
3:          DecryptedPacket ← Dec_{K_i}(AID)
4:          DecryptedUUID ← DecryptedPacket.Bytes(0, 8)        ▷ Get rid of the nonce
5:          if DecryptedUUID = UUID_i then
6:              return UUID_i
7:          end if
8:      end for
9:      return null
10: end procedure
```

While this approach successfully decrypts the AIDs, it scales poorly, as its time complexity is linearly dependent on the number of users in the system. The following section describes techniques employed in the system that mitigate this problem, making the Bellrock approach feasible with an arbitrary number of users.

4.3 AID decryption heuristics

To mitigate the curse of the brute-force AID decryption, i.e. the linear relationship between the AID decryption time and the number of users, Bellrock employs spatial and temporal heuristics. These heuristics turn the AID decryption amortised time complexity to constant.

4.3.1 Expected device heuristic

A last-recently-seen cache for each Bellrock user keeps UUIDs of other Bellrock clients the client saw recently. It is quite probable that these UUIDs will be observed again, since throughout the day people tend to meet the same set of people (think about the people you meet usually during a day at the Computer Laboratory). Hence the keys of these users are tried first for AID decryption of a given user.

4.3.2 Spatial heuristic

As mentioned in Section 3.1.5, each Bellrock client uploads a list of cell towers it was connected to. Bellrock server resolves these into coarse user locations as explained in Section 3.2.5.

Keys of users at the same coarse location are tried first to decrypt the observed AID. Since the number of users per area unit is bounded, this reduces the decryption complexity essentially to amortized $O(1)$. It is amortized as there might be cases when:

- The observer's location is unknown. This might happen for instance when the user is not connected to any cell tower.
- Alice hears Bob who is not at the same coarse location. This happens on cell tower range boundaries or on the cell position rounding boundaries. Here, this heuristic will fail. However, this heuristic could be extended to search in neighbouring coarse locations.

4.3.3 The complete AID decryption algorithm

This section shows the complete decryption algorithm which includes both heuristics described above. Analysis of the impact of the heuristics on the performance is in the Evaluation chapter.

Algorithm 2 AID decryption with heuristics

