Secure Access Control Using Mobile Bluetooth Devices

M.Sc. Thesis

Allan Beaufour Larsen
<beaufour@diku.dk>

July 16, 2003
(Revised: September 30, 2003)

D I K U

Department of Computer Science
University of Copenhagen
Denmark

Abstract

The physical key is an integrated and natural part of most people's life. It is a well-tested and well-known technology, but it also has its flaws. In particular, for companies needing access to many different private buildings, each with its own lock and key, the distribution of keys to the right employee at the right time is a complex and costly affair. Furthermore, carrying a large number of keys is a burden for each employee and increases the chance of keys getting stolen or lost.

Our goal is to design a solution for secure access control that can replace physical keys for accessing private buildings. We propose a solution using digital keys on Bluetooth-enabled mobile phones providing wireless and automatic unlocking. The design allows easy distribution of keys to users, with access control enforced by easily deployable autonomous lock devices allowing a non-centralized multi-company approach.

We present a solution for fully automatic discovery and connection establishment using Bluetooth. Moreover we present a simple and secure authentication and access control protocol, allowing each mobile phone to unlock multiple different locks using a single identity certificate and public unencrypted digital keys. We build a prototype application for a mobile phone, establish a cost model for the mobile phone and measure the exact energy consumption and time usage of the application. Finally, we relax the assumptions about using Bluetooth and a mobile phone, and discuss the requirements for an optimal mobile device.

Acknowledgements

The work of this thesis was conducted by Allan Beaufour Larsen between January 10th, 2003 and July 16th, 2003.

I would like to thank my adviser, Philippe Bonnet, for guidance, comments, suggestions and ideas during the project. The angry red and encrypted handwriting that populated the drafts of the thesis was also invaluable.

I would also like to thank the people in my office for discussing various issues with me. Especially Christian Boesgaard for security issues, Martin Leopold for Bluetooth issues, and Bjarke Buur Mortensen for C++ issues. I would also like to thank the same three, Ole Nimb, Troels Blum, and Mads Dydensborg for their proofreading and comments on the thesis. Thanks also to Janus Lundager for helping me with measurement issues.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

What is more natural than using a key to unlock a door? It is something that everyone does almost every day, and it is an integrated part of our daily life. It is also a very well-known and well-tested technology. Despite these arguments, keys as we know them might not be the device of choice to open doors in the future. We have all experienced problems like carrying too many keys while forgetting the one key we need, or maybe having a lock replaced because we lost a single key. Interestingly, keys never show up in science fiction movies!

The problems with physical keys has even bigger implications for companies in the business of mail or goods delivery. These companies need access to many different private buildings—doors spread over a wide geographical area and governed by many different owners. Personnel need to carry keys for each single door on the delivery route. Carrying all these keys (literally kilograms) is a hassle in the daily use and vulnerable to theft. But just as important, routes, personnel, and locks change over time, making it a resource consuming operation to assure that all personnel have the right keys at the right time.

To tackle the above issues, we propose to replace physical keys with digital keys that:

- can easily be distributed to users
- can only be used by the correct user
- can be specialized for each user so each key will be unique
- can be restricted to a given time or date range
- allows many keys to be contained in the same small device

To hold the keys some kind of *mobile device* is needed: A portable electronic device that contains the keys and can communicate with a device that can unlock a door, a *lock device*. To be usable, a portable device needs to have a small form factor and needs to be powered by an integrated power source. These two factors influence processing power, storage capacity and available energy, normally making these resources scarce. Door unlocking need not be the only purpose of the mobile device though. It could be a secondary functionality of the user's *Personal Server* [78].

We envision the following scenario:

> The mobile device contains all the keys. When the user approaches a door, the appropriate key is transfered to the lock automatically and without physical contact, the lock is unlocked, and the door can be opened.

The automatic unlocking makes it possible for the user to unlock the door while carrying goods, making the system very user-friendly.

We propose to use a mobile phone equipped with Bluetooth as the mobile device. First, mobile phones are getting very common and bringing your mobile phone with you is becoming as natural as bringing your keys. Second, mobile phones are becoming increasingly open to third-party developers, while the hardware at the same time is getting more and more powerful. This makes mobile phones a suitable platform for uses beyond making and receiving phone calls. Combined with the ability to perform short-range wireless communications via Bluetooth, the mobile phone is a good candidate for the mobile device.

The idea of using a mobile phone as a more general device is not unique—calendar-functionality, browsers, and games are already incorporated. But mobile phones could evolve to be the generic personal device, or in the words of Roy Want: "Cell phones will be the Personal Server of the future." [78]. We see the use of the mobile phone as an unlocking device as a natural step in this direction.

The goal of this thesis is to design a system that fulfills the above scenario, with the main focus on the challenges concerning the mobile device and its interaction with the lock device. Although our starting point is to use a Bluetooth-enabled mobile phone as a mobile device and solving the needs for delivery companies, the solution should be as general as possible. The requirements for both lock device and mobile device should be as limited as possible.

This work is inspired by a previous collaboration with a company developing this kind of solution in Denmark[1]. Some of the assumptions we make in the thesis are dictated by this company's requirements. Also, the study of Bluetooth is of particular interest in this context. We focus on describing the design space for all the problems we study in order to facilitate a possible transfer.

1.1 Challenges

There are several challenges to be met in order to execute the above scenario in practice. The two main challenges are:

[1]Collaboration under a *non-disclosure agreement* prevents us from releasing sensitive informations.

1. Making the system at least as secure as using a physical key.

2. Using Bluetooth as the communication technology.

First, security is naturally of paramount importance in a design aimed at replacing physical keys. We will introduce different aspects of security, discuss the necessity of each of these, and will then analyze which security technologies can be used to secure the system. Interestingly, there is not much directly related work as the mobile phone hardware seems to fall in between the extremely constrained hardware of sensor nodes (see [38]) and the fully equipped PCs.

Second, Bluetooth looks like the natural choice for communication. The number of mobile phones equipped with Bluetooth is steadily increasing, and it is becoming the widely accepted standard for short-range point-to-point communication. But there are important decisions to be made about discovery and connection strategies that will influence the overall system performance. There are also general questions of the overall performance of Bluetooth and the cost overhead of implementing Bluetooth in a device. We study these issues and how well Bluetooth generally suits this scenario.

In handling these challenges, two aspects have vital influence on any decision made: Energy conservation and usability. First, energy conservation plays a major role since we are dealing with a mobile phone that *per se* is energy constrained, as it is battery powered. Thus we have to consider how much energy each part of the system uses, so as not to drain the battery. It does not matter how secure the system is, if the mobile phone cannot operate for more than half an hour. Second, it is important for the system to be at least as easy to use as a normal key. If not, the system will never gain acceptance. We will have to take these aspects into consideration throughout the project.

1.2 Context and Assumptions

We have two basic assumptions about the context. The first concerns energy. We assume that the mobile device is energy constrained and the lock device is not, which we believe is a reasonable assumption since the mobile device is powered by an integrated power source. The lock device is mounted next to a door, most likely in a house where it can be connected to a stable and fixed power source.

The second assumption concerns the communication. Requiring a network connection for each lock device makes the system infeasible, especially for large scale deployment. We therefore assume that the lock device communicates only with the mobile device (via Bluetooth); it is not connected to a network of any kind and will have to function autonomously. The mobile device on the other hand, being a mobile phone, already has the capabilities for communicating with other systems.

A consequence of this is that each mobile device either has to be authorized by direct contact with each lock device, or that an external entity that is trusted by the lock device can authorize mobile devices on its behalf. We choose the latter as it corresponds with the way that keys are normally handled, and seems more viable than the first. There are thus three entities in our scenario:

Lock device A fixed device that is attached to the physical lock unit, and can unlock it.

Mobile Device A mobile device carried by a person. Can communicate with the lock device, and may have a user interface.

External Authority A third party that can communicate with the mobile device, and is trusted by the lock. No direct communication with the lock device is possible.

The overall scenario is seen in Figure 1.1. One of our goals is to make distribution of keys easy, but exactly how the mobile device obtains the keys from the external authority is beyond the scope of this thesis. We assume that the mobile device has the keys it needs.

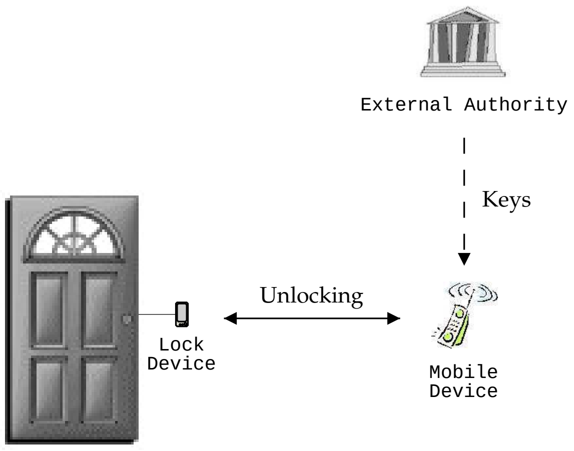


Figure 1.1: The context of the project.

We also have some general assumptions about the hardware. First of all we assume that physical size is an issue for both devices. The mobile phone is carried by a person, and the lock device will have to be mounted near a door as unobtrusively as possible. The physical size, and the manufacturing cost will limit the possible hardware in both devices. The mobile device will furthermore be constrained by being battery powered, which also limits the hardware. All this leads up to the fact that both lock device and mobile device have limited hardware, both in terms of processing power and storage capacity. We do assume though, that the lock device is less constrained than the mobile device as it is neither mobile nor battery-powered.

Lastly, we also assume that there will be a maximum acceptable waiting time for the user before the door is unlocked. The exact amount depends on the individual user, so it is difficult to set a general limit. To have an upper limit, we believe that most users will find a maximum of five seconds tolerable.

1.3 Problem Description and Approach

To summarize, our goal is to design a system that allows a Bluetooth-enabled mobile device to unlock a door without the need for user interaction. The design needs to fulfill the following criteria:

- Both mobile device and lock device are hardware constrained, and the system should pose minimal requirements on the hardware.
- Access control is maintained by an external authority that cannot communicate with the lock devices.
- The lock device must function autonomously.
- Keys can be customized to individual users.
- Key distribution must be easy.
- The system must be just as easy to use as using physical keys.

Our starting point will be a mobile phone as a mobile device and solving the needs of mail and goods delivery companies.

To tackle this problem we will design and implement a prototype for a mobile phone and the surrounding infrastructure. We approach the problem in a bottom-up fashion, and will start by analyzing the discovery of one Bluetooth device by another, and will then gradually build up a system that meets the challenges. We will then set up an experimentation setting that lets us measure the performance of the prototype in terms of time and energy usage. With these experiences we can further analyze how a generic device can be designed, and present criteria and requirements for optimal an optimal mobile device.

1.4 Related Work

The different types of access control systems that exist to replace physical keys do not match the requirements of our scenario. Existing systems either rely on a wireless transceiver or a card. KEELOQ [49] is a solution based on a wireless transceiver. A remote keyless entry solution using KEELOQ is presented in [74]. Most cars have the possibility of remote unlocking. Some car manufactures even allow *keyless start and go* where the user simply needs to carry the key device to be able not just to open, but also start the car[2]. The problem with these kinds of solutions is that they are designed for a one-to-one situation, where one transceiver fits exactly one door.

Card-based access control systems are commonly used in large corporations and use a card equipped either with a magnetic stripe or a microchip. The latter can either be operated so that the users needs to insert the card into

[2]F.x. the Nissan Micra (`http://www.nissan-micra.com/`) or the Mercedes S-class (`http://www.mercedes-benz.com/`).

a reader, or *contactless* where the user just has hold the card near a reader (see [2] for an overview of the technology). Card-based microchip systems are normally passive in the sense that the card itself is activated by and receives power from the reader. It is not capable of functioning autonomously. These systems are typically meant to be used in the domain of one company, and are normally designed so that each lock device is connected to a central access control server.

We have only found one product that use Bluetooth-based mobile devices for access control purposes. XyLoc[3] is a commercial key-less authentication product that enables automatic authorization, designed for Windows-based PCs and requires connection to a central access control server. Bluetooth is an optional communication technology that can replace their normal proprietary system. We have also found a case study using Bluetooth-enabled mobile phones to control access for *gates*, which is not specified further[4]. Unfortunately only marketing information is available.

The different elements of our design naturally build on existing work. Related work on security and Bluetooth are discussed in the relevant sections.

1.5 Contribution

The primary individual contributions of this project are:

- The design of a fully automated Bluetooth connection establishment strategy, under consideration of speed and energy consumption
- The design of a simple, specialized, and secure authentication protocol using asymmetric cryptography on constrained devices
- The combination of the above into an application for automatic door unlocking using existing mobile phones
- The implementation of a prototype application and the measurements of the exact energy and time usage for it, including energy measurements for many features on the specific mobile phone chosen for the prototype

In addition we have contributed with patches and bug-fixes to the Affix Linux Bluetooth stack, the *sdk2unix* Symbian SDK Linux conversion tool and the Symbian SDK documentation.

1.6 Thesis Outline

The thesis consists of six chapters. In Chapter 2, we analyze how the mobile device and lock device automatically discover and connect to each other using Bluetooth. Once the connection is established we can analyze and design a secure access control system using a Bluetooth connection in Chapter 3. We then

[3]`http://www.ensuretech.com/`.
[4]`http://www.codeisland.com/studies/studies.dvd.asp`.

describe the prototype hardware and software implementation of the design in Chapter 4. In Chapter 5 we test and measure the energy and time usage of both hardware and software, and evaluate the overall design. We then use the knowledge gathered to analyze the requirements for an optimal device in Chapter 6. In Chapter 7, we finish the thesis with a conclusion on the results, the goals reached, and topics for future research.

To fully appreciate this thesis the reader should have a background in computer science and have knowledge of Bluetooth and security.

CHAPTER 2

Making the Connection

Fortune speed us.
– William Shakespeare, A Winter's Tale, Act IV

In this chapter we will analyze how a lock device and a mobile device successfully establishes a connection via Bluetooth. To fulfill our requirements this needs to happen without any user interaction, which means that it has to happen automatically. That is, the connection needs to be established automatically when a user is in front of a door that she wants to open. Bluetooth is primarily designed as a cable replacement technology, not quite for our scenario. The main challenge is to establish the connection as fast as possible without any user interaction, while still being energy efficient.