```
1:  procedure DecryptAIDWithHeuristics(AID, ObserverUUID)
2:      DecryptedUUID ← DecryptAID(AID, LastRecentlySeen[ObserverUUID])
3:      if DecryptedUUID ≠ null then
4:          LastRecentlySeen[ObserverUUID].Add(DecryptedUUID)
5:          return DecryptedUUID
6:      end if
7:
8:      DecryptedUUID ← DecryptAID(AID, NearbyDevices[ObserverUUID])
9:      if DecryptedUUID ≠ null then return DecryptedUUID end if
10:
11:     DecryptedUUID ← DecryptAID(AID, FullKeyUUIDMap)
12:     if DecryptedUUID ≠ null then return DecryptedUUID end if
13:
14:     return null        ▷ The observed AID wasn't generated by a registered Bellrock user
15: end procedure
16:
17: procedure DecryptAID(AID, KeyUUIDMap)
18:     for (K_i, UUID_i) in KeyUUIDMap do
19:         DecryptedPacket ← Dec_{K_i}(AID)
20:         DecryptedUUID ← DecryptedPacket.Substring(0, 8)      ▷ Get rid of the nonce
21:         if DecryptedUUID = UUID_i then
22:             return UUID_i
23:         end if
24:     end for
25:     return null
26: end procedure
```

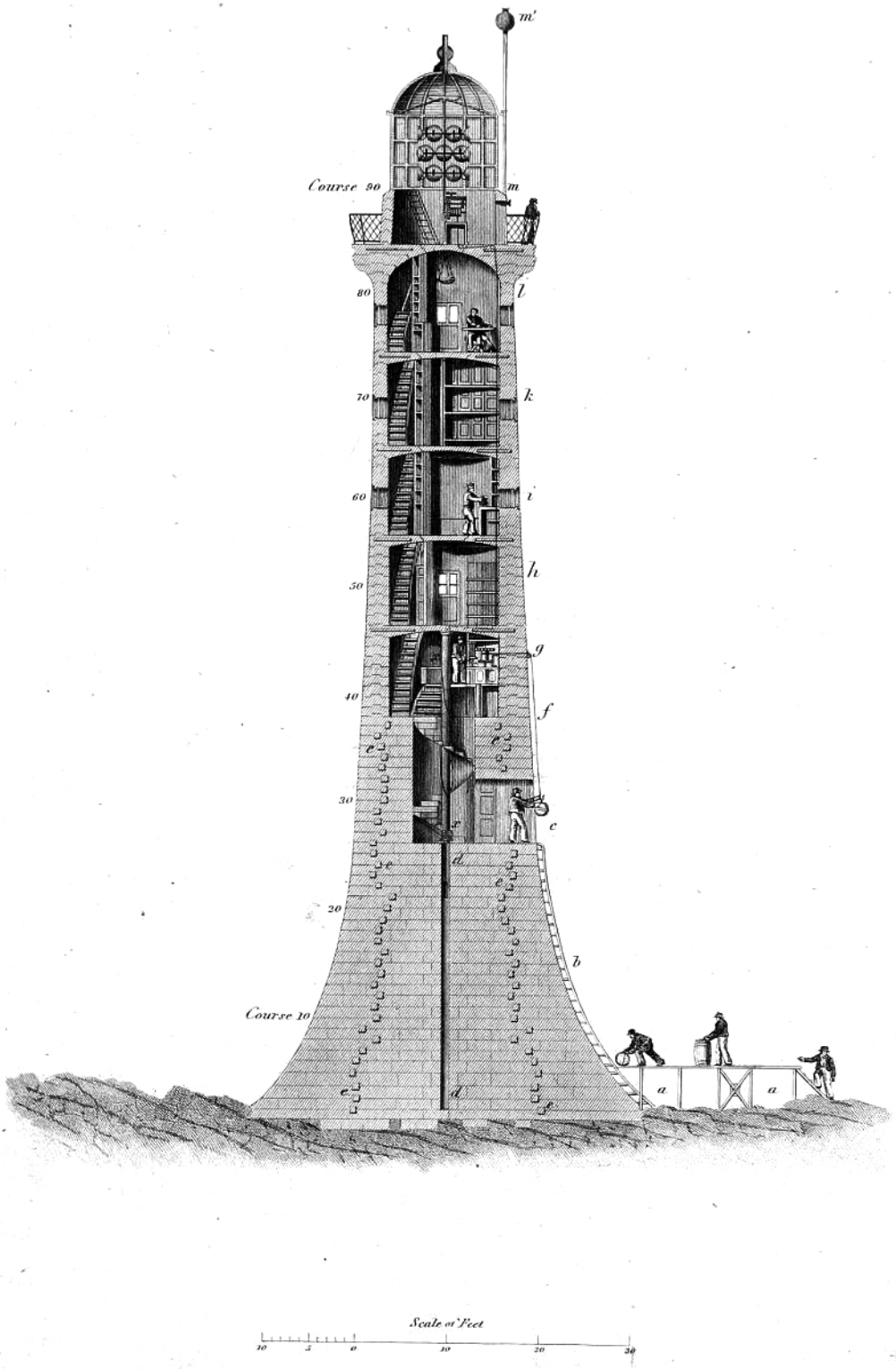
4.4 Server implementation optimisations

Although the heuristics above increase the speed rapidly (details in the Evaluation chapter), the server still needs to be able to perform a large number of AID decryptions very quickly. The following optimisations were incorporated into the algorithm:

1. **Using ECB[1] mode.** ECB mode is not considered secure in many applications, but here it is sufficient as only one block is decrypted at a time. Using the ECB mode made decryption significantly faster compared to e.g. CBC[2] or CTR[3] modes.
2. **Pre-loaded `Cipher` objects.** In Java, the `Cipher` object is used for AES encryption or decryption and one `Cipher` object is kept per Bellrock user. However, initialisation of this object takes significant time. To speed up repeated decryption, the first time a Bellrock user's `Cipher` object is needed for decryption, it is initialised and then it is kept in memory to make subsequent decryptions with that user's key faster. While this approach is safe as per Java documentation, the performance savings are substantial.
3. **Local OpenCellID database.** A local `HashMap` backed version of the OpenCellID database speeds up cell tower locations look-ups by more than two orders of magnitude.
4. **Fast LRU cache.** The last-recently-used (LRU) cache employed for caching recently seen users by a Bellrock client is backed by a `LinkedHashMap` implementation, achieving both fast look-ups and fast evicted item removal.
5. **Parallel decryption.** The decryption is highly parallelisable, as each decryption using one user's key is absolutely independent on the other decryptions. Java 8 parallel `Stream` is used to implement the parallel decryption which makes the Bellrock server automatically scalable on systems with more threads. Since the jobs of the threads are small (1 AID decryption) and there is not much blocking (only checking if a thread did not decrypt the AID successfully), the speed-up is almost linear with the number of threads.
6. **CPU-accelerated AES decryption.** Java 8 (7u40+) use AES-NI (Intel® Advanced Encryption Standard Instructions) which enable accelerated AES decryption [17]. The achieved speed-up was about 32%.

With all the optimisations put together, Bellrock achieves raw decryption speed of about **5.5 million AIDs per second** on reasonably modest hardware: Intel Core i5-2410M CPU, 2.3 GHz running with 1.5 GB of RAM allocated to the JVM. This means, that even *without any heuristics*, the Bellrock server (running on the hardware mentioned above) with 100,000 users is able to achieve average AID decryption time of 9 ms.

[1] Electronic Codebook
[2] Cipher Block Chaining
[3] Counter



Furbishing the Bell Rock Lighthouse. [2]

Chapter 5

Evaluation system

Evaluation of the Bellrock system was challenging, due to the aforementioned lack of devices that support BLE advertisement mode. It was therefore not possible to deploy Bellrock among volunteers and test it directly. Although the Bellrock client functionality was tested on testing devices (Nexus 7 and Nexus 5X), this approach was unable to produce enough data to fully evaluate the performance of the Bellrock server.

In order to fully evaluate the Bellrock system and to have a strong basis for assumptions about user behaviour which are incorporated in the Bellrock server heuristics, a simulation of the system was used.

The simulation had two parts: In the first part, real data of user movements was used to provide the initial estimates of user behaviour. In the second part, an environment with 10,000–100,000 users was simulated and the performance of the Bellrock server was tested on it.

This chapter deals with the details of the evaluation system details, so that the Evaluation chapter can be more focused.

5.1 Bat Logs

In 2002, Andy Harter, Andy Hopper and their team built the Cambridge Bat System [18, 19]. The Bat System was an indoor location system based on radio and ultrasound with features such as automatic door opening and user-aware areas which could trigger an action when a user entered them.

This system was employed at the AT&T Laboratories in Cambridge, UK and later also at the Computer Laboratory in Cambridge. Most importantly, from the perspective of this project, the movement of Bat System users was logged and these logs were used in this project to provide initial statistics about user behaviour.

The data in the logs was recorded between 2001-11-06 and 2002-03-25, i.e. there are 139 days of data. The logs consist of records in the format

`<timestamp>,<user>,<room>,<x>,<y>,<z>,<rotation>.`

The AT&T logs have about 21.5 million of such records, that is about 1.7 GB of data.

5.1.1 Pre-processing Bat Logs

The following steps were performed on the raw Bat Logs. All the pre-processing below reduced the Bat Log size to 1.1 GB.

Merge. The Bat Logs were obtained in multiple files, each with a timestamp as a filename and the times within the file relative to that timestamp. These files were merged into a single file with absolute times by adding the timestamp of the file to the relative time of every entry in that file.

Removal of irrelevant data. The room ID in the raw Bat Logs was a 16-character ID. This was reduced to a 4-character ID as that is sufficient. Moreover, the rotation field was removed as it is irrelevant for this evaluation.

Removal of beacons and anonymisation. The data contained observations of static beacons. The observations of these beacons were removed, as they would skew the statistics of the real users. Then, the usernames were anonymised, replacing each 16-character username with a random 2-character unique string and the random mapping was discarded.

Chronology check. The Bat Logs contained duplicates and non-chronologically ordered data. The duplicates were present due to bad alignment of subsequent files – for some reason, a file often contained multiple records from the beginning of the next file at its end. These were removed and in the final pass, the chronology of the entire processed log was checked. This proved crucial as the subsequent processing of the logs is dependent on the entries being sorted in chronological order.

5.1.2 Generation of other logs

The pre-processed Bat Logs were used to generate other logs which were then subjected to detailed analysis which later enabled easier and faster information extraction. The following logs were generated:

Active users log

For a given time interval length t, the active users log stores all users that were *active* during this interval, i.e. that there were at least c observations of the given user in the time interval.

User count log

For a given time interval length t, the user count log stores the number of users that were active during that interval. Generating this log is trivial, it just involves counting the number of users in each interval in the active users log.

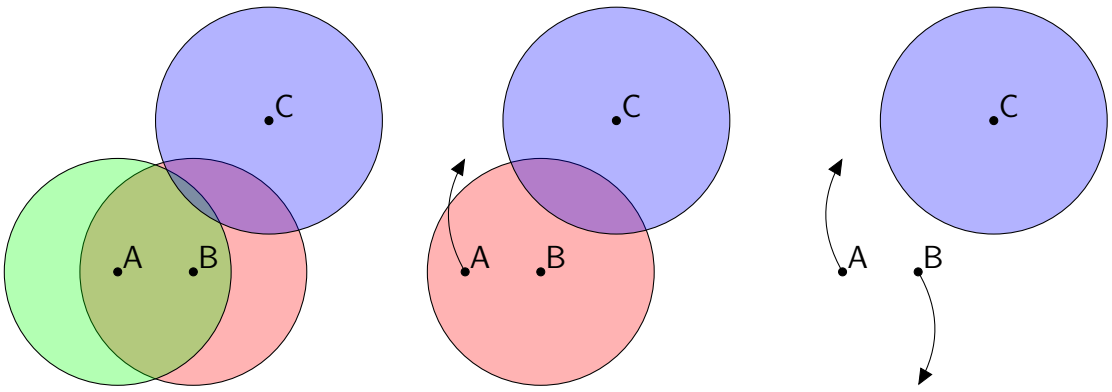
Encounters log

This log is generated by scanning the Bat Log and looking for pairs of users that, within a given time interval t, were within d meters from each other. When such event occurs, we consider the users to "meet" and an encounter is recorded in the encounters log. An encounter has a timestamp, the usernames of the two users that met and a position. Encounters are is symmetric – if Alice met Bob, then Bob met Alice at the same time and place. This log provides a *baseline* for the number of encounters each device had with other devices.

Beacon log

This log is similar to the encounters log, but simulates the beaconing strategy of Bellrock clients. That is, when Alice is stationary, she broadcasts her AID. When she starts moving, the broadcasting stops. When she stops moving again, she resumes the broadcast with a new AID.

The beacon log determines the movement speed of the Bat System users in 5 second intervals (to smooth out random position fluctuations in small time intervals) and stops the broadcast if the speed is greater than 0.5 m/s. The beacon log is therefore a subset of the encounters log and provides an accurate simulation of events that would happen if Bellrock was deployed on smartphones of the Bat System users. The difference between the encounters log and the beacon log is best illustrated with pictures of three different user meeting scenarios.



(a) A, B and C static (b) A moving, B and C static (c) A and B moving, C static

Figure 5.1: The difference between the encounters log and the beacon log. The coloured circles around devices A, B and C are the ranges of their BLE advertisement. Although C is static, it is too far from A and B so they cannot hear it.

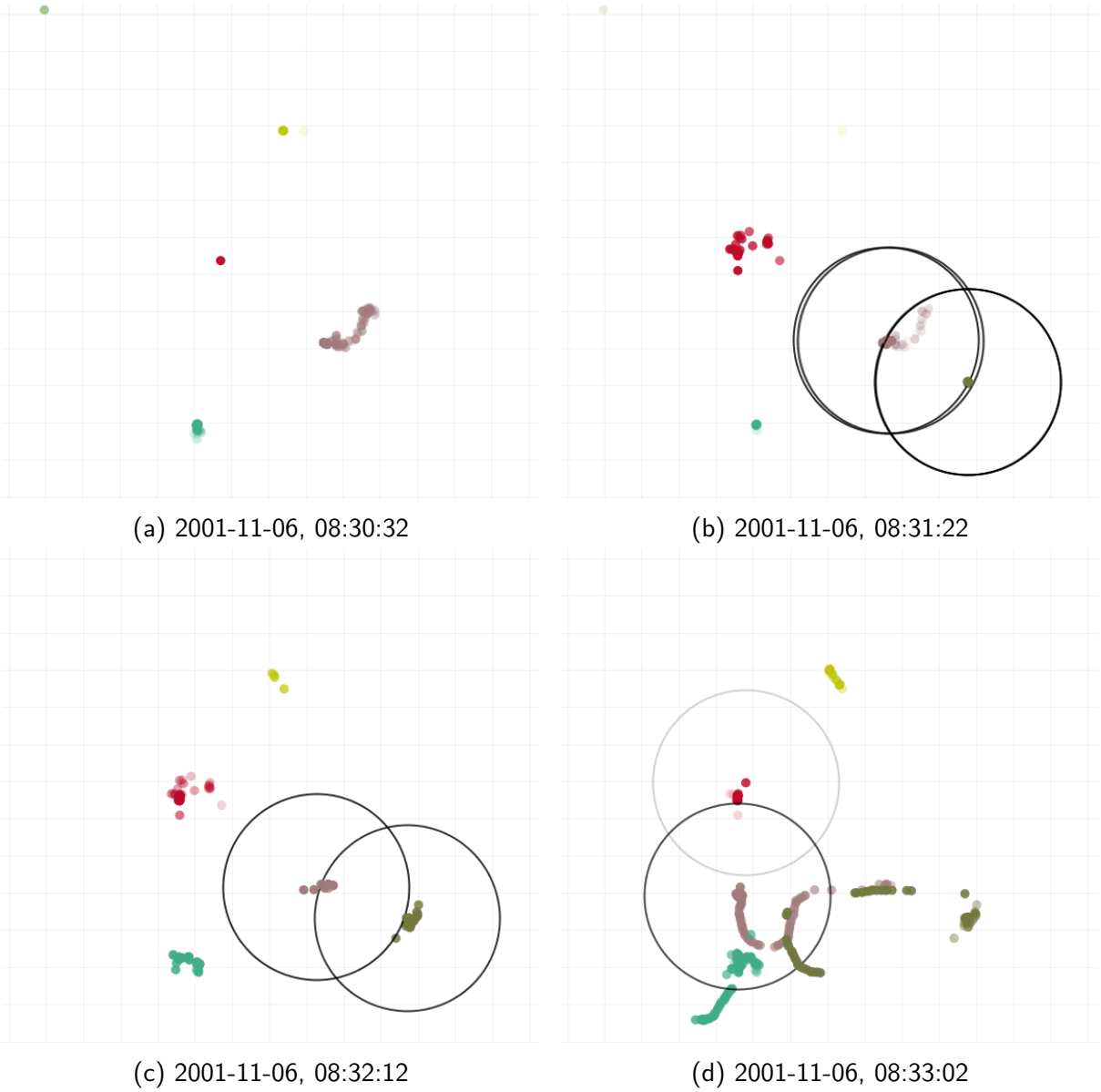
The table below shows the different events logged in the encounters and the beacon logs in the three cases shown in Figure 5.1:

Figure	Encounters log	Beacon log
Figure 5.1a	A observed B B observed A	A observed B B observed A
Figure 5.1b	A observed B B observed A	A observed B –
Figure 5.1c	A observed B B observed A	– –

5.1.3 Visualisation

It is hard to check the correctness of the log extraction algorithms by inspecting the logs with tens of millions of records. To make checking easier, I used Minuscule [20] – my own Java graphics library designed for rapid prototyping – to provide a visualisation of the Bat Logs.

The visualisation represents users as colour dots which move around and leave a trace that disappears with time. This shows how users move in time. Moreover, whenever there is an exchange of AID between two users, a circle with radius of the AID broadcast range is drawn around the user who sent their AID to the other user. The encounters also fade with time. The following four screenshots were taken at four consecutive moments 50 seconds apart.



(a) 2001-11-06, 08:30:32

(b) 2001-11-06, 08:31:22

(c) 2001-11-06, 08:32:12

(d) 2001-11-06, 08:33:02

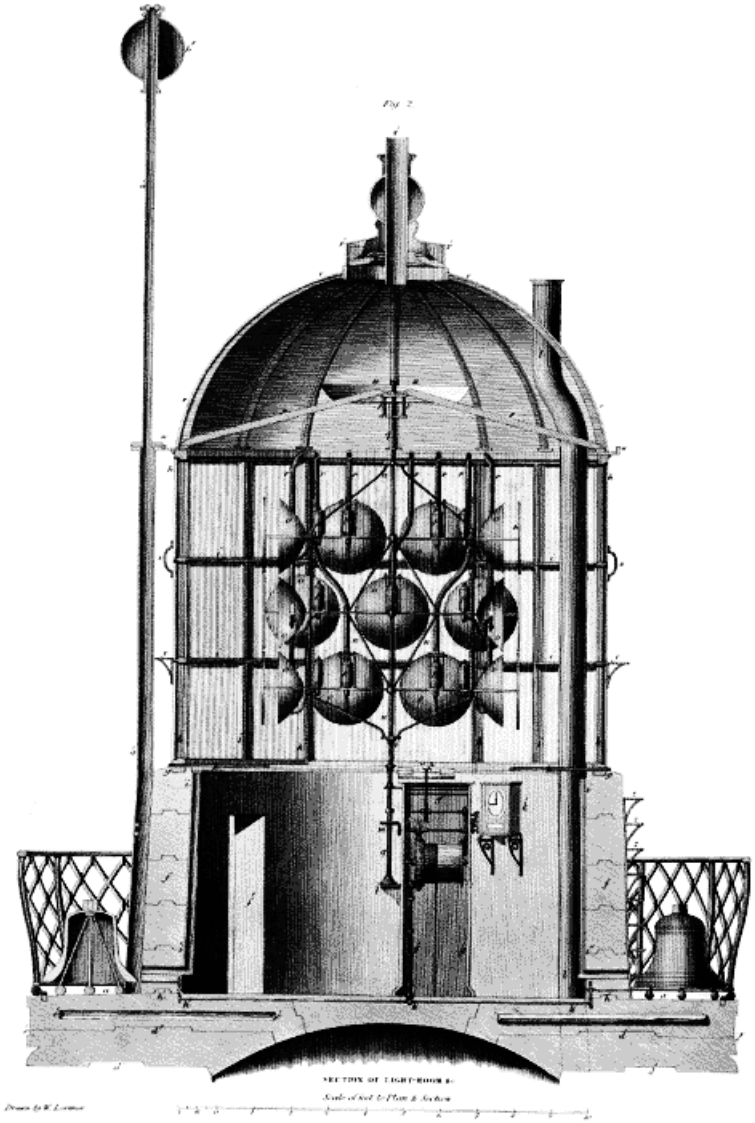
Figure 5.2: Visualisation of the Bat System log data for 4 time points separated by 50 seconds. Each colour dot represents one user and the dots fade with time so that user movement is visible. This is why there are multiple dots of the same colour with increasing transparency. The big black circles centred on the broadcasting user represent a successful broadcast of an AID. The broadcast circles also fade with time.

5.2 Simulation

A simulation was done to test the performance of the Bellrock server and to quantify the impact of the employed heuristics. The simulation consisted of the following steps:

1. A country is selected and all its cell towers are obtained from the OpenCellID database.
2. A random subset C of cell towers from the given country is generated.
3. A given number n of random users are generated and each user is assigned to a random cell tower.
4. At each cell tower, a random user is selected to be the *observer*.
5. At each cell tower, a random subset S of users is selected to be the broadcasting users at that cell tower.
6. At each cell tower, the observer makes r observations of AIDs of randomly selected users from S.
7. The Bellrock server is then used to resolve all the observed AIDs from all observers and the time taken to decrypt these AIDs is measured.

The time complexity without any heuristics is $O(n \cdot o \cdot r)$, where n is the number of users, o is the number of observers and r is the number of observations each observer made. With the heuristics, we show that the amortised time complexity becomes $O(o \cdot r)$, making the system performance *independent* of the number of users in the system!



Details of the Bell Rock Lighthouse lightroom. Note the light turning mechanism below the lamps and two warning bells on the sides. [2]

Chapter 6

Evaluation

This chapter begins with a feasibility study, where the behaviour of people is discussed and it is hence shown that Bellrock should at least theoretically work. The next part looks at the performance of the Bellrock server under the conditions assumed from the feasibility study. The penultimate part discusses possible attacks on the Bellrock system and counter measures that are either already present or that could be added into the system. The final part discusses the performance of the Bellrock client.

6.1 Feasibility study

The following subsections examine various user statistics in the Bat Logs and evaluate whether Bellrock's approach would be feasible in such conditions. The obtained statistics are valid only for office environment, a separate study would need to be conducted for other environments, such as shops, public transport or homes. It is expected that users in other office environments will have similar statistical behaviour.

6.1.1 Feasibility of purely dynamic beacons

The Bellrock system relies on the fact that for each client there are always at least a few other Bellrock clients nearby which this client can use for information inference. For instance, when a user comes to work, their Bellrock client needs to "hear" a few of their colleagues' smartphones in order to find our that they reached the workplace.

To verify this claim, the worst-case scenario with no static beacons (which could be used as anchors to provide a quite reliable source of information about the position) was examined to find out the proportion of time when a Bellrock client is sufficiently close to another Bellrock client to hear its broadcast.

The following histogram compares the number of near-by users to the number of active users in 5 second intervals using the encounters and the active users logs. This corresponds to the 5 second beaconing interval that is assumed in the following figures. The beaconing radius was assumed to be 7.5 m. While BLE range is higher, this is a conservative assumption that improves the validity of the presented results.

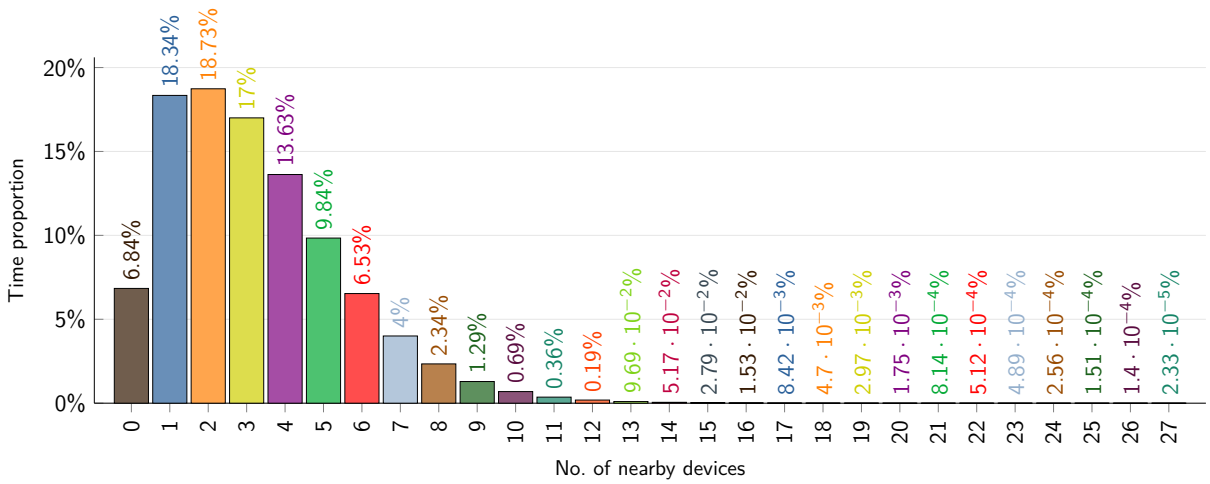


Figure 6.1: A histogram of the number of devices that were within the broadcast distance of a Bellrock client at any time.

Figure 6.1 shows that $100\% - 6.84\% = 93.16\%$ of the time, there was at least one nearby device for every Bellrock client. This shows that, in a workplace environment, using Bellrock would be feasible – i.e. there would be enough data for the system to do the analysis on.