We will start by analyzing the Bluetooth device discovery mechanism and afterwards the connection establishment. In the last section we will summarize our choices, and evaluate the suitability of Bluetooth in our scenario.

For an introduction to Bluetooth see [14].

2.1 Automatic Device Discovery

When two devices have to discover each other using Bluetooth, one of the devices transmit search requests, *inquiry*, while the other device has to be listening for these requests in the same time period, *inquiry scan*. The modes are complementary, so it is not possible for a device to do both at the same time.

The actual discovery mechanism is a complicated procedure because of energy considerations and the lack of a global synchronized clock. Bluetooth is designed such that the energy burden is placed on the part doing inquiry. To save energy, the listening device will only listen for inquiries periodically. Without a synchronized clock it is not possible for the inquiring device to know exactly when, requiring it to transmit search requests continuously. The specification recommends performing inquiry for 10.24 seconds to assure that all devices are found [10, Part B, 10.7.3]. Practical experiments done with two Bluetooth units show that after 1.910, 4.728, and 5.449 seconds, the discovery was successful in respectively 50, 95, and 99 percent of all tests, 2.221 seconds being the average [40]. Similar experiments with different equipment reports 2.772, 4.184, and 6.311 seconds for the same percentages, and an average of 2.33 seconds [43].

These numbers are fairly large compared to the five seconds maximum we decided in Section 1.2. Making a device switch between doing inquiry and inquiry scan does not seem like a feasible scenario. Since the modes are complementary and there is no synchronized clock, both devices could end up being in the same mode at overlapping time ranges, where discovery would be impossible. Hence, the only way to minimize the discovery time, is to let one of the devices perform inquiry all the time.

As mentioned, Bluetooth is designed such that inquiry is the expensive operation while inquiry scan is cheap. The reason is that while an inquiring device broadcasts inquiry packets continuously during the entire inquiry period, a device doing inquiry scan with default parameters only listens for inquiry messages for 11.25 ms every 1.28 seconds. Quite a difference in radio usage, and thus energy usage. Kasten and Langheinrich [40] reports that inquiry is twice as expensive as inquiry scan, namely 200 mW and 100 mW respectively[1]. We have searched for specifications for other Bluetooth modules, but we have not found any with specifications for inquiry energy cost[2].

Even though we do not have good references for the actual power consumption, it is our opinion that it should not be the mobile device doing inquiries as it is energy constrained. If the Bluetooth chip used in the mobile device is properly designed, the power consumption of doing inquiry scans should be negligible compared to continuous inquiries. Our initial experiments also confirm this (see Section 5.2.2). Based on this, we choose to place the burden of the inquiries on the lock device.

2.2 Handling Multiple Devices

The Bluetooth devices with the shortest radio range have a range of 10 meters. This means that many Bluetooth devices could be in range of each other, especially in an office environment where mobile phones, printers, PDAs, etc. could be Bluetooth enabled. So as a result of an inquiry the lock device could have a list of devices, which could be mobile devices, irrelevant devices, or any combination of these. The lock device has to connect to each device to examine whether it is a mobile device and if it is, whether it wants to, and is allowed to, unlock the door. Even without any irrelevant devices, two users could ap-

[1]It is unclear from the article whether the inquiry scan consumption is a continuous consumption or only in the listening periods.

[2]There are 30 listed modules on `http://www.btdesigner.com/` and the ones with any specification of energy consumption only has numbers for idle mode and connections, none has for inquiry cost.

proach a door at the same time, both carrying a mobile device. Two mobile devices would then be in range at the same time, and it may be that only one of them carries a valid key.

Prior work has shown that establishing a connection takes around three seconds [7], and since connection establishment is handled solely by the Bluetooth hardware, the lock device cannot influence this. So any unnecessary connections will take a lot of time for the user, and it is thus crucial to limit these. Another important aspect is that any connection to a Bluetooth device takes power, and if the device is battery powered the lock device will use valuable energy on the given device with any unnecessary connection attempts. To summarize, the problem to be solved is:

> Minimize the energy consumption and latency for automatic connection establishment between the mobile device and the lock device, with multiple discoverable devices present.

2.2.1 Discovering and Connecting to Devices

The inquiry command in Bluetooth takes two arguments that determine when the inquiry is done (see specification in [10]): A timeout on the maximum amount of time to listen for answers (a modulo of 1.28 seconds) and the maximum number of device answers to listen for. But the inquiry can also be stopped manually at any time. It is not possible to try to establish a connection while an inquiry is running, so the lock device will have to do inquiry, stop the inquiry, and then try to connect to the device(s) found.

To get the fastest discovery time, the lock device can wait for the first device to answer and stop the inquiry immediately and try to connect to it. However if there are multiple devices in range the lock device will only see the first one answering, which may not be the right device to connect to. The lock device will then have to do inquiry again, and if it follows the same strategy the same device may be the first to answer again. Waiting a specific amount of time after the first device has answered is also a possibility, but the first device that answers can be the correct device and it may have been in range for a long time (up till 10.24 seconds according to the standard), so waiting could make an already long wait even longer.

Instead we propose for the lock device to keep a history of previously discovered devices, and use this to differentiate between the devices discovered during inquiry. The lock device can record whether a device has been discovered multiple times but has failed to respond properly to a connection request. The device is probably not a mobile device then, and the lock device will wait for further answers before finishing the inquiry. This will filter out any Bluetooth devices that either are irrelevant (f.x. a nearby printer) or are mobile devices not used with this lock device. A device will have to have more than one non-successful response before it should be tagged as an irrelevant device, as connection attempts may fail from time to time. The lock device also has to clear the history for a device periodically, as a device that may not have been used with a lock device might obtain the necessary keys or software. Waiting more than a day before being able to use newly obtained keys would be an annoyance to the user, while waiting less before clearing the history will lead to too many failed connection attempts. The lock device can then clear the history once a day, at a time where the lock device is not used frequently.

Multiple Devices

If two new devices enter the range of the lock at the same time, and only one of them wants to open the door, there is still a problem with the above strategy. The lock device stops inquiring after it has found the first device, tries to connect to it, which could either succeed or fail. If it fails this could either be because of a general Bluetooth connection failure, or that it is the second device that wants to unlock the door. The problem is that the lock device does not know that there is a second device, and in the case of a connection failure it cannot rule out that it is the first device that wants to open the door. Restarting the inquiry procedure might return the first device or the second device, either of them could be the wrong one, so inquiry might have to be done multiple times before the right device is found. This strategy is nondeterministic and might take a long time.

If the lock device, when it starts inquiring again after an unsuccessful connection attempt, inquiries until all mobile devices are found before starting to connect, the discovery procedure will have an upper limit. We do not have figures for inquiry time with multiple devices in range, but Bluetooth has various mechanisms to avoid collisions for inquiry answers. As an example, a Bluetooth device receiving an inquiry has to wait a random amount of time before answering it. Thus, for a limited amount of devices in range, the figures for discovering one device should apply for multiple devices too. This means that if the lock device inquiries for 6.4 seconds ($= 5 \times 1.28$) there is 99% chance of a device being found. Waiting the 10.24 seconds that the standard specifies is much too long, and with a 99% chance of finding all within 6.4 seconds the possible failure in one out of 100 cases of this special scenario is acceptable. Even in case of a failure, the lock device will eventually discover the device in a subsequent inquiry. All in all, we believe that although any second attempt takes time, it is better to set an upper limit on the discovery time. We therefore choose to let the lock device inquiry for 6.4 seconds after an unsuccessful connection.

If there are multiple devices found during the inquiry, the lock device has to choose which one to connect to first, since it is only possible to establish a connection to one device at a time in Bluetooth. The lock device will have to choose a device from the list of devices and try to connect to it. If the connection fails, it removes the device from the list and choose another one, and so on until the lock is unlocked or the list is empty. If there exist no prior knowledge about the devices, nothing else can be done but changing the order in which devices are chosen from the list. Since the inquiry procedure involves multiple random elements the order of the list will be nondeterministic, and only practical experiments can show whether for example random choosing is better than just taking the head of the list.

Over time the lock device can also use the history to choose which device to connect to first. If the lock device has the choice of connecting to a mobile device that has previously unlocked it, or another device that it has not seen before or has previous connection failures, it is clear that it should connect to the first device first. This use of the history is only used to prioritize devices not to disregard them totally, so for this purpose the history can be retained longer.

2.2.2 Limiting the Number of Devices in Range

To limit the number of possible devices to connect to we can also choose to limit the number of devices in range of the lock device, which can be obtained by adjusting the Bluetooth radio on the lock device. First, the range of the radio can be limited, since the user carrying the correct mobile device should be located close to the lock device. This can easily be done as it is just a question of decreasing the power to the radio or alternatively, if this is not possible, shielding the radio which would weaken the signal and thus decrease the range. The natural downside to this is that the shorter the range, the closer the user has to be before the mobile device can be discovered, increasing the time she will have to wait in front of the door. The optimal range will have to be found through experiments in an actual installation.

Second, the radio can be directional. If the lock device is located on a door that can be opened without a key from one side, there is no reason to try to connect to mobile devices located on that side of the door. Most doors in office environments and apartment blocks are like this. If the lock device is located right next to the door, our estimate is that it should be something like 160–180 degrees. But again the actual setting of the radius will have to be tested through experiments, and will depend on the placement of the lock device, the door, etc.

It is also possible to use features in Bluetooth to filter out irrelevant devices (i.e. Bluetooth devices that do not function as a mobile device in our context). One method is to set the *Inquiry Access Code* (IAC) used in the inquiry. The Bluetooth device listening for inquiry requests must also have the same IAC set, or else it will not answer. All mobile devices could then have a special IAC that they listen for. The problem is that the standard only defines two different IACs the *General* IAC (GIAC) and the *Limited* IAC (LIAC). The GIAC is the one normally used, while LIAC is used when a device wants to use *limited discoverable mode*. It is explicitly stated that this mode "is only intended to be used for limited time periods in scenarios where both sides have been explicitly caused to enter this state, usually by user action." [69, Section 1] and that a device "should not be in limited discoverable mode for more than $T_{GAP}(104)$" [11, Section 4.1.2.1] ($T_{GAP}(104)$ is 1 minute). Thus the mobile device cannot use the LIAC and still conform to the standard. Moreover it is likely that a Bluetooth device only supports one IAC at a time, which makes the mobile device indiscoverable for other uses, which could be unacceptable to the user. All in all, it is not possible to use this feature.

Another method is to use the data returned in an inquiry response. Besides information needed to connect to the device, there is also a *Class Of Device/Service* field (CoD) (use defined in [11] , content in [69]). The CoD is user configurable and is, as the name suggests, split into a *Bluetooth Device Class* and a *Bluetooth Service Type* part. If every mobile device has some common data in the CoD, the lock device can filter out any device without this data.

The *class* part defines the primary functionality of the device: a computer, a phone, a peripheral, etc. Although it is possible to configure the class part, it will influence all other contexts where the device is being discovered, which may not be satisfactory to the user. Secondly, there is no definition for access control purposes and if there was, it would not exactly be true to say that access control was the primary function of a mobile phone.

The *service* part of the CoD is a 11-bit field specifying which services are available on the device, with one bit for each of the service types: positioning, networking, rendering, capturing, object transfer, audio, telephony, information, limited discovery, and two reserved bits. None of the first eight bits fits directly to access control—although we do transfer an object (the key), it is probably to stress the concept. Of the three remaining bits, two of the bits are reserved so they cannot be used and the third can only be used in *limited discoverable mode*. So adhering strictly to the standard, it will not be possible to use the service class either.

The service class may still be used though, by a creative reading of the standard. It is only stated that the *limited discoverable mode* bit has to be set when the device is in limited discoverable mode, not that it cannot be set when the device is not in this mode. To our knowledge the LIAC is infrequently used, so the chance of other devices having this bit set is small. So all mobile devices could have this bit set, letting the lock device filter out any irrelevant devices in an effective and easy way. It is on border of being a hack, but it is our opinion that it obeys the standard. We choose to use this feature as it is an effective filter, and will save many unnecessary connections and thus valuable time.

Lastly, a method for avoiding a lot of unnecessary connections, is to make the lock device able to detect whether the door is locked or not—logically no connections should be done if the door is unlocked.

2.2.3 Context Awareness

Until now we have concentrated on how the lock device discovers and connects to the right mobile device, but the problem can also be tackled from the viewpoint of the mobile device: If the mobile device does not do inquiry scans the lock device will not detect it, which apart from making the connection establishment easier also saves valuable energy on the mobile device. In fact the mobile device does not have to be discoverable all the time, it only needs to be discoverable when it needs to unlock a door. The problem is that without user interaction, the device automatically has to detect when a door needs to be unlocked—it needs to be aware of its own context.

Context awareness is used for numerous mobile device projects like ParcTAB [77], Cyberguide [45], Smart-Its [63], TEA [26], and numerous others. Schmidt et al. [65] organize context into either *human factors* which relates to the users current tasks, environment, known habits, etc., or *physical environment* which relates to the current location, the resources at the location and the physical conditions (light, motion, etc.). It is beyond the scope of this project to do a full analysis of context awareness, but we will elaborate on two examples of the use of the physical environment in our scenario: location, and physical conditions. For an elaborate analysis of context in ubiquitous computing see [64].

Using Location

Knowing the current location will enable the mobile device to only be discoverable when it is near a known location of a lock device. How the mobile device knows the locations is a topic in itself, but a key could have door location information attached or the mobile device could learn the locations as it communicates with the lock devices.

The system that delivers the location information can either be a proprietary or an existing system. A proprietary system will need to be installed at all locations where lock devices are present, which will add extra cost and complications to each lock device installation. Moreover it also needs to be installed on the mobile phone. All in all we are not in favor of a proprietary solution.

Of globally available systems, there seems only to be two choices: GPS [33] or GSM [32]. GPS would give the mobile device absolute coordinates, but is still rather expensive and is not found in mobile phones today. Furthermore it does not work well indoors. GSM is the European digital mobile phone standard, which is used allover Europe[3] and is gaining market in the US. Each base station antenna in a GSM network has a unique cell ID, which can be retrieved by the mobile phone. By keeping a list of known cell IDs for given locations, these cell IDs can be used to deduce the location. Single base station antennas cover a big area though, especially in rural districts (multiple kilometers), and cell usage can depend on weather conditions, operator restructuring, etc. The result is large granularity and uncertainty so GSM cannot be used to distinguish between locks in an office environment. It may not even be used to distinguish between apartment blocks. However in many cases it can enable the mobile phone to detect whether the user is at home, at work, etc.