Figure 6.1 takes only the client distance into account. However, Bellrock clients broadcast only when stationary and this needs to be considered, as it lowers the number of clients that are beaconing at a given time. Figure 6.2 shows the same kind of histogram as Figure 6.1 but with user movement – and corresponding pauses in beaconing – taken into account.

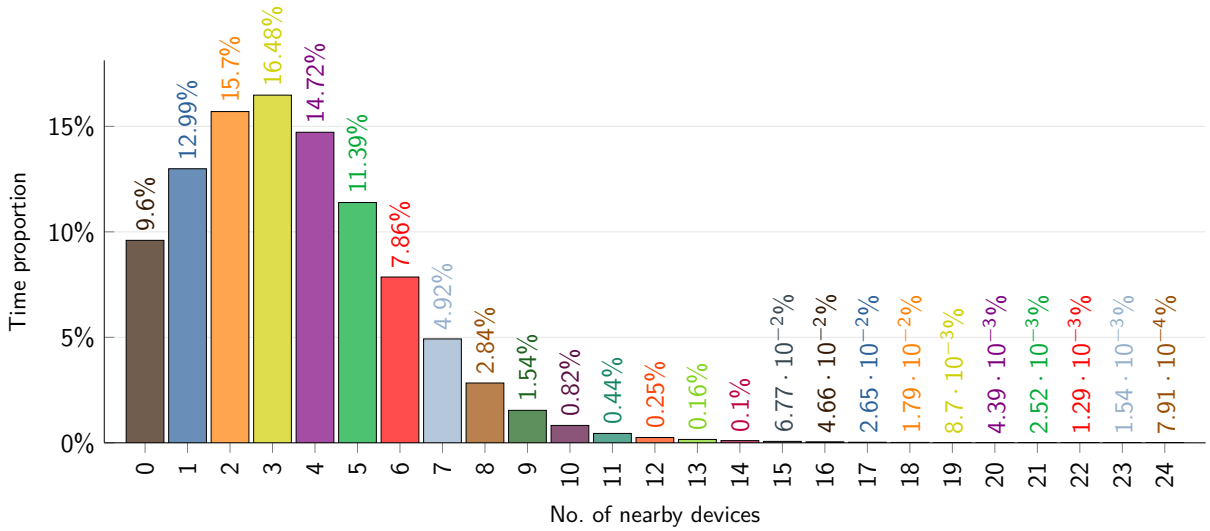


Figure 6.2: A histogram of the number of devices that were actually broadcasting within the broadcast distance of a Bellrock client at any time.

We see increase in the "0" bin, that is, in the proportion of time when a Bellrock client didn't hear any other Bellrock clients. This is understandable: take a case when a person went to the kitchen to get a cup of tea. In the first histogram, this user would be seen by users sitting in their rooms along the way. In this case, this user was practically invisible.

However, such a user would hear his colleague's devices when walking along the corridor, hence the user would essentially be able to infer the same kind of information as in the first case. The only case where information would be lost due to the privacy protection mechanics of Bellrock, would be in cases when both users are moving. In that case neither of them would beacon and hence the server would have no way of knowing that those two users met.

We also see decrease in the maximum number of Bellrock clients a user could hear at a given time from 27 to 24. However, at least one another Bellrock client was within range in $100\% - 9.6\% = 90.4\%$ cases which is still feasible for the purposes of Bellrock.

The obtained statistics are applicable only to a workplace-like environments. However, this is where most people spend 40+ hours out of a 168-hour week. The rest of the time is spent mostly at home, where similar smaller-scale behaviour would be expected.

Of course, there are many pathological situations when there would not be any other Bellrock beacons around, in particular outdoors. However, Bellrock is supposed to provide information mostly indoors, where we assume higher concentration of people.

6.1.2 Beaconing statistics

In the previous section, we looked at whether there would be enough other Bellrock clients around to listen to. In this section we look at the statistics of the beaconing itself.

Interesting property is the time that each device spends not beaconing due to movement. Figure 6.3 shows the proportion of time each Bat System user spent not beaconing. We see that even the most active user spent over 90% of the time beaconing.

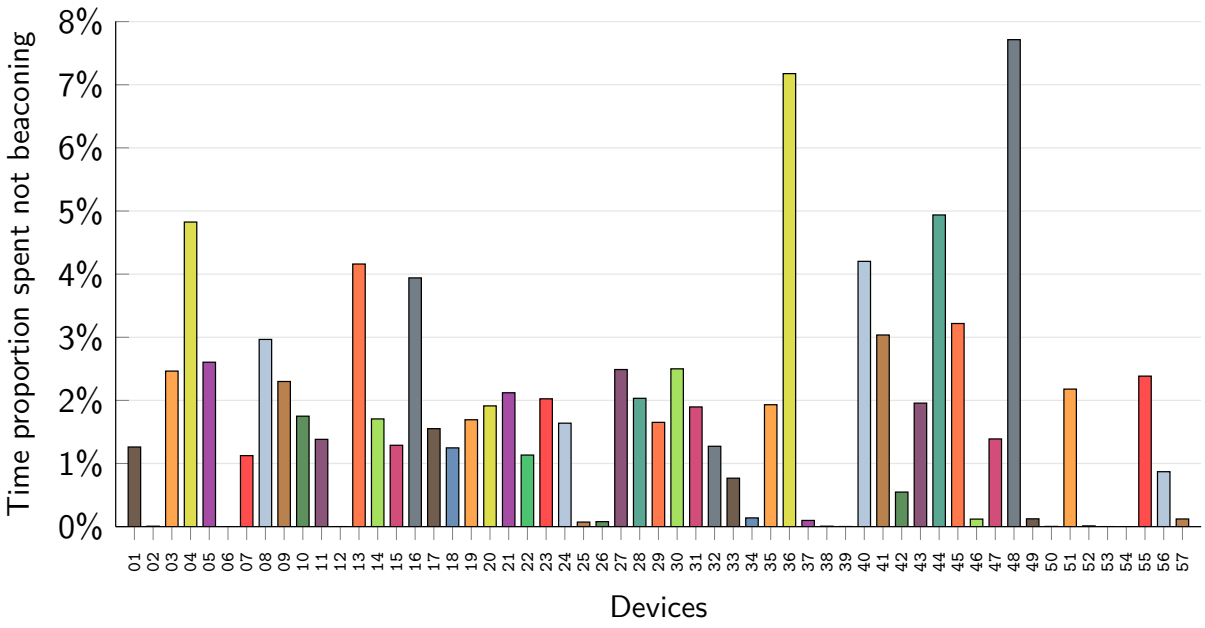


Figure 6.3: Proportion of time each user spent *not* beaconing while being active in the system.

The average proportion of time with no broadcasting is 1.8 ± 1.7 % and the median is 1.8%. This means that the limitations imposed by the Bellrock privacy-maintaining system are almost negligible. Bellrock therefore achieves user's non-trackability while loosing only a small proportion of potential measurements.

6.1.3 AID statistics

This section deals with the following questions concerning AID statistics:

1. How long do users keep broadcasting the same AID?
2. How many times do users need to change their AID per day?
3. How many AID observations do users make per day?

AID durations

Figure 6.4 shows that most AIDs are short lived, lasting 10–19 seconds. However, the distribution has a very heavy and significant tail: over 23.8% of AIDs live longer than 10 minutes. This corresponds to the assumed user behaviour – when walking around, there might be periods of time when beaconing is turned on for a short while until walking resumes again. However, once a user becomes stationary (e.g. at their desk), AIDs become long-lived.

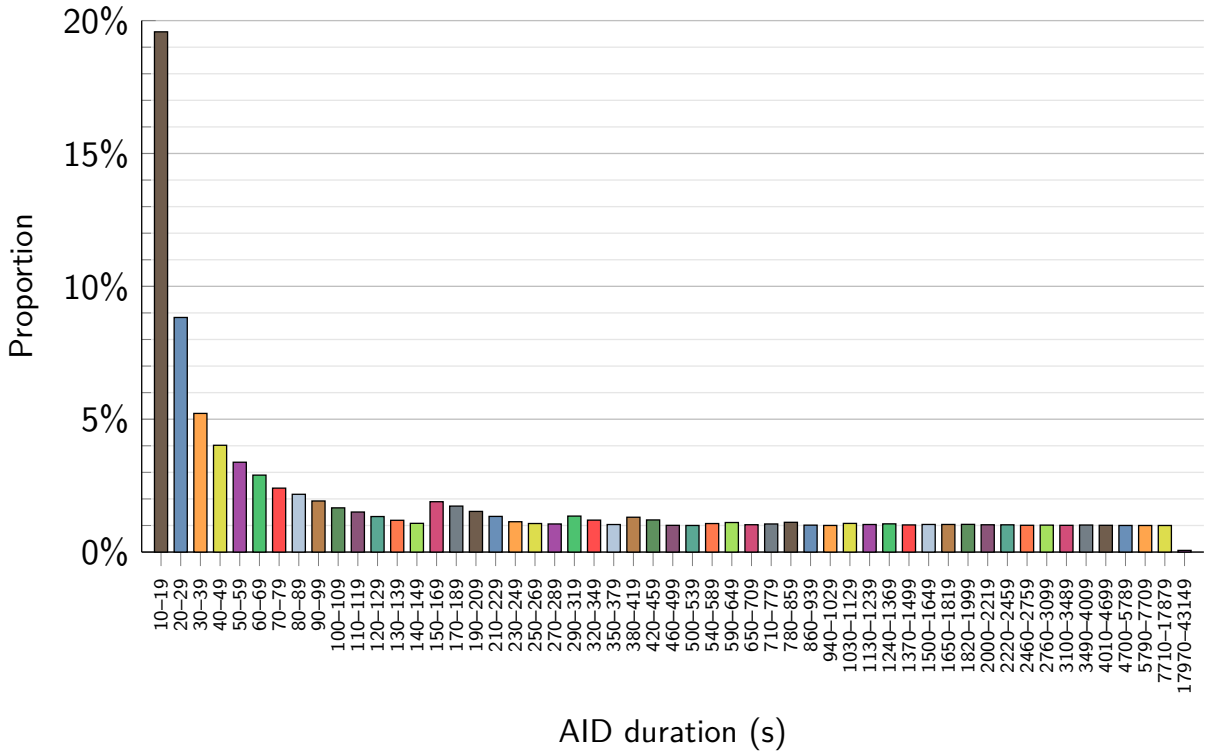


Figure 6.4: A histogram of AID durations. Note that the length of intervals is grouped so that the proportion is always at least 1%, with the exception of the last bin.

While it might seem alarming that a lot of the AIDs are very short lived, it does not prove to be a problem since this distribution has a very heavy tail. This is supported by Figure 6.5 which shows that the actual average number of AID changes per user per day is not large (less than 100). Since Figure 6.5 was generated using the same procedure as Figure 6.4, this is a valid claim.

The shape of the histogram in Figure 6.4 also makes sense from the following perspective: One can have many more short lived AIDs in a given time interval than long lived AIDs. For

instance, an one hour interval can be spanned by a single 1-hour AID or by 60 1-minute AIDs. For instance, it will be spanned by a 50-minute AID and 2 4-minute AIDs, as the user sat for 50 minutes and went twice to have a chat with their colleague, each time for 4 minutes. Such histogram will seemingly be dominated by short lived AIDs, but only due to the fact that more of such short-lived AIDs can fit into the given time interval.

AID changes

In relation with the previous subsection, an interesting statistic is the average number of AID changes per user per day. Note, that the number of days is the number of work days in the measured period, to prevent data skewing by weekends. Figure 6.5 reveals multiple interesting insights:

1. Some users clearly did not wear their Bats (i.e. did not participate in the collection of the data) or they were away during the experiment.
2. Some users made more observations than others. This is probably due to the fact they shared a small room with more people so they kept hearing each others AIDs.
3. The average number of AID changes per day is 23 ± 19. In reality this number will be higher, as this is only counted for 8 hours a day and it is skewed down by the people who did not participate in the measurements. The high standard deviation is caused most likely by different people behaviour and also by their varying availability. While some people spend most of their day in a single place, others socialise and move around the office, causing more AID changes.

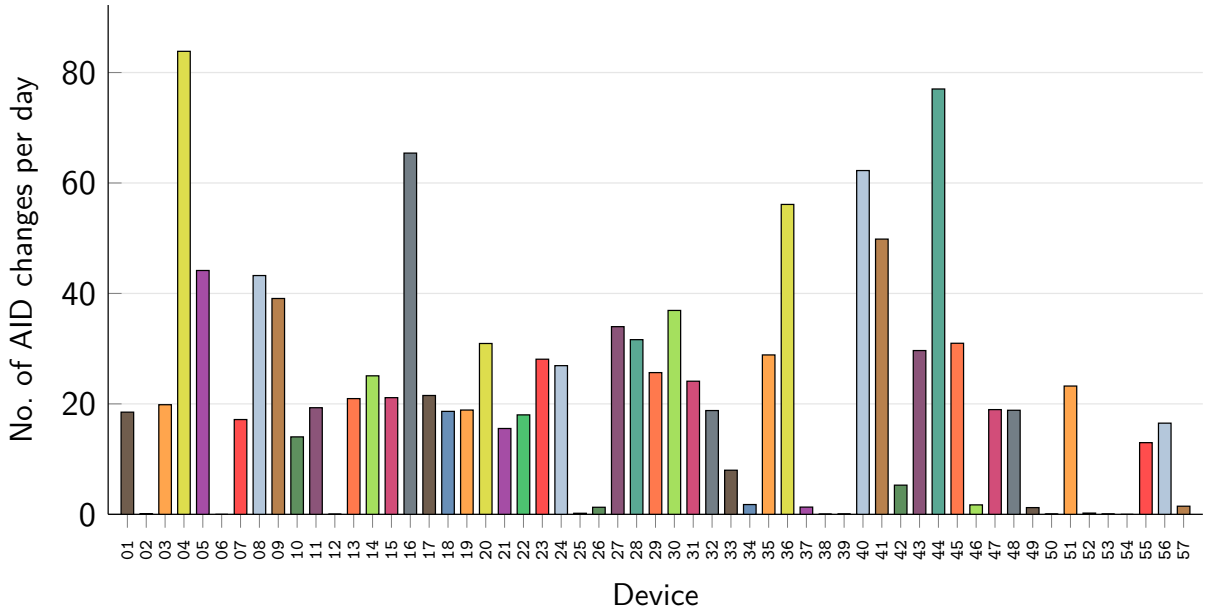


Figure 6.5: The average number of AID changes per user per day. This is averaged only over the working days in the observed period.

AID observations

The last statistic concerning AIDs extracted from the logs is the average number of AID observations each user made each day. Note, that the number of days is the number of work

days in the measured period, to prevent data skewing by weekends. Again, there were some interesting observations:

1. We observe the same two patterns as in Figure 6.5 – inactive users and large variance. The reasons are the same as in the previous case.

2. The average number of AID observations per user per day was 5316 ± 4614 with maximum average per day being around 20,000. Since the users were actively using the Bat System about 8 hours a day, this means about 1 AID observation every $(8 \cdot 3600)/20000 = 1.44$ seconds. This value assumes an AID broadcast every 5 seconds and a 7.5 m beaconing radius.

3. The maximum number of AID observations gives an upper bound on the amount of data a client would need to send the server. Each observation consists of an 16-byte AID and an 8-byte timestamp, that is 24 bytes per observation. The number of observations per day is bounded by 20,000 observations, hence at worst a user would need to send about $20000 \cdot 24$ B ≈ 500 kB. This is clearly not an issue in 2016 and it even leaves a generous margin – 5 MB or 50 MB per day are still acceptable.

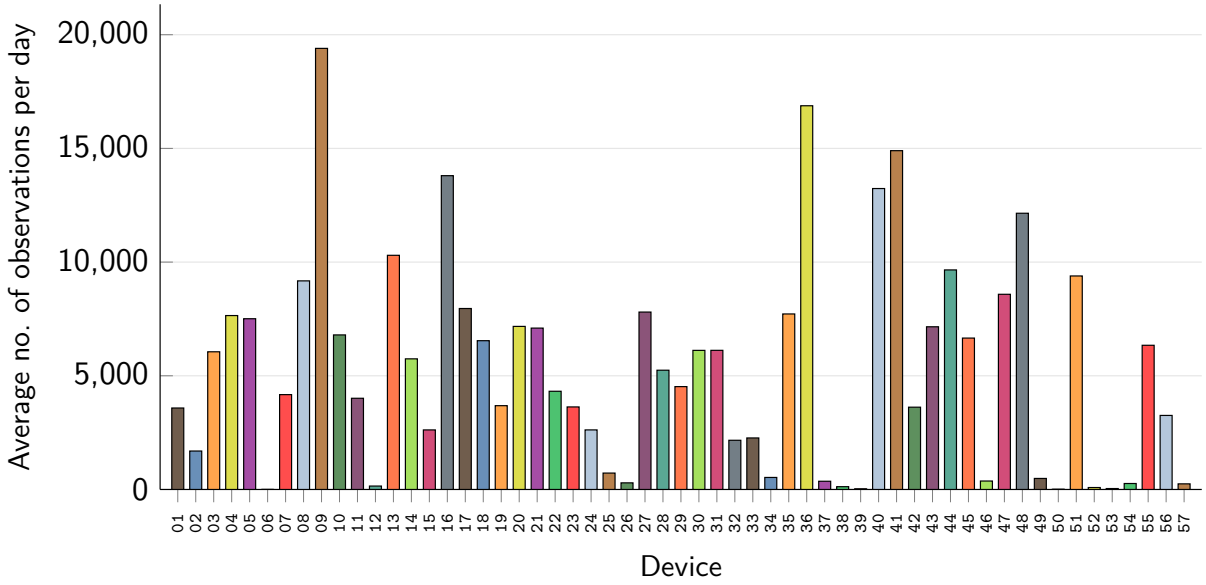


Figure 6.6: The average number of AID observations each device made per day.

User visibility

Figure 6.7 shows the availability of users in the Bat System. That is, the number of minutes each user was observed on average in the system per day. While this doesn't directly contribute towards the the server performance evaluation, it serves as a good check of the results presented above. From figures 6.5, 6.6 and 6.7 we can note correlations between user availability, the number of AID changes and the number of observed AIDs.

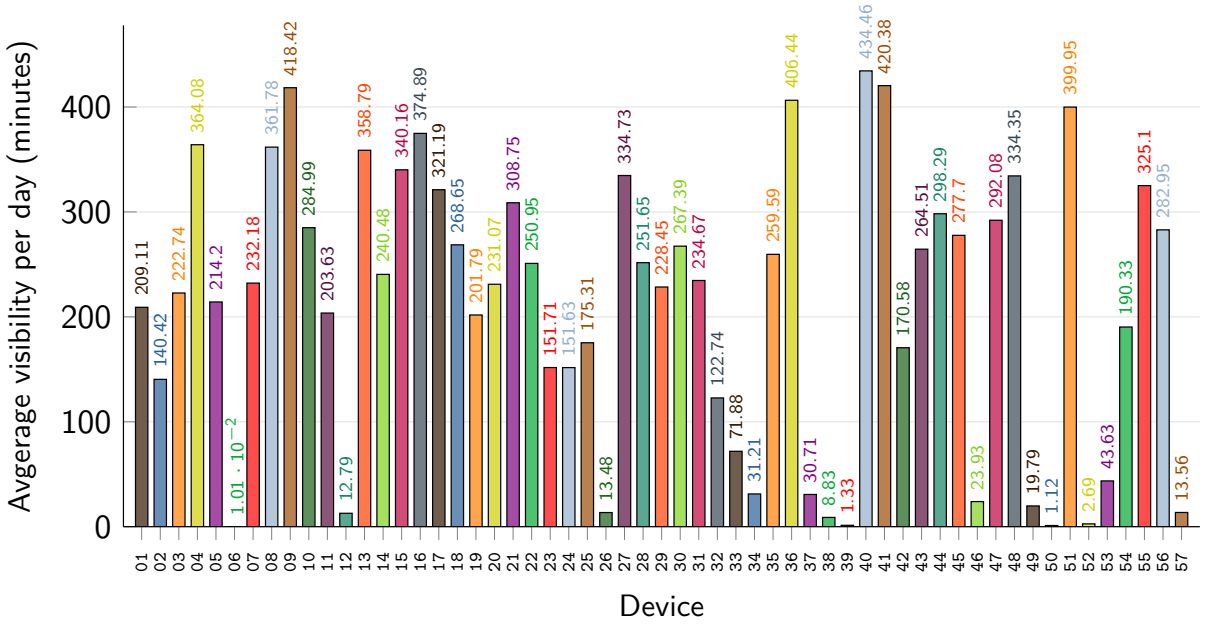


Figure 6.7: Average user visibility per day in minutes.

The average availability per user per day is 212 ± 132 minutes. This corresponds to 3.5 hours and while it might seem to small, it is mostly due to the small user sample size. If the 10 most inactive users are eliminated, the average increases to 270 ± 87 minutes per day.

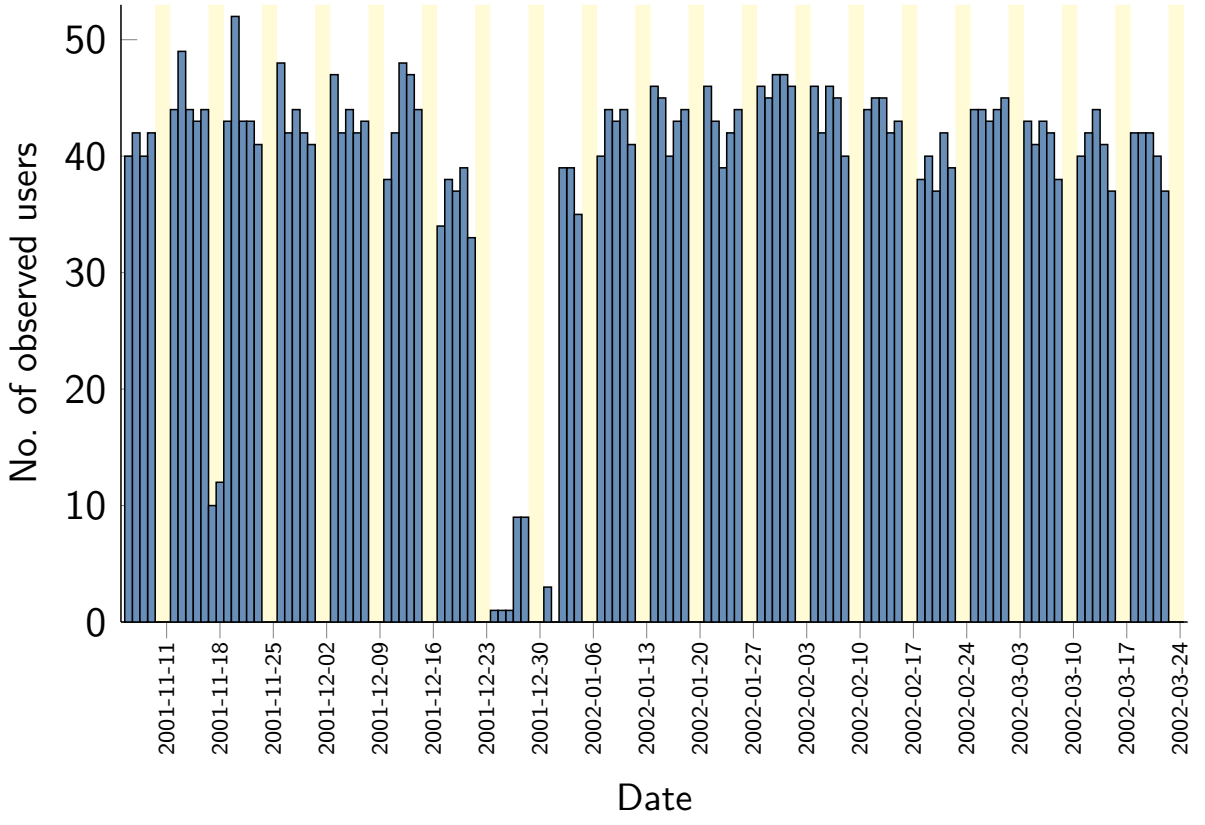


Figure 6.8: Number of active users on every day of the examined time interval.

Figure 6.8 shows the global picture: the number of users that were active (visible) every day during the observed period of time.

We can easily notice weekly patterns, as well as the Christmas Eve of 2001 by the number of active users in the system. During weekdays the number of users oscillates around 40 users which is consistent with the previous observations. This figure serves as a final check of the construction of the Bat Log analysis system and provides further insight into typical behaviour of people in office environments.

6.2 Server performance

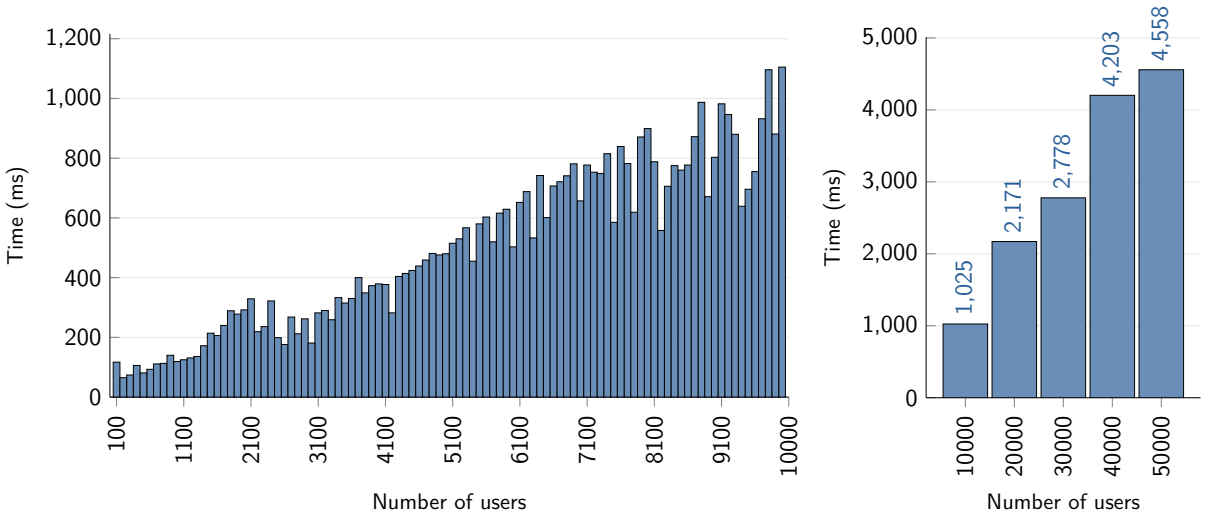
This section first examines the raw performance of the Bellrock server. Then, the impact of the heuristics is measured. All measurements were done on a laptop with Intel Core i5-2410M CPU running at 2.3 GHz with 3 GB of RAM allocated to the JVM.

6.2.1 Raw AID decryption performance

Before the impact of the heuristics is examined, it is essential to know the server's raw decryption performance. There are two factors that impact the decryption times: the total number of users on the Bellrock server and the number of AIDs the Bellrock server decrypts.

Impact of the number of users

Figure 6.9 shows the relationship between the number of users on the Bellrock server and the time that was needed to decrypt 1,000 AIDs that were obtained by randomly sampling from these users. I.e. a random user was selected to generate an AID 1,000 times. No heuristics are being used in this case. The server goes through the users one by one (using $t-1$ parallel threads where t is the number of available cores) until the given AID is successfully decrypted.



(a) No. of users 100–10,000

(b) No. of users 10,000–50,000

Figure 6.9: Time to decrypt 1,000 AIDs with number of users ranging from 100 to 10,000 (a) and 10,000 to 50,000 (b). The AIDs were sampled randomly from all users.

As anticipated, we see a linear relationship, as the expected number of users that need to be tried to decrypt the given AID is half the number of users on the server. This, however, causes the random fluctuations with growing variance as the number of users on the server increases. The peak between 1,200 and 2,000 users was most likely caused by Java garbage collection or an OS background task.

The random fluctuations are demonstrated in Figure 6.10 where the number of users and observations is fixed. Except for the first iteration where the server needed some extra warm-up time, the AID decryption time was constant with random fluctuations. These fluctuations are caused by the non-determinism of the AID brute-force decryption: if the server is "lucky", it hits the right user to decrypt the AID with earlier.

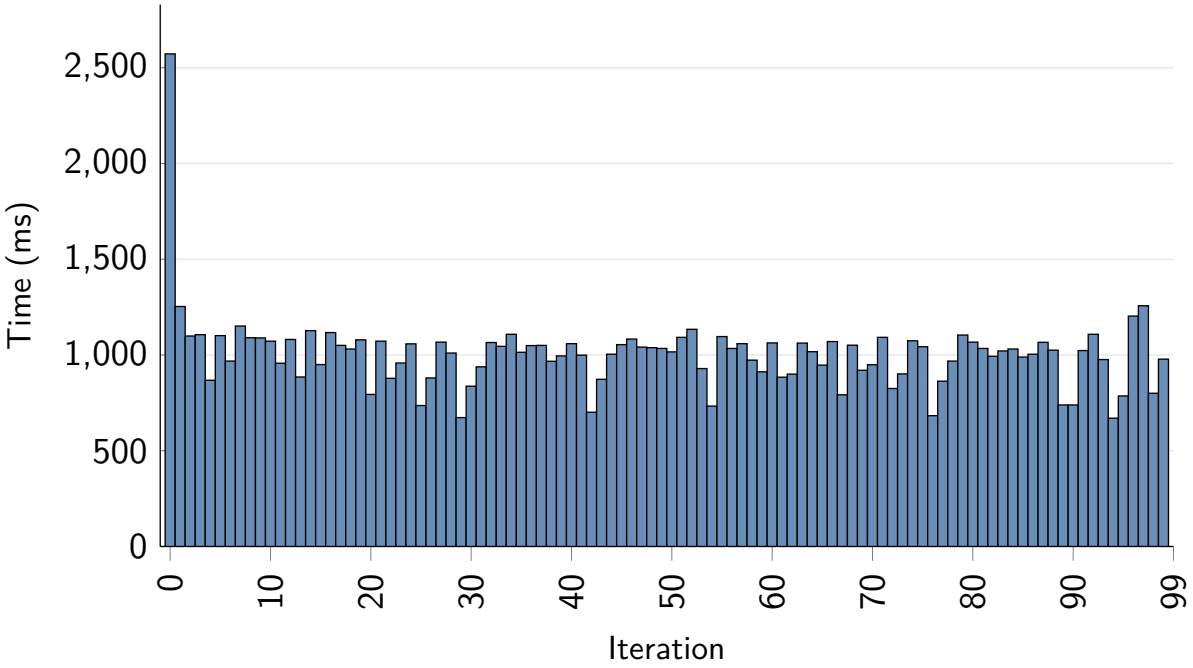
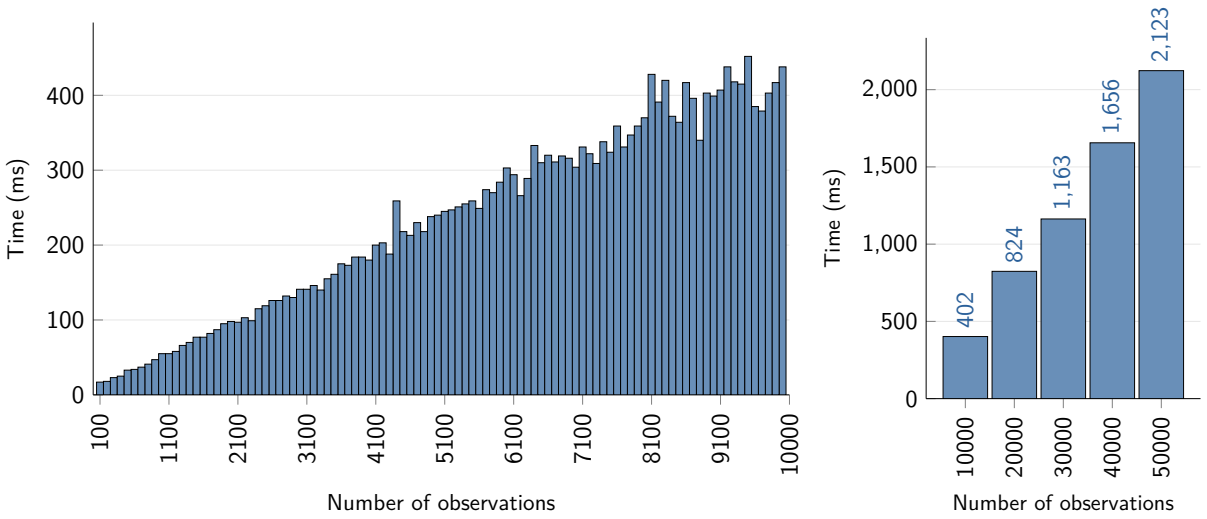


Figure 6.10: Time to decrypt 1,000 AIDs on server with 10,000 users.

Impact of the number of observations

This section examines the impact of the number of random AID observations on the performance. We observe a linear relationship with growing variance due to the same reasons as in the previous case. Again, the outlier at 4,400 observations was most likely caused by Java garbage collection.



(a) No. of observations 100–10,000 (b) No. of obs. 10,000–50,000

Figure 6.11: Time to decrypt the given number of AIDs on a server with 10,000 users. The AIDs were sampled randomly from all users.

Results

Using the data underlying Figure 6.10 (excluding the first iteration which apparently includes server warm-up time) and assuming that on average 5,000 users were tried before the AID was decrypted, we derive the average raw AID decryption speed. The average time to decrypt $5000 \cdot 1000 = 5 \cdot 10^6$ AIDs is

$$t_{avg} \approx \frac{(988 \pm 125)\ \text{ms}}{5 \cdot 10^6} \approx (198 \pm 25) \cdot 10^{-6}\ \text{ms/AID}. \tag{6.1}$$

Therefore, the raw AID decryption speed is

$$v_{avg} \approx 5051 \pm 638\ \text{AID/ms} \approx (5 \pm 0.6) \cdot 10^6\ \text{AID/s}, \tag{6.2}$$

on a i5-2410M CPU with 3 GB of RAM allocated to the JVM. Even on this modest hardware, the server achieves speed of about **5 million AID decryptions per second** which corresponds to about 76 MB/s as each AID is 16 bytes long.

Since the server uses Java 8 parallel `Stream` for the brute-force AID decryption, adding more cores would improve the performance immediately, requiring no code modifications.

6.2.2 Server performance with heuristics

This section demonstrates that once the server starts using heuristics exploiting locality and caching, its performance increases. To test this, the expected user behaviour was simulated as specified in Section 5.2.

Lichtenstein was chosen as the country to run the simulation on. Lichtenstein has 764 cell towers. In each iteration, the number of users was increased and a subset of cell towers was selected so that the number of users per cell tower remained set to 1,000. Every user was allocated to a random cell tower and an observer was selected at each cell tower. Each

observer collected 1,000 AIDs from random users at their cell tower. The total number of Bellrock users n was varied from 10,000 to 100,000. The total number of resolved AIDs was therefore $1000 \cdot n/1000 = n$ in each case.

The heuristics helped in two ways:

1. When resolving AIDs Alice observed, the server cached the last 1,000 UIDs seen by Alice and tried these before trying other UIDs.

2. When resolving AIDs Alice observed, only users at Alice's cell tower were considered.

Figure 6.12 shows the relationship between the number of users on the server and the time needed to decrypt 10,000 AIDs. The number of cell towers is set so that the average number of users per cell tower is constant, i.e. 1,000. So for instance with 40,000 users the number of cell towers is set to 40.

The cell towers are sampled randomly from the cell towers of Lichtenstein (which has 764 cell towers in total), so they provide a realistic setting concerning density and coarse location overlaps, etc.

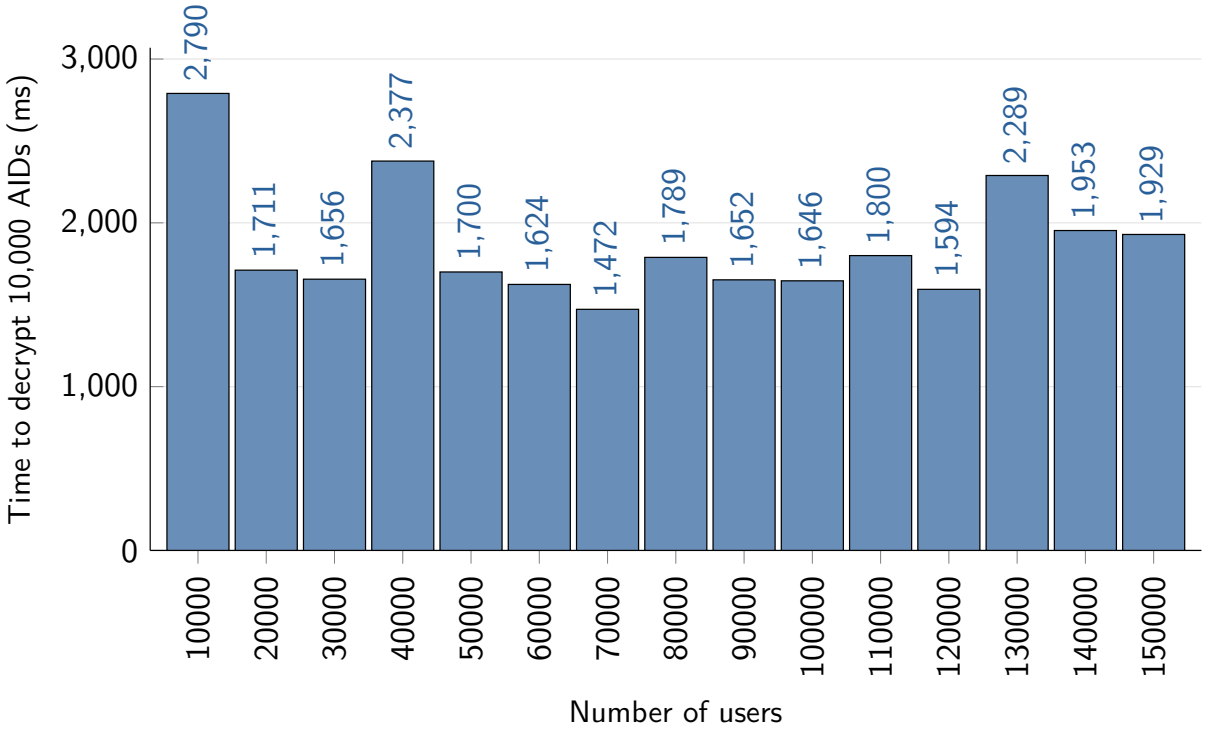


Figure 6.12: Time to decrypt 10,000 AIDs with a varying number of users registered with the Bellrock server. The number of users per cell tower is kept on 1,000 users per cell tower. The heuristics now assures that the number of brute force AID decryptions is bounded by the number of users per cell tower.

This experiment shows the power of the heuristics. Figure 6.13 provides a visual comparison against the system without the heuristics. The time savings increase as the number of users increase. This shows that the heuristics enable Bellrock server scaling.

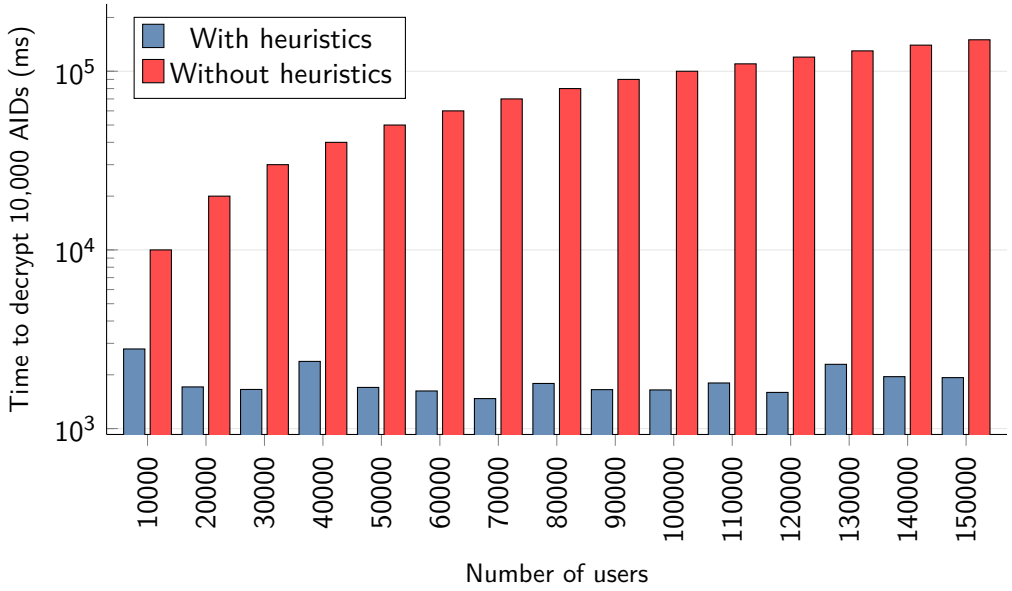


Figure 6.13: Comparison of server decryption speeds with and without heuristics. The number of users per cell tower is set to 1,000. Note that the y-axis has logarithmic scale!

6.3 Possible attacks

This section examines possible attacks on the Bellrock system and how well it copes with them or what would need to be added in order to cope with them. A set of assumptions needs to be established so that the discussion can focus on the security of Bellrock itself rather than on the underlying protocols and technologies:

1. The Bellrock server is secure, i.e. adversaries have no access to its databases, binaries or server memory. The only way the server can be accessed is via the public API.
2. The communication between the Bellrock clients and the Bellrock server over the internet is private and secure.
3. The system has access to a reliable source of random numbers with high entropy.
4. BLE can not be jammed: it is impossible to prevent sending and receiving BLE packets.

Based on these assumptions, the following attacks will be analysed: spoofing, Denial-of-Service (DoS), brute-force UUID guessing and tracking. Throughout the next sections we will consider Alice and Bob to be authorised users of the Bellrock system and Mallory will be a malicious adversary trying to destroy Bellrock.

6.3.1 Spoofing

In the context of the Bellrock system, spoofing means sending BLE advertising packets with AID generated not according to the Bellrock protocol but rather by certain Mallory's scheme.

Mallory can broadcast completely random AIDs. This would cause problems for both the client and the server. Alice would need to record and upload to the server a huge number of AID observations, which might lead to a power leak. Alice can prevent this by putting a limit on the time intervals when she turns her BLE receiver on.

The server would try to decrypt the given AIDs with all heuristics failing due to the fact that likely there is no key in the user database that is able to decrypt the AID received from Mallory.

The server could deal with such packets either by trying to decrypt them using all keys in its database or by dropping them after the temporal and spatial heuristics fail. While currently the first approach is taken, changing to the latter would make the server more resistant to such attacks. In either case, the observations with unresolvable AIDs are simply dropped.

6.3.2 Denial-of-Service attack

The previous attack – spoofing – was a DoS attack in a way. It is, however, possible to conduct a DoS attack on Bellrock server much more directly: Mallory registers multiple new clients, generates a huge number of fake observations and sends them to the server. Moreover, Mallory can be smart and rather than sending random AIDs, he could send valid packets that he observed before.

This can be prevented on the server side by blocking users who exceed a well-established limit of observations per unit time either by completely rejecting their request or by resolving only a limited number of observations.

6.3.3 Brute-force UUID guessing

Mallory could collect a huge number of AIDs from, say, Bob and use a brute-force over the key space to try to decrypt them. I.e. with at least two AIDs, Mallory could try all possible keys to decrypt both of them and if he gets the same first 8 bytes in both cases, then he has very high probability of having correctly guessed Bob's (UUID, key) pair. Alternatively, Mallory could find a (UUID, key) pair that also satisfies this property. Mallory can, however, verify this easily using multiple more fresh AIDs transmitted by Bob.

Firstly, this is *very* costly. Generously assuming that Mallory can try 10^{12} keys per second (for comparison, Bellrock on my laptop can do $5 \cdot 10^6$), it would still take Mallory about

$$\frac{2^{128}\ \text{keys}}{10^{12}\ \text{keys/s}} \approx 10^{29}\ \text{s} \approx 10^{8} \times \text{universe age},$$

assuming AES with 128-bit keys.

It is clearly not feasible for Mallory to decrypt the key. However, for the sake of argument, assume Mallory manages to decrypt the key anyway. While in the case of asymmetric cryptography (see Section 4.1) this would compromise the entire system, in this case only Bob's AIDs are compromised and hence only *one* user in the Bellrock system becomes trackable at a *large* effort. Most likely Mallory would not be willing to pay this cost.

Let us go even further and assume that Mallory is willing to pay the huge price for cracking Bob's secret key. In this case Bellrock can be easily extended to support giving out new keys to its clients. This would complicate the heuristic brute-force decryption (i.e. there would be multiple keys per user to be tried), but only by a small factor.

6.3.4 Tracking

Rather than broadcasting UUIDs, Bellrock clients broadcast an AID and they don't broadcast when the user is moving. When the user becomes stationary again, a new AID is generated and broadcasted.

This prevents tracking in environments with a lot of other Bellrock clients. Imagine a situation, where Bob is the only person in a certain area. Whatever AID he decides to broadcast, since he is the only one there, it is certain he is the broadcaster and hence he can be tracked. However, in such a situation, Bob is easily trackable even without broadcasting any BLE packets, e.g. by GSM, Wi-Fi or just by visual observation.

It is therefore not an issue of the Bellrock system per se that Bob is trackable in such situations. The reason is rather the fact that Bob is the only person in a certain area and in such cases "crowd anonymity" vanishes.

Moreover, Bob could be tracked by the frequency of his AID broadcast. This is however prevented by the Bellrock client by using random broadcasting interval lengths.

6.4 Performance of the Bellrock client

6.4.1 BLE range

The maximum measured range of the Bellrock clients was 20 metres in a rectangular room with no obstacles between the broadcasting smartphone and the listening smartphone. However, in less ideal situations with obstacles (in particular metal obstacles), the BLE range decreased to about 8 metres.

6.4.2 Battery impact

According to measurements from PowerTutor [21], the Bellrock client's average power consumption was 2.0 mA. Since the power consumption of the phone's screen was about 400 mA, this is negligible.

This was also verified by running the Bellrock client for 16 hours on a Nexus 5X device with a 2,700 mAh battery. The Bellrock client was the only running application (ignoring the Android OS), the Wi-Fi, the mobile network and the display were off during the experiment. The battery was drained by 5%. This puts an *upper bound* on the Bellrock client power consumption to

$$\frac{2700 \text{ mAh} \cdot 0.05}{16 \text{ h}} = 8.5375 \text{ mA},$$

which is consistent with the PowerTutor measurements. The Bellrock client is therefore suitable for running on user smartphones without causing any inconvenience.

The Bell Rock Lighthouse, standing strong in the midst of a storm, guiding ships to safety. [2]

Chapter 7

Conclusion

Rather than building a framework for providing exact (indoor) location, Bellrock provides facilities for examining the *context* in which a device is. This can be used to provide an approximate location, tell about proximity of friends or it can be used in sociological research to determine human behaviour and interactions.

Using Anonymous AIDs, Bellrock achieves user anonymity and privacy while enjoying the same benefits that would be present if UUIDs were broadcasted. Rather than using asymmetric (public-key) cryptography, due to BLE constraints and security issues, Bellrock uses symmetric-key cryptography to encrypt UUIDs into AIDs. While this might seem not feasible, the evaluation shows the server's performance and temporal and spatial heuristics enable excellent scalability.

The system was evaluated using a simulation based on the information about user movement obtained from the Bat System logs. The Bat System logs were used to show how users move around, how much time they spend moving, how many encounters with other users they have per day and the distribution of the number of users they hear broadcasting. The evaluation showed that the approach taken by Bellrock (i.e. using dynamic BLE beacons) is feasible and could be used in office-like environments.

Based on the evaluation of the user behaviour, the performance of the Bellrock server was evaluated using simulated data together with real data about the positions of cell towers in Lichtenstein. This showed that the Bellrock server performs well and that the heuristics really enable AID decryption performance that is independent of the total number of Bellrock users on the Bellrock server.

The Bellrock client was evaluated to test its real-life capabilities. It was shown that the assumption about 7.5 meter broadcasting radius is realistic. The power usage impact of the Bellrock client was examined using an Android profiling application PowerTutor and also by letting the Bellrock client run for 16 hours. It was shown that the performance impact of the Bellrock client is small, with less than 8 mA power consumption which is negligible on today's devices with about 2,500 mAh batteries.

Overall, the project explores a novel way of using the BLE beacons and shows that it is feasible to turn every smartphone into a BLE beacon without compromising user privacy, anonymity or their device's battery life. The Bellrock platform is built so that it can be easily extended and it can be used as a basis for providing context-aware services in the future.

7.1 Future work

The Bellrock system could be further enhanced in the following ways:

- **Crowd broadcast:** The current approach of Bellrock clients is to stop broadcasting whenever the user is moving to prevent tracking. However, this would be problematic in areas like London Underground, where people move most of the time. Hence most of the clients won't broadcast there. The solution would be to change Bellrock's behaviour in cases when there is a large amount of Bellrock users around: Instead of stopping the broadcast, change the AID randomly with a mean of, say, 10 seconds. Since large number of Bellrock clients around would do the same, a random observer would not be able to track users while the Bellrock server would get extra information.
- **Location service:** The location service would provide location information based on the currently observed UUIDs. Inferring this information would be rather challenging, as it would mean finding user patterns in space and time and linking them together. For example if Alice hears Bob, the baker, only at the bakery, then Alice is quite likely at the bakery when she hears Bob. Or if Alice hears her colleagues Matthew and Natalie at weekdays, she is most likely at work. But if she hears them on Sunday and she also hears UUID of barman Dominik, she is most likely at a bar, not at work. This would involve building a huge graph incorporating these relationships on the server.
- **Information sharing:** The BLE packets sent by each client could incorporate more information that would be used for further inference by the server and the neighbouring clients. For instance, if a client knows their location, it could broadcast it in the advertisement packets along with its AID.
- **Incorporation of static beacons:** There are already Wi-Fi networks that could be used to provide a coarse location estimate. Moreover, static BLE beacons (such as iBeacons or Eddystone) could be incorporated along with a database of their locations.

Bibliography

[1] Wikipedia, "Robert Stevenson (civil engineer) — Wikipedia, The Free Encyclopedia," 2016. [Online; accessed 30-May-2016].

[2] R. Stevenson, "An Account of the Bell Rock Light-house," *Constable, Edinburgh*, 1824.

[3] J. Poushter, "Smartphone ownership and internet usage continues to climb in emerging economies." http://www.pewglobal.org/2016/02/22/smartphone-ownership-and-internet-usage-continues-to-climb-in-emerging-economies/. [Online; accessed 05-May-2016].

[4] Wikipedia, "Harry Beck — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Harry_Beck&oldid=712205648, 2016. [Online; accessed 19-May-2016].

[5] Wikipedia, "Tube map — Wikipedia, The Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Tube_map&oldid=710567178, 2016. [Online; accessed 19-May-2016].

[6] H. Liu, H. Darabi, P. Banerjee, and J. Liu, "Survey of wireless indoor positioning techniques and systems," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 37, no. 6, pp. 1067–1080, 2007.

[7] Y. Gu, A. Lo, and I. Niemegeers, "A survey of indoor positioning systems for wireless personal networks," *Communications Surveys & Tutorials, IEEE*, vol. 11, no. 1, pp. 13–32, 2009.

[8] Apple Inc., "iBeacon for Developers." https://developer.apple.com/ibeacon/. [Online; accessed 27-May-2016].

[9] AltBeacon, " AltBeacon, The Open and Interoperable Proximity Beacon Specification ." http://altbeacon.org/. [Online; accessed 27-May-2016].

[10] S. Feldmann, K. Kyamakya, A. Zapater, and Z. Lue, "An indoor bluetooth-based positioning system: Concept, implementation and experimental evaluation.," in *International Conference on Wireless Networks*, pp. 109–113, 2003.

[11] Y. Inoue, A. Sashima, and K. Kurumatani, "Indoor positioning system using beacon devices for practical pedestrian navigation on mobile phone," in *Ubiquitous Intelligence and Computing*, pp. 251–265, Springer, 2009.

[12] M. Choi, W.-K. Park, and I. Lee, "Smart office energy management system using bluetooth low energy based beacons and a mobile app," in *Consumer Electronics (ICCE), 2015 IEEE International Conference on*, pp. 501–502, IEEE, 2015.

[13] K. C. Cheung, S. S. Intille, and K. Larson, "An inexpensive bluetooth-based indoor positioning hack," in *Proceedings of UbiComp*, vol. 6, Citeseer, 2006.

[14] S. Bluetooth, "Bluetooth specification version 4.2," *Bluetooth SIG*, 2014.

[15] "OpenCellId." http://opencellid.org/. [Online; accessed 05-May-2016].

[16] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "NIST special publication 800-57," *NIST Special Publication*, vol. 800, no. 57, pp. 1–142, 2007.

[17] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel AVX: New frontiers in performance improvements and energy efficiency," *Intel white paper*, 2008.

[18] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper, "Implementing a sentient computing system," *Computer*, vol. 34, no. 8, pp. 50–56, 2001.

[19] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, "The anatomy of a context-aware application," *Wireless Networks*, vol. 8, no. 2/3, pp. 187–197, 2002.

[20] A. Zidek, "Minuscule." http://augustin.zidek.eu/minuscule. [Online; accessed 12-May-2016].

[21] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 105–114, ACM, 2010.

Appendix A

Source code

This chapter contains the most important parts of Bellrock system source code.

A.1 IDAnonymizer