The location information offered by GSM can be augmented through services offered by the GSM operators. As an example, Ericsson has a service which provides a position with typically 200 to 300 meters accuracy in an urban environment, without the need for special hardware on the mobile device (see [16]). This increases accuracy, and thus the usefulness, of the system, but the general availability of these services are not known to us.

[3]There are still some analog systems like NMT in use, but it is being phased out. Moreover devices for analog systems tend not to have Bluetooth installed.

Using Physical Conditions

Another approach is to use physical conditions (following the terms from [65]) to obtain information about the context, by using sensors to obtain information about the physical environment and measure light, pressure, acceleration, temperature, audio, etc. A mobile phone in our scenario is pr. default equipped with three inputs that could be classified as a sort of sensors: the microphone, the GSM radio, and the Bluetooth radio. It is not immediately apparent what these could be used for in our context (except for the location information from GSM as explained previously). It is possible though that they can be used in conjunction with other sensor data to strengthen or weaken a given context recognition (i.e., *sensor fusion*).

An interesting sensor in our scenario would be an acceleration sensor: As long as a mobile device is stationary, and has been for a while, it would mean that the device is not carried around, and thus there is little chance that the user needs to open a door. Such a sensor has successfully been used in the TEA, Mediacup, and Smart-ITs projects (all described in [26]), among others in an experiment to make a mobile phone automatically change profiles depending on the context. The mobile phone was able to recognize the context with a certainty of more than 87%. Their only problem was that it could take up to 30 seconds before the context was detected and switched to. Still it shows that physical conditions can be detected in practice using simple sensors.

2.3 Final Design

Our choice for the design is that the lock device will do endless inquiries, and a mobile device must do inquiry scans and have the *Limited Discoverable Mode* bit set in the CoD. Ideally only valid mobile devices should be returned from the inquiry, since all irrelevant devices should have been filtered out, either by the history or by the CoD.

The procedure will find the mobile device within 2.3 seconds on average (based on previous experiments), or if the first connect fails, in average time plus 6.4 seconds in 99% of the cases. The same applies if multiple devices are present and the mobile device is not the first to answer. In the latter case it will also take time to connect to any other device found during the inquiry, if more than two new devices has entered the range at the same time.

The discovery time directly affects the waiting time in front of the door, but it also depends on the radio range of the lock device as discovery starts as soon as the mobile device is in range. It is reasonable to assume that a normal walking pace is 5 km/h, or 1.4 m/s, so for each meter of added radio range, 0.7 seconds can be subtracted from the waiting time. Even with a short range of only 4 meters, the device will be discovered before the user reaches the door. In the special inquiry case, the average discovery time will be 8.7 seconds (6.4 + 2.3), so the range will have to be 12.5 meters for the same to happen. Besides the trouble of many other Bluetooth devices also being in range, we will later in this section discuss why this might not be feasible.

There are still some challenges in the use of the system, primarily related to the automatic unlocking of the door. First and extremely important, the system must not unlock a door without the user being aware of it. There must be some kind of feedback when the door is unlocked, preferably on both lock device and mobile device. On the mobile device, it can either be an audio signal or use of the vibrator function if the mobile device has one. The lock device can also use an audio signal, but the unlocking mechanism of the physical lock may have feedback mechanisms already, as it is common in many intercoms in apartment blocks. Still, the user can be unaware of the unlocking and unintentionally leave a door unlocked. For the system to be secure, the door should only be unlocked for a brief period of time, still leaving enough time for the user to open the door. A couple of seconds should be enough or until the door is opened if the lock device is able to detect this.

Another problem is how to avoid unwanted unlocking of the door, as the system will automatically unlock the door if the mobile device is allowed and is discoverable. There are three scenarios where a mobile device is in range, but where the user does not want to open the door:

1. The user is incidentally located near the door.
2. The user is leaving.
3. The user is passing by the door.

First, the user may incidentally be standing next to the door talking to someone else, which the lock device cannot detect. If the user has to open the door anyway the door can just be opened. If not, the user can either move out of range, or have some easy method to choose not to open the door on the mobile device, either permanently or for a limited period.

Second, a user does not want to unlock the door when she is leaving the area. A solution is not to try to unlock the door shortly after it has been closed. First, this assumes that the lock device can detect this, and secondly the user may want to unlock the door again immediately after locking it, because she might have forgotten something. Waiting a few seconds for this to happen will be tolerable though, so we propose that the lock device waits 5 seconds after the door is closed before performing inquiries again. This will allow the user to get out of range before the lock device tries to unlock the door again.

Third, the user may just be passing by the door. This could be solved by requiring the mobile device to be near the door and remaining immobile for a short time before opening the door. This could be done by a sensor on the mobile phone, or radio range measurements by the lock device. Unfortunately Bluetooth does not have explicit support for this, but it is possible for a Bluetooth module to support it and return an indication of the range. Either through the `Get_Link_Quality` command [10, p. 715] which is module vendor specific or through a vendor-specific command. We have not found any devices supporting this. So in our scenario this can only be solved by limiting the radio range of the lock device, minimizing the time a mobile device passing by is in range and thus the chance of successful discovery and connection setup. In Figure 2.1, we present the complete diagram for the state machine of the lock device including all above mentioned features.

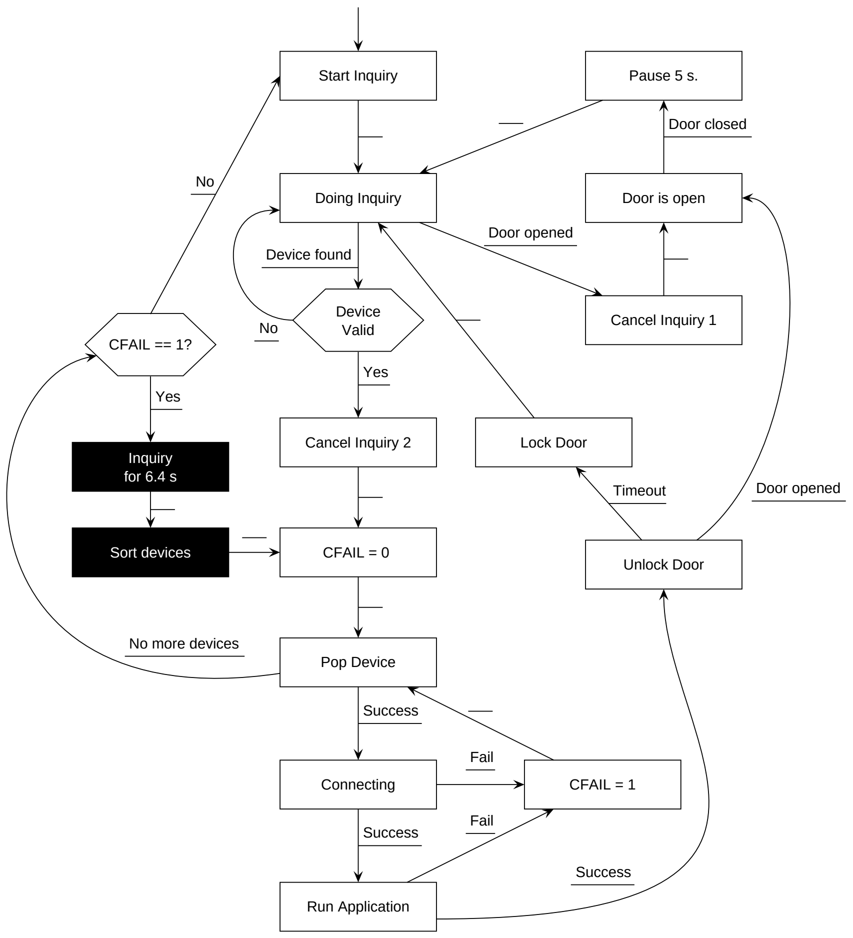


Figure 2.1: The inquiry and connection strategy for the lock device. The device performs constant inquiries until a valid device is found, and then tries to connect to it. If any connection or unlocking attempts fails it enters the special case where it does inquiry for 6.4 seconds to discover all devices (shown in inverse coloring on the graph). When the door is opened the device aborts inquiry, and restarts operation after a 5 second pause when the door is closed again.

The conclusion is that the radio range should be as short as possible. Presuming that the mobile device is the first to be connected to, the unlocking application 1.0 second, average connection time is 3.0 seconds, and average discovery time is 2.33 seconds unlocking totals 6.33 seconds. To achieve an average wait time in front of the door of 5 seconds, the mobile device will have to be in range 1.33 seconds before the user reaches the door, which gives a minimum radio range of approximately 2 meters. Observe though that this is only average timings, guaranteeing a maximum wait time is not possible. Moreover, if the first attempt fails the average waiting time will be longer.

An average person walking past the lock device will pass that area in around 3 seconds, which in most cases will not be enough time to unlock the door. But with a slow moving person or fast discovery and connection establishment, unneeded unlocking will occur. We cannot influence the Bluetooth part, but tightening the unlocking application part to 0.5 seconds will decrease the minimum radio range to a bit more than 1 meter. The unlocking application should at a maximum take one second then, preferably less. We will analyze the requirements for this in the next chapter.

2.4 Relay Attack

There is one security vulnerability in this scenario that allows someone to mount an attack on the system, a *relay attack*. The attack allows attackers to unlock the door with a mobile device of a user, when the user is far from the door itself. The user will preferably be notified of the unlocking (see previous section), but being possibly far away from the door this poses a risk.

Mounting the attack will require two attackers working together, one following the mobile device (L_{fake}), and one at the lock device (M_{fake}). The attackers can then either amplify the signals from the device radios or relay the communication through another communication channel with longer range. The easiest way to mount the attack is through the use of two Bluetooth devices, where it is possible to set the Bluetooth address. L_{fake} will set its Bluetooth address to the address of the lock device, and M_{fake} will set its Bluetooth address to the address of the mobile device. M_{fake} will be discovered by the lock device, which will then connect to it. L_{fake} will connect to the mobile device, acting as the lock device. When both connections are established M_{fake} and L_{fake} will relay all traffic over another communication channel, for example a GSM data link. There is no detectable difference in the Bluetooth communication for the real devices, and as the attack requires no knowledge or alteration of the contents of the communication it can be mounted against any application.

The problem is that the lock and mobile device use an untrusted communication channel (the atmosphere) and has no idea about the physical distance between them. This can be solved by disallowing door unlocking to happen if devices are located far from each other, that is, more than a couple of meters. This needs either absolute or relative positioning. If both devices have access to absolute positioning and transfers their position to the other part securely, they can abort the communication if the distance is to big. The lock device can have its position saved once, but the mobile device needs to obtain its position continuously and with an accuracy far exceeding that of GSM, so that is not possible.

Relative positioning, by measuring the time it takes for the radio signal to travel from sender to receiver, cannot solve the problem, even if the Bluetooth devices supported this. One attacker will be located right next to the lock device, and if the other attacker is located at roughly the same distance to the mobile device, the distance to the communication partner observed by the two devices will be correct. So the distance has to be assured via the application. The problem is, that the application on the lock device will have to be able to measure the round-trip time difference between a packet sent over 1 meter, and one sent over minimum 10 meters. That is the difference between $\frac{2m}{3.0\times10^{8}\frac{m}{s}} = \frac{2}{3}\times10^{-8}s$ and $\frac{20m}{3.0\times10^{8}\frac{m}{s}} = \frac{2}{3}\times10^{-7}s$, a difference between 10 and 100 nanoseconds! First the lock device would need a timer capable of measuring this and secondly it would need complete knowledge of timings in the Bluetooth stack in both itself and the mobile device. Theoretically this might be possible for the lock device, but the mobile device could be any type of hardware where this information will be impossible to obtain. Thus this is not an option either.

However, with some user interaction we can limit the attack. We propose that the user needs to actively turn on the unlocking function of the mobile device, but that it automatically turns off again after a user defined period. The period could either be the time since unlocking was enabled, or the time since the last unlocking. This will not guard against the relay attack, but only make the device susceptible to the attack when it is being used—not when the user is sleeping, on a break, etc. It does however need user interaction, so the choice of using this must be left to the specific user.

All in all, we unfortunately have no solution on how to guard the system against this type of attack in our context. This can only be solved by measuring the distance which we cannot, or by the user acknowledging the unlocking which is not what we intended.

2.5 Conclusion

In this chapter we have presented a method enabling automatic connection establishment between a lock device and a mobile device using Bluetooth. We have tackled the challenges in doing this without user interaction, and in an energy efficient way.

The procedure will achieve an unlocking of the door in less than five seconds on average. In cases where multiple mobile devices enter the range at the same time, and the one who answers first either does not have the right key or connection setup fails, the wait will be at least twelve seconds. The reason for this is the combination of Bluetooth and the required automatic unlocking.

Is Bluetooth then suitable for this scenario? It is not a perfect match, but it works. The main problem with Bluetooth is the slow discovery and connection establishment. There is not much room for error when the average time for doing both is more than five seconds, and we for other reasons have to limit the radio range to two meters. An improvement of these critical steps in Bluetooth would be of great help.

A feature that could be of great value for us, that Bluetooth unfortunately does not have, is a *limited discoverable mode* based on the Bluetooth address of the inquiring device. If a mobile device would know the Bluetooth addresses of all the lock devices it has the possibility of unlocking, it could limit inquiry answers to those devices (if the user allows it) and save unnecessary connection attempts.

Many of the challenges we have faced in this section originates in the need for automatic unlocking. While this is a valuable feature and makes the scenario easy to use, it might pose too big a problem when based on Bluetooth. Especially the relay attack is troublesome, but:

> In many cases, middleperson attacks are possible but not economic. In the case of car keys, it should certainly be possible to steal a car by having an accomplice follow the driver, and electronically relay the radio challenge to you as you work the lock. But it would be a lot simpler to just pick the driver's pocket and mug him. ([4], page 20.)

So whether this attack is a real problem in a practical scenario is an open question. But as long as the user is notified about every unlocking, both successful and unsuccessful, the user will be aware of the attack and will be able to take action.

In the following chapters we will analyze how to build a secure but fast protocol supporting our scenario. The protocol procedure must take less than one second, but preferably be as fast as possible to minimize waiting time or radio range. Further, we will measure the speed of doing inquiry and connection establishment to verify the average speed with our specific equipment. We will also measure the energy cost of doing inquiry versus inquiry scan, to test our assumption of the former being expensive as opposed to the latter.