```
1  package eu.zidek.augustin.bellrock.identification;
2
3  import java.security.InvalidKeyException;
4  import java.security.Key;
5  import java.security.NoSuchAlgorithmException;
6  import java.security.SecureRandom;
7
8  import javax.crypto.BadPaddingException;
9  import javax.crypto.Cipher;
10 import javax.crypto.IllegalBlockSizeException;
11 import javax.crypto.NoSuchPaddingException;
12
13 /**
14  * Class for turning UIDs into Anonymous IDs (AIDs).
15  * Created by Augustin Zidek on 2016-01-18.
16  */
17 public class IDAnonymizer {
18
19     /**
20      * @param byteCount Number of random bytes.
21      * @return Array with the given number of random bytes.
22      */
23     private byte[] getRandomByteArray(final int byteCount) {
24         final SecureRandom random = new SecureRandom();
25         final byte[] randomBytes = new byte[byteCount];
26         random.nextBytes(randomBytes);
27         return randomBytes;
28     }
29
30     private byte[] padID(final byte[] id) {
31         final byte[] idWithPad = new byte[id.length + MsgConstants.NONCE_LENGTH];
32         final byte[] pad = getRandomByteArray(MsgConstants.NONCE_LENGTH);
33
34         // Copy original ID
35         for (int i = 0; i < id.length; i++) {
```

```
36              idWithPad[i] = id[i];
37          }
38
39          // Copy the padding
40          for (int i = id.length; i < idWithPad.length; i++) {
41              idWithPad[i] = pad[i - id.length];
42          }
43
44          return idWithPad;
45      }
46
47      /**
48       * Anonymizes the given ID by appending a random 8 byte nonce and encrypting the
             whole thing.
49       *
50       * @param uid The ID to be anonymized.
51       * @return An anonymous ID. The ID is anonymized by appending a random
52       * 8 byte nonce to it and then encrypting in using a AES block cipher.
53       */
54      public AnonymousID getAnonymousID(final UID uid, final Key key) throws
             SecurityException {
55          try {
56              // Initialize the cipher: AES in ECB (Electronic Code Book) mode, since
                 we are
57              // encrypting only a single block here. Hence, use also no padding.
58              final Cipher cipher = Cipher.getInstance(MsgConstants.CIPHER_PARAMETERS);
59              cipher.init(Cipher.ENCRYPT_MODE, key);
60
61              // Pad the ID with a nonce
62              final byte[] idWithPad = this.padID(uid.getBytes());
63
64              // Encrypt the ID+pad, i.e. return a random ID from observer's point of
                 view
65              final byte[] anonymousID = cipher.doFinal(idWithPad);
66              return new AnonymousID(anonymousID);
67
68          }
69          // Catch the problems with the AES initialisation and throw as a
                SecurityException
70          catch (NoSuchAlgorithmException | NoSuchPaddingException |
                InvalidKeyException e) {
71              throw new SecurityException("The␣AES␣could␣not␣be␣initiated:␣" + e.
                    getMessage());
72          }
73          // Catch the problems with the encryption and throw as a SecurityException
74          catch (BadPaddingException | IllegalBlockSizeException e) {
75              throw new SecurityException("There␣was␣an␣error␣encrypting␣the␣ID:␣" + e.
                    getMessage());
76          }
77
78      }
79  }
```

A.2 IDDecryptor