CHAPTER 3

Secure Access Control

Complex systems are less secure than simple ones, guaranteed.
– Bruce Schneier, Secrets & Lies [67]

In the previous chapter we analyzed how the mobile device and the lock device automatically discovers each other and establishes a Bluetooth connection. In this section we will analyze how to design a secure access control system that fulfills the project's requirements on top of this connection. The overall design is based on the ideas in previous work (see [12]).

We will start by presenting the basic design of our protocol and discuss the specific security requirements for the protocol. We will then analyze and design a protocol that incorporates these requirements. Afterwards we will discuss which cryptographic algorithm is best suited for the system and analyze the use of the security features in Bluetooth. After that we will present the exact requirements for both devices, and finally consider two extensions to the basic scenario (presented in Chapter 1).

Throughout the analysis we will try to adhere to all the advises given in [1, 67], but will pay special attention to two of them: First, using well-known and well-tested security mechanisms whenever possible. Opposed to building our own, this enables us to build on the existing expertise and knowledge invested in these by the security community. Second, keeping the system as simple as possible. Adhering to this will make the system not just easier to implement but also hopefully easier to secure, as discussed in [67].

The main problem in this chapter is to:

> Design a secure protocol that allows a *mobile device* with a valid key to unlock a door. Keys are created by an *external authority* that cannot communicate with the *lock device* that handles the unlocking of the door. The protocol must fulfill the requirements given in Section 1.3.

An important requirement is that the protocol must work on resource constrained devices (limited CPU, memory, and energy). This means that the system should only incorporate the absolutely required features. This also aids in keeping the system simple, and thus hopefully more secure. Another goal is to make the protocol independent of the underlying communication system. If Bluetooth should fail to fulfill the requirements or another technology is found more suitable or gains wider acceptance, the protocol needs be easy to transfer to a different communication medium. So even though we have chosen to use Bluetooth, the protocol should not depend on it.

3.1 Basic Design

We will start by defining what a key is. A key is normally an object that in itself allows the holder to unlock one or more locks. We will redefine this slightly in our system, and propose two distinct characteristics for a key in our design, an *access key*: an access key is not secret and is bound to an *identity* (what an identity is will be analyzed later). This means that an access key will still allow the unlocking of one or more locks, but it cannot be used alone. To unlock a door, the mobile device needs both a valid access key for the specific lock and the matching identity. This eases the problem of creating and distributing keys, because access keys can be transfered unencrypted by untrusted channels and can only be used by the mobile device that is authenticated as the right identity (i.e. *authentication*, which will be discussed later). Moreover this also makes the information required to prove the identity the only secret information in the system, and thus the only information that needs to be guarded with care.

The following structure constitutes the baseline for our system, and this is the structure that we will analyze in depth in the following sections to make it secure.

1. The lock device connects and presents itself to the mobile device.
2. The mobile device sends an access key and proof of identity to the lock device.
3. The lock device unlocks the door if the information is valid, and informs the mobile device of the result.

The first step is needed for the mobile device to know which access key to use, and in the second step it transfers the necessary information. The third step provides feedback and logging possibilities for the mobile device. Each of the steps will be mapped into a concrete packet type in the protocol: `LockGreet`, `MobileCred`, and `Message` accordingly. The structure and contents of these will be analyzed in the later sections. However, if a mobile device is contacted by a lock device when it is not interested in unlocking the door or does not have a valid access key, it must also be possible to abort the protocol after step 1. We might need to add extra steps or packet types to make this possible, but our goal is to keep as close to the above described structure as possible to keep the system simple.

3.1.1 Existing Protocols

Building secure authentication on top of a loss-less data connection is a problem that has been solved before. The three commonly used protocols are TLS [17], IPSec [75], and SSH [6]—all three offer the same functionality, but are used in different contexts. However they are all general protocols allowing secure and authenticated communication of any kind to take place, offering a multitude of different systems for authentication and encryption. Many features that are not necessary in this simple scenario, which takes up space and makes them complex to implement. As an example the establishment of a TLS connection requires a minimum of eight packets. Moreover the transfer of the access key information needs to take place on top of these protocols, as none of them support access control, only authentication. Systems like Akenti [52] that also use asymmetric cryptography for access control purposes also builds on top of an existing secure transfer layer.

Commonly used systems are made for more general purposes, and are not designed for the constrained hardware. By designing our own protocol it will exactly match the requirements. This however introduces the risk of security vulnerabilities. Keeping that in mind, we still believe that a simple custom protocol that allows simple implementations of both lock device and mobile device is best suited to this system. The simplicity will also help reduce the possible security risks in a custom protocol.

3.2 Security Needs

In this section we will analyze what sort of security is needed in the system. We choose to divide security into the following classes:

Confidentiality: Ensuring that data can only be read by the intended recipient.

Integrity: Ensuring that data is not inserted, modified, duplicated, reordered, or replayed.

Availability: Ensuring that a service is available when needed.

Authentication: Establishing the identity of the communicating parties.

Non-repudiation: Ensuring that it is not possible for a party to send or receive data, and deny it later.

Other classes can be defined and used, but these are the most important and are commonly agreed to be part of the term *security*[1].

Every added security measure will increase the complexity of the system and most probably increase the resource usage. This directly contradicts the goals of the design, so we need to analyze exactly which security classes are needed in the protocol and only include these. This is one of the overall challenges in this chapter.

In the rest of this section we will analyze exactly what security classes are needed and whether to use asymmetric or symmetric cryptography.

3.2.1 Authentication and Integrity

Of the five classes of security described above, two classes are an absolute requirement for the mobile device. The lock device will need to assure that the identity stated in the access key information is identical to the mobile device identity. In other words it will need to *authenticate* the mobile device. This is the main problem that needs to be solved by the protocol.

To be able to trust the information received by the mobile device, the lock device will also need to make sure it is not altered in the transfer from the mobile device. It has to assure that the packet *integrity* is secured. This will also be required to assure that an attacker cannot reuse previous authentication information in a replay attack. Hence data from the mobile device needs authentication and integrity.

Lock Device

Whether the lock device needs to be authenticated, allowing the mobile device to be sure of the identity of the lock device, is not clear. The first problem is to establish the identity of the lock in our scenario. When there is no user interaction, the mobile device does not know the correct identity of the lock device that it needs to open. Even with user interaction, the user would have to confirm that the lock device connecting to the mobile device, is the lock device attached to the door she wants to open. To do this she must be presented with information that allows her to verify that, for example a number or common name for the door. This has to be verified at each unlocking which will be tedious, and most probably be ignored by many users. Hence the mobile device can authenticate the lock device, but it cannot use this to confirm that the lock device is the one on the door that the user is near. A *relay attack* can still be mounted (see Section 2.4). Authentication can still be used to differ between lock devices with a valid identity, and fake lock devices. The question is whether there are any reasons to pose as a fake lock device.

A fake lock device could keep connecting to the mobile device to try to drain the battery, exploit errors in the software to crash the application, keep it from connecting to a real lock device, etc.—just to be a nuisance or to make

[1] For a further discussion of this and a general introduction to security and security issues please see [4, 72].

a Denial of Service (DoS) attack. But except for errors in the software, all this can be done without any knowledge about the protocol, as any Bluetooth communication takes energy as the radio has to be turned on and packets received. Furthermore, making a DoS attack is quite easy in the wireless world as an attacker can always jam the spectrum with a radio transmitter.

Another reason to pose as a fake lock device could be to obtain access keys and identity information from the mobile device. But access keys are not secret and we see no reason for the identity information to contain secrets either. Even with authentication, the same attack would still be possible by listening to the communication with a valid lock device, as long as the communication is not encrypted.

An active version of the *relay attack* can also be done where the fake lock device can parse and alter the packets as it sends them between the two real devices (both types of attacks are called a *man-in-the-middle* (MITM) attack). This will not pose a greater security problem than the *relay attack*, changing the packets will only make the unlocking fail because of the integrity check.

Integrity for the messages from the lock device cannot be guaranteed without authenticating the lock device, so if integrity is needed this could be the reason to require authentication. We see no reason to assure the integrity of the `LockGreet` packet. If something is wrong with the packet, the door will not be unlocked posing no security risks. Integrity is needed to be able to fully trust the contents of the `Message` packet though. Whether a door is unlocked or not does not depend on this packet as it is only an informational message. As we see unlocking as the main problem of the function, we believe that the complexity needed to guarantee both authentication and integrity for the lock device is too much for an informational message.

We do not have any absolute reasons for authenticating the lock device or securing integrity for the messages. Furthermore, the mentioned attacks are all limited by the radio range of the mobile device, so an attacker needs to be near the target. We believe that the gain of authentication or integrity will be relatively small compared to the extra computation and complexity it involves. A consequence is that the user should be notified of all connection attempts, not just successful ones, so the user is made aware of possible attacks and can act accordingly.

3.2.2 Confidentiality

There is no need for *confidentiality* to guarantee the other security classes, it will only be needed if the system needs to transfer information that is deemed sensitive by either party. The only interesting information transfered is the identity of the mobile device, and possibly the identity of the user carrying it, which might be considered private information. If that is the case, there is all the more reason to require confidentiality since our scenario uses wireless communication, where eavesdropping is easier than in a wired communication scenario. However confidentiality implies encryption, which will increase hardware demands or needed time and system complexity (see [1]). Second, the Bluetooth address will always be public so it will also be possible to deduce the user from this. Third, confidentiality is implemented by encrypting the communication between the mobile device and the lock device. To encrypt the communication to the lock device the mobile device needs to establish a shared secret with the lock device. But to guard against MITM attacks the lock device needs to be authenticated, which we have found not to be feasible. So it is not possible to guarantee confidentiality.

3.2.3 Non-repudiation

Whether *non-repudiation* is needed to obtain a secure access control, is a matter of definition and context. As for confidentiality it is not needed by the other security classes. Non-repudiation can be used by the lock device, or rather the owner of the lock device, to prove that an identity has been used to open the lock at a given time. So from the viewpoint of the lock device, non-repudiation is a positive thing. This is not necessarily true for the mobile device, as non-repudiation can be used, legally or not, to prove the whereabouts of the mobile device. So it seems that non-repudiation is a question of either security or privacy depending on the viewpoint, and should be an optional feature.

3.2.4 Availability

The question of *availability* cannot directly be divided into an absolute or optional need. If the system is not available, it does not matter how secure it is. However, availability issues, most prominently DoS attacks, are often handled by authentication of both parties in a communication which will possibly increase hardware demands and complexity to much for our scenario. Moreover a DoS attack is always possible in a radio communication system, by simply jamming the used radio spectrum. Furthermore as long as the system is using Bluetooth, it is always possible to do endless connection establishment attempts, thus blocking the device discovery and connection for other devices. We will consider the availability issues throughout the design, but keep in mind that our application builds on top of systems where DoS attacks are easy.

3.2.5 Symmetric or Asymmetric Cryptography

When it comes to implementing the security classes, a fundamental choice will have to be made: Whether to use symmetric or asymmetric cryptography. Briefly, in symmetric cryptography there is only one key that is used for both encryption and decryption. In asymmetric cryptography two keys are used, a *public key* and a *private key*, constituting a *key pair*—it is not possible (in reasonable time) to calculate one key from the other. Data encrypted with the private key can be decrypted with the public key, and vice versa. Normally, the private key is, as the name suggests, kept private while the public key can be freely distributed. For a further introduction and discussion of symmetric and asymmetric cryptography see [48, 66, 72].

Symmetric algorithms has some features which makes them very practical for our scenario. In comparison with asymmetric algorithms they can use relatively small key sizes, are very fast, and can be implemented in very little space both in terms of code and storage space. So based on the constrained hardware in our system, symmetric algorithms seems to be the logical choice. Several systems offering both integrity and authentication has been designed using symmetric communication. A good example is SNEP [58], that provides these two classes for sensor nodes with very limited hardware.

As we see it, there are two major problems with using symmetric key cryptography in our scenario. First, the mobile device needs to establish a shared secret with every party that it needs to communicate with. So in order for the lock device to authenticate the mobile device, it must have exactly the same information available as the mobile device. In other words, when the lock device can authenticate an identity it can also use it, that is, identify itself to other lock devices with this identity. This means that the user of a mobile device will need to fully trust lock devices not to misuse the mobile device identity. Otherwise the mobile device needs to have a unique identity for each lock device. The latter seems to destroy the purpose of having an identity, as the external authority will need to certify each separate identity and the amount of data that needs to be kept secret increases significantly. Using asymmetric cryptography the system can still be kept very simple, while placing no trust requirements in the lock device.

Second, to verify access keys the lock device also has the information needed to create these. A attacker getting access to read the information stored on a lock device (a *passive attacker*), can create valid access keys for the lock. This makes it possible to get access to a door, without changing anything on the lock device. With asymmetric cryptography the lock only needs to be able to verify access keys, not to create them.

These arguments are strong enough for us to choose to base our system on asymmetric cryptography, even though it may be challenging to implement on the constrained hardware the system is assumed to run on. Another consequence is that the security features built into the Bluetooth protocol itself cannot be used, as it only supports symmetric cryptography (see [10, Part B, section 14]). But we believe that this decision ultimately will make the system easier to use and more secure.

3.2.6 Summary

All in all, the system will need to implement authentication and integrity for the mobile device. Optionally non-repudiation should also be supported but only if it can be incorporated with no or limited increase in hardware demands and complexity. Moreover the system needs to use asymmetric cryptography.

3.3 Protocol Design

In this section we will analyze and design the protocol for the system. That is, fully develop the structure given in Section 3.1, using asymmetric cryptography and provide authentication and integrity for the `MobileCred` packet.

To accomplish this while still making the system small, simple, and fast is the challenge in this section. We will start by analyzing how the mobile device is authenticated, then the structure of the access key, and lastly the communication with the lock device.

3.3.1 The Mobile Device Identity

The lock device needs to establish the identity of the mobile device, and check whether it is allowed to open the door. The lock device may never have communicated with the mobile device before, and as a starting point only trusts the external authority.

For authentication each mobile device has a unique key pair installed, and the mobile device is identified by a token, I_M, that (uniquely) identifies the public key. To authenticate itself to the lock device, the mobile device has to send I_M and its public key to the lock device and then prove that it has the private key. The easiest way to design I_M is to make it identical to the public key. Or to save space, take a secure hash of the public key (using for example SHA-1[53]) and use that as I_M. This will generate a token that is unique and created directly from the public key. The problem with this is that it is not possible to have any additional requirements attached to I_M: validity ranges, usage restrictions, or simply information about the owner of I_M. It moreover limits I_M to be bound to one key pair which will not permit a mobile device to change key pairs while retaining the same I_M. It also limits usage scenarios where I_M is a role instead of a device and is shared by many devices and thus key pairs.

To tackle these issues we propose to use a document that associates I_M with the necessary information, a *certificate*. For the lock device to trust the information in the certificate, the external authority signs the certificate with its private key[2]. To validate the signature on the certificate and thus establish trust to the information about I_M and its public key, the lock device only needs to have the public key of the external authority. It is assumed that the key pair of an external authority is not changed within the lifetime of the lock device, so the public key of the external authority can be pre-installed on the lock device. The exact form of I_M can be decided by the external authority, it just needs to be unique and be stored in a separate field in the certificate. The size of I_M will influence both the certificate itself and also the access keys. We propose to use a SHA-1 hash of the public key, as it is both small and unique.

When it comes to the actual design of the certificates, we choose to use X.509 certificates [34]. X.509 is the *de facto* standard for making identity certificates, which makes it possible to use any existing external authority to create the certificates. The external authority, called a *Certificate Authority* (CA) in X.509, can either be established for this purpose only or be an existing CA used for example inside the corporation governing the lock device. It could also be a global CA like VeriSign Inc.[3] or a national CA like TDC[4] here in Denmark. If

[2]Making a digital signature on the document. In the following *signature* will always signify *digital signature*. For more information about digital signatures see [66].

[3]`http://www.verisign.com/`

[4]`http://www.certifikat.dk/` (in Danish)

any certificate mechanisms exist on the mobile device it is most probably for X.509, so using X.509 might make it possible to use existing certificate handling mechanisms. So using this standard adds a lot of benefits and flexibility to the design, and also follows our design goal of using standards.

There are two accepted storage formats for the certificates, *Privacy Enhanced Mail* [44] (PEM) and *Distinguished Encoding Rules* [36] (DER). They both offer the same functionality and it is possible to convert from one to the other, but they differ a lot in size. PEM, as the name implies, is meant for use in emails and is encoded following the requirements of SMTP [59]. The most important impact is that data is stored in 7-bit ASCII which naturally takes space. We have full control over the transport layer, and Bluetooth sets no such limits, so we choose to use DER encoded certificates which are generally 30-40% smaller than PEM encoded ones.

3.3.2 Authentication and Integrity

The actual proof that the mobile device has the private key is also handled by signatures. The mobile device takes the contents of the `MobileCred` packet, generates a signature, and appends this to the packet making the packet contain:

- C_M, identity certificate for the mobile device
- AK_M, access key for the mobile device
- Sig, signature

Where $Sig = s(C_M \parallel AK_M, KR_M)$, $\parallel$ is a concatenation function[5], KR_M is the private key matching the public key in C_M, and $s()$ is the signature generator. The lock device obtains and verifies the public key of the mobile device through C_M, and can then verify that the signature is made by the mobile device. This authenticates the mobile device. The problem is that anyone can reuse this packet. This can be avoided by using a challenge; the lock device sends a *nonce* N to the mobile device as part of the `LockGreet` packet. Nonce is short for "number used once" and is a (theoretically) unique number created for each packet. The mobile device concatenates the nonce, the certificate, and the access key, signs the result and uses this as the signature, that is, $Sig = s(N \parallel C_M \parallel AK_M, KR_M)$. The lock device does the same concatenation and verifies the signature. This assures both authentication, non-repudiation, and integrity, since no one else than the holder of the private key can make a correct signature for the packet, and it cannot be reused since the signature includes the unique nonce.

If there is sufficient room available on the lock device, it is possible to save transfer time by caching the certificate, so it only needs to be sent the first time or if the cached information is invalid. With caching, the lock device moreover only needs to verify the certificate the first time saving computation time also. The mobile device can be informed about cached information in the

[5]This concatenation is further specified in Appendix A.

`LockGreet` packet. For this use there needs to be a short unique identification for the certificate and the access key, we propose to use a hash of each for this purpose. This only needs to be generated once for each unique access key and certificate and can be pre-generated, so performance is not an issue.

The lock device does not know the mobile device identity when it sends it the `LockGreet` packet. Hence, caching presumes that the underlying communication layer provides some kind of identity information that can be used as cache lookup. With Bluetooth, this can be provided by the device address. If the identity information fails for some reason, the mobile device will detect the missing or wrong information and transmit what is required. The lock device cannot save transfer time when using a communication layer without this information, but it can still cache the access key and certificate and if the received information matches the cached information, it still does not need to verify the signatures.

The lock device can cache the access key, using the same arguments as for the certificate. However this could pose a practical security risk. We assume that the private key is only known by the correct mobile device, but if an attacker should obtain the private key, caching the access key would allow her to open the door without obtaining the appropriate access key. But our system is build on the premise that the private key is known only to the correct mobile device, and that access keys are not secret, so we choose also to cache the access keys in favor of decreasing transfer or verification time.

The above described procedure has a problem though: `MobileCred` is changed to $s(N, KR_M)$ as both C_M and AK_M are empty—the only thing being signed is the nonce. The nonce is created by the lock device, which means that a malicious lock device can get anything signed by the mobile device (see 3 in [5]). The same attack, in a more subtle version, can also be exploited even when the mobile device sends the certificate and access key. If the mobile device is tricked into using a specific access key and certificate (created by the attacker) and the attacker also controls the lock device she will have full control over what gets signed. Naturally this must not be allowed, so the signature needs to contain information that is determined by the mobile device. Of course the information also has to be known by the lock device or else it cannot verify the signature. In practice this will be solved by additional fields included in the signature, required for other purposes which are described later.

3.3.3 The Design of the Access Key

Two issues needs to be solved in the design of the access key. First we have to analyze how to specify which identity is authorized to open which doors. Second, we need to make sure that the information can be trusted by the lock device. The problem of access rights is a problem of how to specify these, so that the lock device and mobile device easily can find and use the needed access key, while still making the system as flexible as possible for the authority that governs the lock device. As ever in our scenario, the resource usage has to be considered also.

There is a large number of combinations of access rights that one could think of. As a start, an access key could specify access for:

- one mobile device to one lock device
- multiple mobile devices to one lock device
- one mobile device to multiple lock devices
- multiple mobile devices to multiple lock devices

But an access key could also be limited to a specific time frame, for example a key that could only be used Monday to Friday in the working hours, or a key that is only valid from October 1st to November 1st. We could either design our own access key structure with room for the needed information, or use a language to specify the rights in. An example of such a language could be:

```
IF DOOR == 123 THEN
  IF USER == 'Alice' AND DAY == ('SATURDAY' || 'SUNDAY') THEN
    IF TIME >= 12.00 AND TIME <= 17.00 THEN
      RETURN true
    ELSE
      RETURN false
    END IF
  ELSE
    RETURN true
  END IF
ELSE
  return false
END IF
```

Where this signifies an access key that allows the user `Alice` to unlock the door `123` at all times, except on Saturdays and Sundays where it is only allowed between 12.00 and 17.00. This allows one to specify the access rights in an extremely flexible way, should make it possible to create access keys that will fulfill most scenarios. An example of the semantics for such a language is KeyNote [9]. A more complete system is found in *Simple Public Key Infrastructure* [21] which defines structures for both authorizations, signatures, and certificates and also allows for the use of X.509 certificates. Unfortunately it is still in a draft stage, so work has still to be done before it can be used directly.

A problem with these approaches in general is that we have to be sure that the language semantically does not have security flaws, and that the parser is implemented correctly. Moreover the flexibility of the language can also increase the chance of user-made security problems. If the creation of a access key gets too complex, the creator may fail to understand the correct usage and make malfunctioning access keys. Another problem is that implementing a parser for the language on the lock device will make implementation unnecessarily complicated, and will take valuable resources from the lock device. Furthermore, making a system that satisfies every possible scenario is probably impossible. To follow our design decision of keeping the system simple when we can, we propose the following fixed structure:

Doors	One or more doors for which this access key is valid
Identities	One or more identities for which this access key is valid
Validity	Start date and end date where the access key is valid
Creator	Identity of the creator of the access key

This has the basic information we consider necessary, and still makes understanding, creation, and parsing of the information straightforward. The problem is that it only fits the uses that we have thought of, but we believe that this structure can satisfy most needs. The exact form of `Creator` can be chosen for the same reasons as I_M. And for the same reasons we propose to use a secure hash of the public key of the creator. The timestamps can be in standard UNIX time format (seconds since 1970) and `Doors` and `Identities` are lists of unique lock device (I_L) and mobile device identities (I_M). A technical description of the access key is found in Appendix A.

With the certificate, the access key, and the signature generated by the mobile device, the lock device will have to verify three signatures if a full packet is received. It might be useful to split `MobileCred` into a packet which only contains the certificate and the access key, and another packet which contains the rest of the required information including the signature. This will allow the lock device to start verifying the certificate and access key, while the mobile device generates the signature. This is an unnecessary complication if the computation time is negligible, so the decision depends the speed of the algorithms on both devices which will have to be tested in practice (we will analyze this in Chapter 5).

3.3.4 Communication with the Lock Device

As there are no requirements to secure the communication with the lock device, `LockGreet` should just contain:

- I_L, door id
- N, the nonce
- $h(C_M)$, the identity certificate
- $h(AK_M)$, the access key

Where the I_L is required by the mobile device to find the right access key to send, and $h(C_M)$ and $h(AK_M)$ are the hashes of the cached certificate and access key respectively.

3.3.5 Decreasing Authentication Time

The above scenario fulfills our design requirements but asymmetric cryptography takes time. Depending on the specific lock device and mobile device, the full protocol may take more than the one second limit found in Chapter 2, making the system unusable. We propose to only use asymmetric cryptography for the first connection and introduce a shared secret, S, that can be used for future communications.

After successful unlocking, the lock device has established trust to the mobile device, and knows that it has a valid access key. It can then generate S, encrypt it with the public key of the mobile device and include it in the `Message` packet. Since S is encrypted with the public key of the mobile device no one else can decrypt it, and S is only known by the lock device and the mobile device. The next time the lock device connects to the mobile device it can choose to let the mobile device sign the `MobileCred` with S instead of its private key. This will save both time and energy after the first successful connect between each unique device pair.

Assuming that the used symmetric algorithm is secure, the security is not weakened on the mobile device as long as the size of S has the same security level as the asymmetric key used (we will analyze this later), and S is guarded just as well as the private key. Naturally it is also important that S never has longer validity than the access key or the mobile device identity. The result will be a much faster unlocking of the door, as a symmetric algorithm will be magnitudes faster than its asymmetric counterpart.

The problem in this scenario is that an attacker can obtain all the necessary information needed to unlock a door, just by reading information from the lock device. To tighten security, S should have a limited validity so the full authentication procedure has to be done frequently. The exact lifetime can depend on calendar time, on number of uses, or both. Nevertheless, is it still less secure than using the full authentication procedure, so this feature should be optional and be allowed or disallowed on each device.

3.3.6 Handling Theft

In our design an access key cannot be used without the proper identity, so loosing an access key does not directly pose a security risk. The vital point is to secure the private key of the mobile device. As a starting point we must assume that the private key is only known to the mobile device, as the security of the system builds on this. Theft can happen though, and in this section we will analyze how to handle this.

We propose two ways to guard the private key. First, it should be secured on the mobile device. Ideally, once the private key is installed on the mobile device, it should never be possible to extract it and it should not exist anywhere else. Gutmann [28] reports on many of the security issues that arise if the private key is extractable. In the event of a loss, to software or hardware crash, theft, etc., a new key pair should be created. Preventing other applications from accessing the private key requires support from the operating system, the hardware or both—the application itself cannot guard against this. An ideal operating system will allow installation of the private key to a write-only store, that handles signature generation and verification for the application. This keeps the private key safe from possibly malicious programs coexisting on the mobile device, and possible bugs in the mobile device application.

Second, the private key can be password-protected. The application could require the users to enter a password to decrypt the private key before use, for example once a day, to limit usage if the mobile device is stolen. The problem is that the password needs to be entered on a keypad of a mobile phone and needs to be easy to remember, it will probably be limited to a few numeric characters. The mobile device can still be used until the password is required, and the limited password could easily be broken, but it takes more than just obtaining the mobile device itself. Again, this procedure should be supported by the operating system. If not, the private key could be copied to another device, the password could be broken, and the key could be used on the new device without limitations.

If the private key or the whole device is stolen, it is important to limit its use. The first problem is to detect the theft. If the mobile device itself is stolen detection is easy, but the private key may be stolen without detection. To detect unauthorized usage of the private key, the mobile device and the lock device should keep the time when a key pair is used with a specific lock device, a time stamp TS_{lu}, which is updated after each exchange. The lock device sends TS_{lu} in the `Message` and if it is not identical to the time stamp the mobile device has, theft must be assumed and a new private key created for the mobile device.

The problem is how to inform a lock device that a given mobile device identity is invalid. Normally this is distributed via *Certificate Revocation Lists* (CRLs), but as lock devices only communicate with mobile devices this is tricky (even if there were direct communication with the lock devices, there are still many challenges with CRLs, see [29, slide 42ff]). Distributing these securely through the mobile devices, and assuring that all lock devices receive them, would complicate the protocol and take (valuable) time and energy on both devices. Instead we propose to keep access key validity periods short. With access keys only being valid for example one week, an attacker could in the worst case only use the access key for one week, if the device gets stolen at the start of the period. After that period the attacker needs to obtain new access keys, which should not be created for the identity as soon as the theft is detected[6].

One last precaution that can hinder an attacker from stealing a private key and use it on another mobile phone, is to include the Bluetooth address of the mobile device in its identity certificate. The lock device can then control whether the address of the mobile device corresponds to the certificate, and disallow unlocking if it does not. This will hinder many attacks as most Bluetooth devices cannot change their address—we have however found one development board capable of this. The information can be ignored when not using Bluetooth, and will not make the design depend on Bluetooth.

Finally it should be noted that if something is stored electronically, it is possible for an attacker to obtain it if she has physical access to the device—there are unfortunately no means of making a device completely *tamper proof*. If somebody steals a mobile device and is determined to retrieve the private key it will be possible, no matter the software and hardware precautions taken. For more information on this issue please see [3, 4].

All in all, with all the above precautions implemented, even though a private key or mobile device is stolen the possible damage is limited.

[6]New access keys could be distributed by SMS for example.