```java
package eu.zidek.augustin.bellrock.identification;

import java.util.List;
import java.util.NoSuchElementException;

import eu.zidek.augustin.bellrock.server.BellrockUser;

/**
 * Class for Anonymous ID decryption. Created by Augustin Zidek on 2016-01-25.
 */
public class IDDecryptor {

    /**
     * Decrypt the Anonymous ID without knowing the key, but rather only the map
     * between the keys and the UIDs used in the system. The nonce that was
     * encrypted together with the UID is discarded.
     *
     * @param anonymousID The Anonymous ID to be encrypted.
     * @param decryptingUsers Bellrock users that should be tried to decrypt
     *            this AID.
     * @return The user whose UID was encrypted as the Anonymous ID.
     */
    public static BellrockUser decryptAnonymousID(final AnonymousID anonymousID,
            final List<BellrockUser> decryptingUsers) {
        // Try to decrypt the AID using all user keys in the system
        userLoop: for (final BellrockUser user : decryptingUsers) {
            final byte[] userUID = user.getUID().getBytes();
            try {
                // Decrypt the AID using this user's key
                final byte[] decryptedUID = user.decryptAID(anonymousID);
                // If the decrypted UID matches this user's UID, we found it!
                for (int i = 0; i < MsgConstants.UID_LENGTH; i++) {
                    if (decryptedUID[i] != userUID[i]) {
                        continue userLoop;
                    }
                }
                return user;
            }
            // In case there is a Security Exception, skip this particular key
            catch (final SecurityException e) {
                continue userLoop;
            }
        }
        // No UID matches the given Anonymous ID and the key to UID map
        return null;
    }

    /**
     * Decrypt the Anonymous ID without knowing the key, but rather only the map
     * between the keys and the UIDs used in the system. The nonce that was
     * encrypted together with the UID is discarded. This method uses parallel
     * stream to speed up the decryption. If there is sufficient memory
     * available to the system this allows performance improvement.
     *
     * @param anonymousID The Anonymous ID to be encrypted.
     * @param decryptingUsers Bellrock users that should be tried to decrypt
     *            this AID.
```

```
58      * @return The user whose UID was encrypted as the Anonymous ID.
59      */
60     public static BellrockUser decryptAnonymousIDParallel(
61             final AnonymousID anonymousID,
62             final List<BellrockUser> decryptingUsers) {
63         try {
64             // Use each of the key to decrypt the Anonymous ID
65             return decryptingUsers.parallelStream().filter(user -> {
66                 final byte[] userUID = user.getUID().getBytes();
67                 try {
68                     // Decrypt the AID using this user's key
69                     final byte[] decryptedUID = user.decryptAID(anonymousID);
70                     // If the decrypted UID matches this user's UID, found it!
71                     for (int i = 0; i < MsgConstants.UID_LENGTH; i++) {
72                         if (decryptedUID[i] != userUID[i]) {
73                             return false;
74                         }
75                     }
76                     return true;
77                 }
78                 // If there is a Security Exception, skip this particular key
79                 catch (final SecurityException e) {
80                     return false;
81                 }
82             }).findAny().get();
83         }
84         catch (final NoSuchElementException e) {
85             return null;
86         }
87     }
88 }
```

A.3 BellrockServer:addObservation()

```
1
2     /**
3      * Adds all the observations into the database. Moreover, Anonymous IDs in
4      * the observations are resolved, if possible.
5      *
6      * @param observations The observations made by a user. The observations
7      *             must be sorted by the time.
8      * @return The number of observations that were successfully resolved.
9      */
10    public int addObservations(final Observations observations) {
11        // Get the observer
12        final BellrockUser observer = this.um
13                .getUser(observations.getObserver());
14
15        // Get the list of observer's locations
16        final Observation firstObs = observations.getFirstObservation();
17        final Observation lastObs = observations.getLastObservation();
18        final List<UserLocation> observerLocations = observer
19                .getLocations(firstObs.getTime(), lastObs.getTime());
20
21        // The observer went to several locations. At each location, they could
```

```
22          // have met some users. Get the lists of them and then use these lists
23          // later when resolving the individual observations.
24          final Map<CoarseLocation, List<BellrockUser>> usersMetAtLocations = new
                HashMap<>();
25          for (final UserLocation observerLoc : observerLocations) {
26              final List<UID> potentialUsersUIDs = this.db.getUsersAtPlaceAtTime(
27                      observerLoc.getLocation(), observerLoc.getStart(),
28                      observerLoc.getEnd());
29              final List<BellrockUser> usersMetAtLocation = this.um
30                      .getUsers(potentialUsersUIDs);
31              usersMetAtLocations.put(observerLoc.getLocation(),
32                      usersMetAtLocation);
33          }
34
35          int resolvedObsCount = 0;
36          for (final Observation observation : observations.getObservations()) {
37              // Decrypt using user's most recently seen UIDs and peers
38              final boolean resolvedUsingRecentAndPeers = observer
39                      .resolveObservationUsingRecentAndPeers(observation);
40              if (resolvedUsingRecentAndPeers) {
41                  resolvedObsCount++;
42                  continue;
43              }
44
45              // Decrypt using UIDs who were at the same location at the same time
46              final List<BellrockUser> potentialUsers = usersMetAtLocations
47                      .get(observation.getLocation().toCoarseLocation());
48
49              final BellrockUser resolvedUser = observation
50                      .resolveAID(potentialUsers);
51              if (resolvedUser != null) {
52                  observer.addLastSeenUser(resolvedUser);
53                  resolvedObsCount++;
54              }
55          }
56
57          // Store all (some possible resolved) observations in the db
58          this.db.batchAddObservations(observations);
59          return resolvedObsCount;
60      }
```

A.4 BellrockUser

```
1  package eu.zidek.augustin.bellrock.server;
2
3  import java.security.InvalidKeyException;
4  import java.security.Key;
5  import java.security.NoSuchAlgorithmException;
6  import java.time.Instant;
7  import java.util.ArrayList;
8  import java.util.Iterator;
9  import java.util.List;
10 import java.util.Map;
11
12 import javax.crypto.Cipher;
```

```
13 import javax.crypto.NoSuchPaddingException;
14
15 import eu.zidek.augustin.bellrock.identification.AnonymousID;
16 import eu.zidek.augustin.bellrock.identification.MsgConstants;
17 import eu.zidek.augustin.bellrock.identification.UID;
18 import eu.zidek.augustin.bellrock.util.LRUCache;
19
20 /**
21  * Class for storing user of the Bellrock system. The most essential information
22  * is the UID and the shared server-client key.
23  *
24  * @author Augustin Zidek
25  *
26  */
27 public final class BellrockUser {
28     // User information
29     final UID userUID;
30     final Key userKey;
31     Cipher cipher = null;
32
33     // Locations of the users
34     final List<UserLocation> locations = new ArrayList<>();
35
36     // Peers and lastly seen UIDs
37     final List<BellrockUser> peers = new ArrayList<>();
38     final Map<UID, BellrockUser> userLRUCache = new LRUCache<>(
39             ServerConsts.LAST_SEEN_USER_CACHE_SIZE);
40
41     BellrockUser(final UID userUID, final Key key) {
42         this.userUID = userUID;
43         this.userKey = key;
44     }
45
46     BellrockUser(final UID userUID, final Key key,
47             final Cipher initialisedCipher) {
48         this.userUID = userUID;
49         this.userKey = key;
50         this.cipher = initialisedCipher;
51     }
52
53     /**
54      * @return The unique user ID.
55      */
56     public UID getUID() {
57         return this.userUID;
58     }
59
60     /**
61      * @return The user-server shared key used for encrypting UIDs when
62      *         broadcasting Anonymous IDs.
63      */
64     public Key getKey() {
65         return this.userKey;
66     }
67
68     /**
69      * @return The list of peers of this Bellrock user.
70      */
```

```
71      public List<BellrockUser> getPeers() {
72          return this.peers;
73      }
74
75      /**
76       * Adds a peer to this Bellrock user.
77       *
78       * @param peer The new peer of the user.
79       */
80      public void addPeer(final BellrockUser peer) {
81          this.peers.add(peer);
82      }
83
84      /**
85       * Removes a peer of this Bellrock user.
86       *
87       * @param peer The peer to be removed.
88       */
89      public void deletePeer(final BellrockUser peer) {
90          this.peers.remove(peer);
91      }
92
93      /**
94       * @return The collection of Bellrock users this Bellrock user saw recently.
95       */
96      public List<BellrockUser> getLastSeenUIDs() {
97          final List<BellrockUser> recentUsers = new ArrayList<>(
98                  ServerConsts.LAST_SEEN_USER_CACHE_SIZE);
99          for (final BellrockUser u : this.userLRUCache.values()) {
100             recentUsers.add(u);
101         }
102         return recentUsers;
103     }
104
105     /**
106      * Add a user into the list of users the Bellrock user has seen lately. A
107      * LRU cache is used.
108      *
109      * @param user The UID to be added.
110      */
111     public void addLastSeenUser(final BellrockUser user) {
112         // The cache doesn't contain the user, add it.
113         if (!this.userLRUCache.containsKey(user.getUID())) {
114             this.userLRUCache.put(user.getUID(), user);
115         }
116         // The cache contains the user. Remove the user from its old position
117         // and put it in the new position in order to increase its freshness.
118         else {
119             this.userLRUCache.remove(user.getUID());
120             this.userLRUCache.put(user.getUID(), user);
121         }
122     }
123
124     /**
125      * Adds a new location into the list of locations the user visited.
126      *
127      * @param location The location.
128      */
```

```
129     public void addLocation(final UserLocation location) {
130         this.locations.add(location);
131     }
132 
133     /**
134      * Adds a new list of locations into the list of locations the user visited.
135      *
136      * @param locations The list of locations.
137      */
138     public void addLocations(final List<UserLocation> locations) {
139         this.locations.addAll(locations);
140     }
141 
142     /**
143      * Goes through the locations of this user and purges all that happened
144      * before the given time instant.
145      *
146      * @param purgeEnd The time instant after which the locations should be
147      *             kept.
148      */
149     public void purgeOldLocations(final Instant purgeEnd) {
150         final Iterator<UserLocation> i = this.locations.iterator();
151 
152         while (i.hasNext()) {
153             final UserLocation location = i.next();
154             if (location.isBefore(purgeEnd)) {
155                 i.remove();
156             }
157         }
158     }
159 
160     /**
161      *
162      * @param start The start of the interval.
163      * @param end The end of the interval.
164      * @return All user locations where the user was during the given time
165      *         interval.
166      */
167     public List<UserLocation> getLocations(final Instant start,
168             final Instant end) {
169         final List<UserLocation> locationsInInterval = new ArrayList<>();
170         for (final UserLocation location : this.locations) {
171             if (location.overlapsWith(start, end)) {
172                 locationsInInterval.add(location);
173             }
174         }
175         return locationsInInterval;
176     }
177 
178     /**
179      * Tries to resolve the given observation using the user's last seen users
180      * and user's peers.
181      *
182      * @param o The observation to be resolved, i.e. whose sender's UID should
183      *            be identified.
184      * @return <code>true</code> if the observation sender's UID was
185      *         successfully resolved, <code>false</code> otherwise.
186      */
```

```
187     public boolean resolveObservationUsingRecentAndPeers(final Observation o) {
188         // Decrypt using user's most recently seen users
189         BellrockUser resolvedUser = o.resolveAID(this.getLastSeenUIDs());
190         // If resolved, add the user into the list of recently seen users
191         if (resolvedUser != null) {
192             this.addLastSeenUser(resolvedUser);
193             return true;
194         }
195 
196         // Decrypt using user's peers
197         resolvedUser = o.resolveAID(this.peers);
198         // If resolved, add the peer into the list of recently seen users
199         if (resolvedUser != null) {
200             this.addLastSeenUser(resolvedUser);
201             return true;
202         }
203         return false;
204     }
205 
206     /**
207      * Initialising Cipher in decryption mode with a certain key is expensive.
208      * It is therefore optimal to do it only once and then reuse the initialised
209      * cipher. This method initialises new cipher that can decode Anonymous IDs
210      * using this user's key only when called for the fist time. Afterwards, it
211      * returns the same, already initialised, cipher.
212      *
213      * @return A cipher initialised for decryption using the user's Anonymous
214      *         ID. <code>null</code> is returned in case the cipher can't be
215      *         initialised.
216      */
217     public Cipher getInitialisedCipher() {
218         // Cipher not initialised, initialise it
219         if (this.cipher == null) {
220             try {
221                 this.cipher = Cipher
222                         .getInstance(MsgConstants.CIPHER_PARAMETERS);
223                 this.cipher.init(Cipher.DECRYPT_MODE, this.userKey);
224             }
225             // None of these exceptions will happen, since the algorithm is
226             // valid, the padding is valid and the key also.
227             catch (final NoSuchAlgorithmException | NoSuchPaddingException
228                     | InvalidKeyException e) {
229                 e.printStackTrace();
230                 return null;
231             }
232         }
233         return this.cipher;
234     }
235 
236     /**
237      *
238      * Decrypts the given Anonymous ID using this user's key.
239      *
240      * @param anonymousID The Anonymous ID.
241      * @return The UID which was encrypted into Anonymous ID, the decrypted
242      *         nonce is also included.
243      * @throws SecurityException if the decryption fails.
244      */
```

```
245     public byte[] decryptAID(final AnonymousID anonymousID)
246             throws SecurityException {
247         try {
248             // Decrypt the AID using the cipher of this user
249             return this.getInitialisedCipher().update(anonymousID.getBytes());
250         }
251         // In case of problem in decryption, throw a SecurityException
252         catch (final IllegalArgumentException e) {
253             throw new SecurityException(e.getMessage());
254         }
255     }
256 
257     @Override
258     public boolean equals(final Object obj) {
259         if (!(obj instanceof BellrockUser)) {
260             return false;
261         }
262 
263         return this.getUID().equals(((BellrockUser) obj).getUID());
264     }
265 
266     @Override
267     public int hashCode() {
268         return this.getUID().hashCode();
269     }
270 
271 }
```