3.3.7 Keeping Time

To be able to validate the timestamps and time limits on the certificates and access keys, the lock device needs to keep track of the time. We assume that validity checks that are off by even by a minute or two is acceptable for most purposes, so the precision of the clock does not have to be high. Any decent hardware clock can hold that precision for many years, if time was set on the installation of the lock device.

Hardware malfunctions or extremely long power shortage could make the clock loose time completely, and the lock device will need a way to recover. As the lock devices only communicate with the mobile devices, time information has to be received through these. This will not be a problem as most mobile phones are equipped with a clock. The problem is whether the information received from these can be trusted, as the clock could be set wrong, on purpose or not. An attacker aiming to extend the validity of an obtained access key would want to turn back time, or set the time wrong just to create havoc.

To guard against attackers the lock device can store the current time to a stable storage periodically, and in case of a malfunction restore time to the last stored time. Through this the lock device can obtain a base line, and will allow it to disregard any timing updates that would turn back time. Assuming that not all users are evil, the lock device can gather timing updates from a couple of mobile devices and use a combination of these to update the time. As an extra precaution the lock device could have a receiver for the DCF77[7] clock incorporated at only small cost, that allows clock updates to be received easily and works well indoors also. The DCF77 cannot be given full trust either, since it could easily be forged, but would nevertheless give a stable time source in most cases.

We choose to include a time stamp TS_{curr} in each `MobileCred` packet which, combined with a decent hardware clock, should keep timing information correct on the lock device. The actual algorithm used to combine the timing information to correct time in the most suitable way, is beyond the scope of this project. We refer to the Network Time Protocol (NTP) for further information and inspiration [50].

3.3.8 The Final Design

Adopting the notation used in [1], the final structure of the protocol is as follows (a complete list of symbols is found in Table 3.1):

$$\texttt{LockGreet} : L \rightarrow M : I_L, N, US, h(C_M), h(AK_M)$$

The lock device sends the mobile device its identity, a nonce, whether the mobile device can use a previously shared secret (US), and cache information.

$$\texttt{MobileCred} : M \rightarrow L : I'_L, I_M, N', C_M, AK_M, TS_{curr}, US, Sig$$

[7] `http://www.ptb.de/`.

Symbol	**Meaning**
M	Mobile device
L	Lock device
I	Identity
N	Nonce
US	Use shared secret?
$h(x)$	Hash of x
C	Certificate
AK	Access Key
KP	Public key
KR	Private key
TS	Time stamp
$curr$	Current time
Sig	Signature
$s(x, y)$	Generate signature for x with y
S	Shared secret
$v(x, y)$	Verify signature in x with y
RC	Unlocking result
lu	Last usage of KP_M with lock device

Table 3.1: List of symbols used in protocol descriptions.

The mobile device answers with the identity of the lock device and itself, both are included as they are essential to the meaning of the packet, and therefore should be included (see [1]). The nonce (N') does not have to be included in the packet as the lock device obviously knows its value and can concatenate it to the packet upon receiving and before signing. But considering the limited size of the nonce (see Section 3.4.3) and the possibilities of errors when including this before signing and verification on both sides, we include it anyway.

It also sends its identity certificate and access key, but may leave those two empty if they are reported to be cached by the lock device. It also sends its current time (TS_{curr}) and the signature:

$$Sig = s(I'_L||I_M||N'||C_M||AK_M||TS_{curr}||US, KR_M \text{ or } S)$$

It is generated with either the private key of the mobile device (KR_M) or the shared secret (S)—which one is set with US.

The lock device parses AK_M and verifies that the I_L is in the list of doors, I_M is in the list of identities, and that current time is within the date range. It further verifies that:

$$\begin{aligned} N &= N' \\ I_L &= I'_L \\ v(C_M, KP_{CA}) &= true \\ v(AK_M, KP_{CA}) &= true \end{aligned}$$

And lastly verifies Sig with either KP_M (obtained through C_M) or S depending on the contents of US. If all these are correct, the lock device unlocks the door.

$$\texttt{Message} : L \rightarrow M : RC, TS_{lu}, \{S\}_{KP_M}$$

The lock device sends the result of the unlocking attempt (RC), the last time the mobile device key pair was used with this lock device (TS_{lu}), and eventually a shared secret (S). This concludes the protocol interaction and the connection is closed afterwards. A technical protocol description can be found in Appendix A.

To guard against DoS attacks and malfunctioning devices it is necessary to set timing constraints (or timeouts) on each step in the protocol. Otherwise the devices could be locked in one stage of the protocol indefinitely, and block their use. The problem is that it is not possible to give exact constraints as it depends on the hardware of both devices. The sum of the timeouts should be no greater than one second to match the requirement from Chapter 2, so when the lock device hardware is fixed its maximum timeouts can be calculated. This can be used to find the timeouts for the mobile device and thus the minimum requirements for the mobile device.

3.4 Cryptographic Algorithms

To use asymmetric cryptography, we have to analyze two issues: The algorithm and the required key size for the given algorithm, as both of these have direct impact on both the speed and the security level of the system. We will start by looking at the algorithms.

RSA [62] and DSA [54] are two generally trusted and well-tested algorithms that has been used for a long time. In the recent years ECDSA [54] has also gained ground[8], and is regarded as at least as secure the other two (see [66]). There are also other asymmetric algorithms but we choose to only consider RSA, DSA, and ECDSA since they are considered secure and reliable, and has gained general acceptance as common standards by cryptanalysts, software developers and hardware manufacturers. This makes it possible to find both standard software and hardware implementations. They are all assumed to be equally secure, so what makes a difference for us is the size of the keys and the speed of signature generation and verification.

3.4.1 Key Size

The size of the keys depends on how long the information should be kept secret and on the algorithm chosen. Just a few years ago, 512 bit RSA keys was considered sufficient for most uses [48, 66], but recent studies show that 512-bit keys in theory can be broken in mere minutes [68]. This still has to be proved in practice though. Currently the recommended minimum key size is 1024 bits,

[8]DSA and ECDSA are two implementations of the *Digital Signature Standard* (DSS) [54], both using ElGamal [20].

both by *National Institute of Standards and Technology*[9] (NIST) and RSA Laboratories [70]. The current NIST recommendations for key sizes for all three algorithms are shown in Table 3.2. The key size influences both the size of a signature and the speed of both signing and verification, so the smallest key size that the security requirements allow should be chosen.

Level	Years	DSA (bits)	RSA (bits)	ECDSA (bits)
1	Present - 2015	1024	1024	160
2	2016 - 2035	2048	2048	224
3	2036 and beyond	3072	3072	256

Table 3.2: The NIST recommendations for asymmetric key sizes. It shows how long time information encrypted today, by a key of the given size, is believed to be protected. The "Level" definition is our own addition, the rest of the table is taken from [56, Section 8.8].

It seems reasonable to assume that an access key made today is not valid after 2015, so level 1 should suffice. The same is true for the mobile device identity. Recommendations for CAs are level 2 which is also the common practice. As the mobile device needs only to sign with its own private key, level 1 is required on the mobile device. Since the lock device may need to verify certificates signed by the external authorities, it also needs to support level 2. Supporting level 2 will also help prolong the lifetime of the lock device, which probably will be a permanent installation with no or limited possibilities for upgrades. The signature sizes for the three algorithms are [66]:

Level	RSA (bits)	DSA (bits)	ECDSA (bits)
1	1024	368	336
2	2048	*n/a*	464

3.4.2 Choosing an Algorithm

The problem in choosing an algorithm is that the speed is relative to the hardware used, and that there are no common rules for hardware requirements for the specific algorithms. So the choice is normally based on benchmarks, but there has not been focus on cryptographic algorithms on mobile phones, as it has not really been an issue yet. Personal communication with Peter Gutmann confirms this: "Usually the only benchmarks that exist are for crypto hardware (either smart cards or crypto accelerators) or faster general-purpose PCs and servers." [Mail exchange between author and Peter Gutmann on the cryptlib-mailing list, May 5th 2003]. All in all, this makes a general recommendation extremely difficult.

The closest benchmarks we have found is in [15], where the authors implement PGP [25] on a pager (10 MHz custom Intel 386), a Palm Pilot (16 MHz Motorola 68000-type Dragonball) and a PC (400 MHz Intel Pentium II). None of these is a complete match for the mobile devices in our scenario, but proba-

[9] `http://www.nist.gov`

bly somewhere between the pager and the PC[10]. Some of the results are shown in Table 3.3.

Type	System	RSA (ms)	DSA (ms)	ECDSA (ms)
Level 1				
Sign	RIM	15889	9529	1011
Verify	RIM	1008	18566	1826
Sign	PC	67	24	2
Verify	PC	4	47	4
Level 2				
Sign	RIM	111956	*n/a*	1910
Verify	RIM	3608	*n/a*	3701
Sign	PC	441	*n/a*	4
Verify	PC	13	*n/a*	8

Table 3.3: Algorithm timings on the RIM pager and the PC. From [15].

Signing with RSA is slow compared to verification, and will probably be too expensive to do on the mobile device, while verification is fast. DSA on the other hand show the opposite characteristics, with slow verification and fast signing. For our purpose this is good, as it is the mobile device, which is energy constrained, that needs to generate signatures. ECDSA is orders of magnitude faster for signatures, and is only a little slower than RSA for verification on the pager. These differences between the algorithms are very like the general characteristic described in for example [66].

Based on these results, it seems like ECDSA is the best candidate (which also was the conclusion in [15]). If the mobile device hardware is somewhere in between the pager and the PC as we envision, signing the `MobileCred` packet with level 1 will take well under one second, and may also even be feasible with level 2. Moreover key sizes for ECDSA is much smaller than for RSA and DSA (see Table 3.2) which makes both key pairs and all signatures smaller, leading to faster transfer time and smaller storage requirements. For the lock device, more powerful hardware is needed, as it in the worst case needs to verify three signatures (see Section 3.3.3). But all in all ECDSA is the best choice.

One problem with ECDSA is that it still is questionable how many certification authorities and cryptographic libraries support it. So a final decision will have to be based on realistic tests on possible mobile devices, compared with available support for ECDSA.

An issue with both ECDSA and DSA though, is that they are both part of the DSS standard which is a signature standard which limited to be used only for signature generation and verification. This is a problem as the system needs to be able to do both encryption and decryption to exchange the shared secret (see Section 3.3.5). There are two different solutions for this. First, both ECDSA and DSA uses a variant of ElGamal[20], so any implementation of ECDSA and DSA most probably also has support for ElGamal which can be used for encryption

[10]The Palm Pilot is a 16 bit processor, we assume that Bluetooth-enabled mobile phones use 32 bit processor.

and decryption. The problem is that depending on the software or hardware, the ElGamal functionality might not be accessible. This brings us to the other solution, which is to use ECDSA or DSA functionalities to perform ElGamal encryption and decryption. This is described in [66, section 20.1]. The problem is that to encrypt takes three signature generations and to decrypt takes two, so it is time consuming. Furthermore it might not work with all implementations. Our conclusion is that the optimal solution is to use software or hardware, that supports ElGamal encryption and decryption as well as ECDSA. As the functionality is already there to support ECDSA, this should not be hard to assure.

3.4.3 Shared Secret Signature

To make a signature with a shared signature there are two possibilities. Either to use a symmetric cryptography algorithm like AES [55] or to use a keyed *message authenticated code* (MAC) function like HMAC [42]. There are some theoretical security issues with using a keyed MAC (see [48, section 9.5.2] for a discussion), but it is assumed to be secure and is widely used, for example in TLS [17]. We choose to use HMAC-SHA1 (the implementation of HMAC using SHA-1) instead of for example AES, as SHA-1 is already used for caching, and HMAC-SHA1 poses only limited extra overhead on the devices. The recommended key-size for HMAC-SHA1, and symmetric algorithms in general, is 80 bits for level 1 [56]. We propose for the size of the nonce, N, to follow these recommendations also, as it should be at least as difficult to break.

3.5 Using Bluetooth

Even though we have chosen to make the design oblivious to the underlying communication layer, we still need to analyze how the communication should be setup. We will also analyze whether any of the security features in Bluetooth can be used as an option in our design.

3.5.1 Communication

When it comes to the actual transfer of data, Bluetooth offers many different communication facilities. The most basic packet types are ACL and SCO, where ACL is meant for data transfer and SCO for voice transfer. It is evident that we should use ACL as we are transferring data, and needs the loss-free connection that it features. So the choice regards which higher level protocol should be build upon. The lowest available user-level protocol in Bluetooth is L2CAP. On top of ACL it adds bigger packet sizes, reassembly of these, and channel multiplexing.

The other possibility is to use RFCOMM which is a serial port emulation protocol which uses L2CAP, and seems to be *the* choice for most Bluetooth profiles[11]. In addition to the services L2CAP offers RFCOMM offers serial port

[11]Exactly why is a bit of a mystery for us. For cable replacement, RFCOMM offers extremely

settings (baud rate, status bits, etc.) and flow control. We have no requirements for this, so L2CAP is sufficiently.

As L2CAP offers channel multiplexing each connection has a *Protocol/Server Multiplexer* (PSM), the equivalent of a port in the TCP-protocol [19], so the lock device will need to know which PSM to connect to on the mobile device. We can either follow the common practice of many TCP-based applications of "choosing a value that no one else seems to be using" or use the *Service Discovery Protocol* (SDP). The SDP is a query service running on a fixed PSM, that allows a connecting device to obtain a dynamically assigned PSM for an application on the host device, not unlike the *Port Mapper* service used for RPC [71]. Theoretically SDP is usable, but there are problems. First, both device needs to implement the SDP protocol, being general and rather complicated to implement and parse. Second, the transfer of the extra SDP packets and the data processing necessary will take valuable time. Unfortunately we have not found any measurements showing the added cost of using SDP compared to a direct connection to the service. But to avoid the added complexity of implementing SDP on both devices, we choose to use a fixed PSM.

3.5.2 Security Features

As previously mentioned, the security features in Bluetooth are all based on symmetric cryptography and is thus unusable for our main authentication purposes. But as we have seen in Section 3.3.5 the system can also use symmetric cryptography to decrease authentication time. Bluetooth offers authentication using an application-defined shared secret (up to 128 bits) [10, Part B, 14.]. It may be advantageous to use Bluetooth for this, as it is already present on the devices and may be faster than using our own techniques because it may be hardware implemented.

The use of these features presumes trust in the security mechanisms implemented in Bluetooth. Two weaknesses has been identified in the Bluetooth security system in [37], the first one concerning the problem of a MITM (see Section 3.2.1). The second addresses the problem that keys normally has to be entered on both devices manually which normally leads to a short key length. None of these are a problem in this scenario as the mobile device certificate guards against the MITM attack and keys are generated and exchanged by our application.

Then there is the security of the authentication system. Authentication is based upon the knowledge of the shared secret and utilizes the block cipher SAFER+ [46] (one of the contestants for the AES standard). Kelsey et al. [41] analyzes the SAFER+ algorithm and finds weaknesses in the 192-bit and 256-bit versions of the algorithm, but none in the 128-bit version used in Bluetooth. So to our knowledge there should be no security issues associated with the security algorithms in Bluetooth.

The actual authentication setup is handled purely by the Bluetooth devices. So if the lock device assigns the previously exchanged shared secret (S) to the

easy portability for existing applications, but for new applications it is a puzzling choice to build on top of an emulated serial port.

mobile device address and requests authentication for the connection establishment, it is only successful if the mobile device has the correct key. The lock device can then unlock the door and send an affirmative `Message` packet to the mobile device. To find out whether using Bluetooth for this is better than the previously described approach, will need to be tested in practice.

The only problem with this scenario is if the mobile for some reason has lost the shared secret. Connection establishment will fail, and the lock device has to try to establish a new connection without authentication, which takes time. Without using Bluetooth, the mobile device could just sign the `MobileCred` packet with its private key and authentication would succeed on the first try. We cannot see why this event should occur frequently so in practice it should have very little influence on the system as a whole.

3.6 Device Requirements

With the protocol structure defined we can now analyze its computation cost, and storage and transfer size. The storage and transfer size depend on:

1. the cryptographic algorithm chosen, as it influences key sizes and signatures
2. the size of the certificates
3. the format of I

First, we have chosen to use ECDSA and assume that the external authority uses level 2 and mobile devices use level 1, having a signature size of 464 and 336 bits respectively. Second, the size of the certificate depends on the form and amount of information that the external authority chooses to include, for example whether the Bluetooth address is included. Many certificates from commercial CAs include links to homepages, CA information, etc. We have not found any informations about sizes of ECDSA certificates but RSA and DSA certificates (with level 1 for identity and level 2 for CA) ranges from 700 bytes to 2500 bytes. Third, the format of I influences both AK, the protocol, and the certificate. We assume that they all have the size of a SHA-1 hash as proposed, which is 160 bits.

An access key will thus have the following size, assuming one lock and one mobile identity:

$$\begin{aligned} & TS_{start} + TS_{end} + I_L + I_M + I_{CA} + Sig \\ = & \quad 2 \times 32 + 3 \times 160 + 464\, bits \\ = & \quad 1008\, bits \end{aligned}$$

On top of this comes 10 bytes of overhead required to handle the variable sized fields (see Appendix A), making the access key total 136 bytes. This allows an access key to be transfered from the external authority to the mobile phone in one SMS message, as one SMS message can hold 140 bytes of data (see [23, 24].

3.6.1 Storage

This means that a mobile device needs to store:

Item	Cost (bytes)
Certificate	700 - 2500
Access Key	136

To unlock one lock, 826 bytes of storage is required with a 700 bytes certificate, and depending on the access keys either 136 bytes extra per door for each separate access key or 22 bytes per door with multiple I_L in one access key (160 bits I_L plus 2 bytes overhead). Respectively 13.3 Kbytes and 3.0 Kbytes for 100 keys.

As a minimum the lock device only needs to store the certificate for the external authority, 700 - 2500 bytes. For caching purposes, information for one mobile device takes 838 bytes.

If a shared secret is supported (see Section 3.3.5), the mobile device also needs to store 80 bits extra per door (see Section 3.4.3). This can be stored directly in the Bluetooth device if the Bluetooth device supports it. The lock device also needs to store 80 bits extra per mobile device for which it has set up a shared key.

3.6.2 Transfer

Under the same conditions as above, the size of the `LockGreet` packet is:

$$\begin{aligned} & \mathit{Hdr} + I_L + N + \mathit{US} + h(C_M) + h(\mathit{AK}_M) \\ = \quad & 32 + 160 + 80 + 8 + 160 + 160\, bits \\ = \quad & 600\, bits \end{aligned}$$

The *Hdr* is a fixed size common header on all packets (32 bits). On top of this comes a 2 bytes overhead for I_L because it has a variable size, bringing the total to 77 bytes.

The `MobileCred` is:

$$\begin{aligned} & \mathit{Hdr} + I_L + I_M + N + C_M + \mathit{AK}_M + \mathit{TS}_{curr} + \mathit{US} + Sig \\ = \quad & 32 + 160 + 160 + 80 + 5600 + 1088 + 32 + 8 + (336 \text{ or } 128)\, bits \\ = \quad & 7496 \text{ or } 7288\, bits \end{aligned}$$

The size difference lies in the different signature size for ECDSA (336 bits) and HMAC-SHA1 (128 bits). The overhead is 8 bytes, bringing the total to 945 or 919 bytes. If C_M and AK_M are cached, only 101 or 75 bytes are required.

And `Message` is:

$$\begin{aligned} & \mathit{Hdr} + \mathit{RC} + \mathit{TS}_{lu} + \{S\}_{KP_M} \\ = \quad & 32 + 8 + 32 + 336\, bits \\ = \quad & 408\, bits \end{aligned}$$

An encryption with ElGamal has the same size as a ECDSA signature, and there is no overhead in the packet so the total packet size is 51 bytes.

So the maximum amount of data transfered is 1073, and the minimum is 203 bytes if both C_M and AK_M are cached and a shared secret is used for the signature.

3.6.3 Algorithm Support

Besides storage capacity for the above the mobile device has to be able to generate one signature, and the lock device will have to verify three signatures (`MobileCred`, AK_M, and C_M). This has to be done in one second to fulfill the demands identified in the last chapter. If some extra waiting time is tolerable and caching is possible the first time a new C_M or AK_M is used, the three verifications for the lock device decreases to one verification after the first exchange of new information. The lock device furthermore has to support X.509 certificate verification and parsing.

To establish a shared secret, the lock device also needs to perform an encryption and the mobile device a decryption. The latter is not time critical though as it can be done after the unlocking is done. And the support is optional for both devices. They furthermore need to support HMAC-SHA1 to generate and verify signatures with the shared secret.

To support caching the devices should be able to perform SHA-1 to generate $h(C_M)$ and $h(AK_M)$. This is not a requirement though as the mobile device can always send them, and the lock device can still save signature verifications by just comparing the full contents of both.

All in all, the minimum demands for the devices is support for ECDSA signature generation on the mobile device, ECDSA signature verification on the lock device and X.509 support on the lock device.

3.7 Extending the Protocol

Until now we have focused strictly on the scenario and context given in Chapter 1. In this section we will describe two different approaches that extends the use of the protocol.

3.7.1 Multiple Authorities

Until now we have only considered a context where there is one external authority governing a lock (a *CA*), but it may be useful to enable more than one authority to create access keys. This will allow whoever is governing the lock and *CA* to delegate the responsibility of creating and administering keys to a third party, that is, *outsource* it. For mail delivery, it may be useful to allow the national mail company to create keys for its employees. The third party could also function as a lock smith if the mobile device or key is lost.

This can be done by permanently installing multiple certificates on the lock device, but this requires absolute and permanent trust to these authorities given that there is no direct communication with the lock device. Another approach is to create a trust hierarchy, where CA authorizes other authorities to create access keys. In practice this means that CA makes a certificate for another authority CA', which the mobile device can transfer to the lock device along with its access key and identity certificate. This enables the lock device to do $v(CA', C_{CA})$ and then $v(AK_M, C_{CA'})$. This approach means that only one certificate needs to be installed on the lock device when it is installed at the door, and any extra authorities can be enabled later. It also means that CA can grant access key creation privileges to CA' for a limited time period, for example one or two months, instead of giving a permanent carte blanche. The same approach can be used to allow CA' to create identities also, so the lock device can do $v(I_M, KP_{CA'})$.

If this feature is needed the changes to the protocol will be minimal, the only change is that instead of just sending C_M it should be possible for the mobile device to send a list of certificates. The lock device will need better support for handling the certificates though. The entire path through the hierarchy will need to be checked, so that every document in the hierarchy from the C_{CA} to the AK_M or I_M is valid. The mobile device will also have to know which and how many certificates to send. This can be set explicitly for each key and identity or the mobile device could have limited support for parsing X.509 certificates. Caching will still work by defining that the $h(AK_M)$ or $h(C_M)$ in `LockGreet` means that not only the given AK_M or C_M is cached, but also all necessary certificates.

Each certificate will need to have restrictions on their use, or else it would be possible to create access keys with a identity certificate, that is, $v(C_M, C_{CA}) \rightarrow v(AK_M, C_M)$. These restrictions are standardized in X.509 version 3 certificates. Note though that even though a certificate is restricted to create identities and not access keys, it can create matching identities for any valid access key. So in practice trusting a CA' to create identities, will also allow it to grant access to any mobile device. This of course presumes that the identities are not bound to a specific key pair, as proposed by $I_M = h(KP_M)$.

All in all, with one small extension to the protocol the system becomes much more flexible. It will increase hardware and software demands on both devices, so we will leave it out of the proposed system. But we do believe that this scheme should be considered in any real implementation because of the flexibility it gives.

3.7.2 Other Communication Scenarios

Another possibility for extending the protocol, is to imagine other communication scenarios than just mobile device and lock device communication. An example could be for one mobile device to be able to exchange access keys with another mobile device: One device could store backups of access keys on other mobile devices, or a user could receive access keys for a group of mobile devices and distribute them.

The same mechanism could be used if a mobile device could create access keys for a lock device. This would allow a user to use her mobile device to issue temporary or permanent access keys to her apartment to workers, housekeepers, etc., and distribute the access keys directly to the mobile device of the intended user.

Another area that could be interesting is communication between lock devices. This could be used for various purposes like time synchronization, exchange of mobile device information, global authorization for multiple lock devices by one "master lock device", etc. Communication between lock devices will require a new analysis of the discovery strategy in the previous chapter and also possible a second Bluetooth radio on the lock device.

We see these issues as interesting topics for further research, that could extend the use of the system to new scenarios.

3.8 Conclusion

In this chapter we have analyzed and designed a system for access control using non-secret access keys. The authentication mechanism is based on asymmetric cryptography, namely ECDSA and X.509 certificates.

It fulfills the goals set, as the protocol is small and simple, allowing an easy implementation and understanding of it. The hardware requirements are also low, the minimal storage space for 100 keys is 3.0 Kbytes for the mobile device. Only space for the certificate of the external authority is needed on the lock device.

The cryptographic requirements are one signature generation by the mobile device, and three signature verifications by the lock device. With caching enabled, only one signature verification is needed on the lock device when unlocking is done with previously exchanged data. If shared secrets are enabled unlocking can be done using one HMAC-SHA1 generation by both devices. It is also shown how the authentication can be performed using Bluetooth security mechanisms, most likely with an increase in performance.

The access keys will be easy to distribute as they are non-secret, and with the proper parameters they can fit inside one SMS-message. They can also be customized to most purposes allowing them to unlock multiple doors, be used by multiple mobile devices, and have validity restrictions. Furthermore, the only information that needs to be guarded with care is the private key on the mobile device. Moreover, mobile devices need not to trust the lock device.

It is also possible for the mobile device to have different identities, bound to the same key pair or different key pairs. This makes it possible to use the same mobile device with many different external authorities. We have also proposed a scheme that with one modification to the protocol allows multiple authorities to create access key and certificates.

The protocol authenticates the mobile device and we have incorporated and proposed mechanisms to guard against theft and other attacks. As discussed the system is vulnerable to active MITM attacks, but even if we could secure the system against these, it would still be possible to mount a relay attack. We believe that the protocol is the best solution possible for the given requirements and context.

The system still needs to be evaluated in practice, where the general system speed and the different cryptographic algorithms should be tested. Moreover the caching effect on transfer and computation speed should be evaluated, and whether it is feasible to split of `MobileCred` packet in two. This will be done in Chapter 5.

CHAPTER 4

Implementing the Prototype

I've had a wonderful time, but this wasn't it.
– Groucho Marx

To test our design and answer the open design questions regarding time and energy usage we will implement a prototype. In this chapter we will briefly present the implementation.

For the mobile device we have chosen to use a P800 which is the top-of-the-line mobile phone from Sony Ericsson. The lock device will be implemented on a standard PC running Linux to make the implementation easy, as our main focus is on the mobile device.

We will first give a brief description of the P800, and then describe the challenges met during implementation.

4.1 The Mobile Phone

The P800 is the flagship of Sony Ericsson Mobile Communications[1] and is a mixture between a mobile phone and a PDA (see Figure 4.1 for a picture of it). It offers two modes of operation, a traditional mode with a numeric key pad and a full screen mode operated by a stylus using point-and-click and handwriting recognition[2]. Apart from this the only other controls are a 5-way jog-dial and two buttons (camera and Internet). The software includes: calendar, address book, email client, Internet browser, MP3 player, video player, Microsoft Word/Powerpoint/Excel viewer, and lots more[3]. The official hardware specifications are [22]:

[1]`http://www.sonyericsson.com/`.

[2]The keypad is in fact 100% mechanical. Pressing a button just presses a touch-sensitive virtual keypad on the underlying screen.

[3]We have been told that if you are really smart, it is even possible to do a phone call :).

Processor	ARM9
Physical size	117 x 59 x 27 mm, 158g
Screen	Touch-sensitive TFT screen, 208 x 320 pixels, 4096 colors
Battery	Li-polymer (3.6V, 1000 mAh), talk time up to 13 hours, standby time up to 400 hours
Communication	GSM, GPRS, HSCSD, Bluetooth, and IrDA
Operating System	Symbian OS V7.0
RAM	16 MB
Storage	12 MB flash available on board, and 16 MB on Memory Stick Duo
Programmability	C++ and Java (PersonalJava, J2ME CLDC 1.0 and MIDP 1.0)

The CPU speed is not mentioned, but using the SDK this is reported to be 156 MHz. Besides the above, the phone also has a serial port which is accessible through an USB-cradle. All in all a very potent platform!

Figure 4.1: Picture of the Sony Ericsson P800 phone.

4.1.1 The Symbian Operating System

The P800 uses Symbian OS as the operating system. Symbian OS is created by Symbian Ltd.[4] which is a company owned by Ericsson, Nokia, Panasonic, Motorola, Psion, Samsung Electronics, Siemens, and Sony Ericsson—in short, most of the major mobile phone companies. Symbian OS is a continuation the operating system on the PDAs manufactured by Psion PLC[5], called EPOC.

The first mobile phone to use Symbian OS was the Ericsson R380, and currently it is used in Nokia 92xx series (Symbian OS v6.0 / Series 80), Nokia 3650 and 7650 (Symbian OS v6.1 / Series 60), and of course in the Sony Ericsson P800 (Symbian OS v7.0 / UIQ). In *Nokia Developer Network Newsletter - EMEA* (May 7, 2003) the following is reported:

> Levin [David Levin, CEO of Symbian, red.] also highlighted the dramatic rise in Symbian OS phone shipments over the past three

[4]`http://www.symbian.com`
[5]`http://www.psion.com`

> years. He told the audience that in 2001 approximately half a million Symbian OS phones shipped. But in 2002, the number of shipments rose to more than 2.1 million, and in the first quarter of 2003 alone, nearly 1.2 million phones shipped. Looking to the future, Levin revealed that there are currently 21 products in development, of which the SX-1 from Siemens, the D700 from Samsung, the Nokia N-Gage(TM) mobile game deck, and the P30 from BenQ have been publicly announced, and an additional 19 projects are in the early stages of planning.

So Symbian OS is on a rise. The only alternative to Symbian OS is Microsoft Smartphone (based on Windows CE 3.0) which has yet to gain ground. For the moment Symbian OS looks like a platform that has a future.

Symbian OS can be programmed in C++ and Java. Implementing the system in Java will allow us to use the application on any Java equipped device. The problem is that the Bluetooth API for Java (JSR 82 [51]) has only recently been finalized (March 22 2002), and there is no support for it in Symbian OS 7.0. Support has been announced in the newest version of Symbian OS, version 7.0s [18], but whether an upgrade will be available for the P800 is unknown to us[6]. On top of the missing Bluetooth support, the Java support in general is also limited—the libraries available seems to be targeted at game development. Because of these issues and combined with the questionable performance of Java in general, we will implement the system in C++.

4.2 Implementation

In implementing the prototype system we have aimed at making the code as portable as possible. First of all because the exact hardware platform for the devices was not fixed at the start for the project. The delivery time of the mobile phone was uncertain, and it should be possible to move the lock device application from the PC to a more suitable platform. Secondly we did not know whether we would have proper debugging possibilities on the devices, so the more code that could be tested on a PC the better.

The system can be divided into four parts: The application logic, the protocol, Bluetooth support, and cryptographic support. The application logic and the protocol could be implemented generically, but the last two parts needed to be specialized for the two different platforms. We will describe the problems faced when implementing the last two parts using Linux and Symbian OS in the following.

[6]Nokia's newest Symbian OS phone, the Nokia 6600, has support for Java MIDP v2.0 with Bluetooth support. Release is announced to be 4th quarter 2003 (`http://www.forum.nokia.com/main/1,6566,015_291,00.html`).

4.2.1 Linux

Linux has two choices for Bluetooth support, the built-in BlueZ [60] stack or the Nokia-maintained Affix stack [39]. We have previously used BlueZ with success, but found that it lacked the proper support for the *Object Exchange Protocol* (OBEX) [47] that was needed to transfer files to the P800. Affix has full support for OBEX so for ease of use we also used it for the application with success.

For cryptographic support we experimented with two different libraries for Linux: *OpenSSL* [61] and *Mozilla NSS* [57]. We found that the utilities for creating certificates are far from mature in OpenSSL, and it took quite some time for us just to be able to create a valid certificate. Moreover, the library itself is rather complicated and was not intuitive for us. In the Mozilla NSS, the utilities for creating certificates were much more mature and the library was more intuitive for us to use. Moreover, the library automatically handled storage and retrieval of certificates, so we chose to use Mozilla NSS for the implementation.

The actual implementation was then rather straight-forward with no major difficulties. Both the lock device application and the mobile device application are implemented on Linux.

4.2.2 Symbian OS

Implementing on Symbian OS was a different kind of story. The first challenge was to make the SDK work with Linux, as this is our preferred development platform. The SDK is made for Windows, but as the compiler used is GCC it seemed feasible. Others had gotten the same idea and we located a utility to convert the SDK from Windows to Linux[7]. All in all, we did get the SDK to work, and have found no problems related to the fact that we are using it with Linux. We have provided an introduction on how to setup the SDK and develop for Symbian OS with Linux in Appendix B.

The C++ API [73] provided with Symbian OS is extensive, to say the least. There is support for threads, timers, GUI, database systems, certificates, TCP/IP (including IPSec), Bluetooth, multimedia, web-technologies (HTTP, WAP, etc.), and lots more. So the API looks very promising, but we found some challenges in programming for Symbian OS though. First, there is no support for the C++ *Standard Template Library* (STL) at all, not even a `string` class[8]. There are support for data structures like lists and queues, but they are Symbian's own implementations. Second, Symbian has exception support but it is proprietary using preprocessor macros, instead of using the standard C++ exceptions. This means that the programmer manually has to push and pop objects to and from a `CleanupStack` to secure cleanup during exceptions—in our opinion this is

[7]It took some time before we realized that the SDK was working properly, since we kept getting errors when we tried to compile quite a few of the SDK examples. We later found out that this is normal, they do not work with Windows either…

[8]There has been efforts to port STLport to Symbian OS (see `http://www.3glab.org/developer/symbian/`). Unfortunately we did not have time to test it.

an invitation for bugs and most definitely a job for the compiler. Third, although there is some C standard library support and a lot of POSIX-compliant functions, a function like `select` is missing and `fcntl` is in the header files but nowhere to be found in the libraries. These three issues make the learning curve steep for existing C++ programmers, and makes porting of existing applications and multi-platform development difficult—more difficult than it ought to be.

Other issues also make the programming task difficult. One has to do with the way Symbian OS is used by the mobile phone companies. The version of Symbian OS used in the specific mobile phone is not a pure version of the OS. That is, the mobile phone companies specializes the OS for their specific products, and add, leave, or change functionality in the specific version. As the documentation seems to mainly be maintained by Symbian, the discrepancy between the documentation and the actual OS can be significant. In our work we found that the documentation-advertised classes for `CertStore` and `CCertStore` needed for handling certificates, is not included in the SDK. Searching through the header-files we found an undocumented class called `UnifiedCertStore`, but our experiments with it has only resulted in errors. Another challenge is that Sony Ericsson, or Symbian, for some reason keep functionality hidden from third party developers. By disassembling the binary library files we found functions for signature verification and generation with RSA and DSA. These could be used in our application, but with nothing but the names of the functions to work with it would be very unsafe to use them. We never made the cryptographic libraries in Symbian work, so we had to implement the cryptographic support.

For debugging, Symbian OS advertises that it has on-target debugging support using either MetroTRK[9] or the GDB[10], but unfortunately this is one of the features that Sony Ericsson has left out in the P800[11]. Debugging on the phone was difficult, since the phone has limited screen space and offers no keyboard to control debugging. Moreover the OS cleared the program screen when an exception occurred, and from time to time even the OS crashed. All in all debugging directly on the phone was difficult, and we needed some kind of remote debugging. We could not use Bluetooth for debug communication as this was used by the application itself. By trial-and-error we found out that the serial port is not directly accessible, and ended up using the IrDA port and a Palm Pilot as a debugging console (it was the only other device we had with an IrDA port). Luckily we later found a somewhat hidden debugging library that allowed us to do very basic debugging, by printing debugging information to the serial port.

Since the cryptographic libraries in Symbian OS did not work for us, we had to implement the cryptographic support ourselves. We first looked at Mozilla NSS which we used on Linux, but found that it was to big a task to port it. We found another library called *cryptlib* [27], which has a more modular structure and is written in standard ANSI C. We did some attempts to port it, but without a proper debugger we never made it work. Unfortunately we did not

[9] `http://www.metrowerks.com/`
[10] `http://www.gnu.org/software/gdb/`
[11] Rumors say that it is under development.

find any better alternatives. All cryptographic libraries that we have found are very complex and functionalities are heavily interconnected, and it is thus very difficult to isolate and port just the needed functionality. To get some cryptographic support we ported the RSA reference implementation made by RSA Laboratories, the *RSAREF2* toolkit. This allows us to sign and verify with both public and private RSA keys, but no support for parsing certificate content and the highest key size available is unfortunately only 1024 bits.

Implementing the support needed for Bluetooth was not that troublesome when it came to listening, sending, or receiving[12]. We never got inquiry to work properly though. There are header files for doing direct HCI calls, but the libraries are missing. Therefore inquiry has to be done through a generic `HostResolver` class which has no options for inquiry length and maximum number of devices returned. Furthermore, we were never able to retrieve any valid Bluetooth addresses, although doing a *Bluetooth Discovery* through the GUI worked fine. We are either using the API incorrectly or the API is broken. We will further examine this in Chapter 5.

Another part that did not work either was the setup of the Bluetooth authentication. It is not possible to set a PIN code for connections via the SDK. In fact, Symbian OS presumes that this should be set via the GUI and insists on presenting a *Enter PIN code window* when authentication is requested on the mobile device side.

Finally we did get the mobile device implementation to work, but unfortunately with quite limited functionality. It is only capable of using RSA signatures, with no certificate parsing or support for shared secrets. We also implemented testing applications for the energy measurements.

4.3 Resource Cost

The mobile device application for Symbian OS is around 60 Kbytes, including the RSA support[13]. As this is a prototype and we have not optimized the code for running on Symbian OS, the resource cost is not fair in comparison with an optimized implementation. But compared to the resources available on the P800, the resource cost is insignificant.

The lock device implementation is around 80 Kbytes with support for both Mozilla NSS and the RSAREF2, but excluding the library cost of Mozilla NSS and the Affix Bluetooth stack.

4.4 Conclusion

Implementing the protocol itself was easy, which was one of the design goals and the overall implementation for Linux was also straight-forward. Imple-

[12]It did come as a surprise for us though, that a `recv`-command threw away any remaining data if there was not enough room in the buffer given. This behavior is not reflected in the documentation, which we also reported to Symbian whom promised to fix the documentation.

[13]Symbian OS did not let us examine the RAM used.

menting for Symbian OS was more difficult than we had imagined. Most of the issues mentioned in the previous sections was not documented or announced anywhere, and we had to find out by trial-and-error. We also found that the learning curve was rather steep, mostly because of the missing support for STL and the proprietary exception-handling system. This, and the fact that the documentation and the implementation differs quite a lot makes developing more cumbersome and tedious than it ought to be, and made the implementation take a lot of time—too much time. It seems like Symbian OS still has some rough edges that needs to be taken care of, before third-party developers can get the full advantage of the API.

All in all we did make a functional prototype with limited functionality. We would have liked to have other algorithms implemented, and also considered using GnuPG[14] or OpenPGP[15] for this instead of generic cryptographic libraries, but time did not allow us to investigate it further.

For a full implementation a cryptographic library would be needed, preferably the one included in Symbian OS. Besides better, and probably faster, cryptographic support it would allow the application to use security features in the OS to secure the private key. Moreover, the code should be optimized for the specific platform for both mobile and lock device to obtain the best performance.

[14] `http://www.gnupg.org/`
[15] `http://www.openpgp.org/`

CHAPTER 5

Evaluating the Design

Not everything that can be counted counts,
and not everything that counts can be counted.
– Albert Einstein

In this chapter we will evaluate the design using the prototype implementation. Our overall goal is to evaluate the impact of the application on the mobile phone energy consumption. The prototype should preferably have minimal impact on the energy consumption while still being able to unlock the door within our time constraints.

We start by describing the test setup used, and then analyze the P800 mobile phone to construct an energy cost model of the general device behavior. We will then analyze the energy consumption and performance of both Bluetooth and the cryptographic support. After that we will measure the time and energy used by the prototype itself.

5.1 Test Setup

Our first approach to measure the energy usage on the phone was to insert a multimeter directly in series between the battery and the mobile phone, but the phone refused to boot with that arrangement. The second approach was to insert a resistor instead and measure the voltage drop over it. This worked, but with no knowledge about the electrical characteristics of the phone, the best suited resistor had to be found by trial and error. The ideal would be a resistor with high resistance, because the higher the resistance, the bigger the voltage drops, and the less precision needed for the multimeter. However, we found that if the resistance was too high it would leave insufficient energy for the phone itself, and the phone would not function. We tested different sizes and found that 0.3 Ohm was the highest resistance possible. The multimeter used was a *Wavetek Meterman 235*[1] which supports readouts on a serial port approximately twice a second. Readings were recorded to a PC and converted to Watt using Ohm's law. A picture of the test setup can be seen in Figure 5.1.

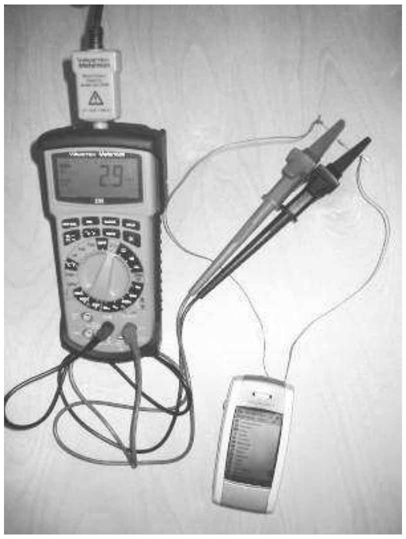


Figure 5.1: Picture of the measuring setup

The precision of the measurements depends on the precision of both the resistors and the multimeter, the stability of the voltage on the battery, and the resistance of the wires connected to the resistor. The resistors have an accuracy of ±5% and the multimeter one of ±0.25%. The stability of the voltage is important because the voltage is used to calculate the Watts using Ohm's law. Ideally the voltage should be measured continuously, but a second multimeter for this purpose was not available during our experiments. We did some initial measurements and found that the battery deviated less than 1% from 3.6V for the first hour at least. All our measurements were made on a full battery. If the measurements are relatively short, the voltage difference is negligible. We also measured the resistance in the wires connected to the resistor and found it to be immeasurable, so they will not influence the measurements either. All in all, the measurements will have an inaccuracy of maximum 6–7%.

Another factor that influences the measurements is the physical environment, as we are measuring wireless communications. Unfortunately we did not have a noise-free environment to test in, so all measurements were performed in an office environment with activity from other mobile phones, wireless LANs, etc. The downside is that the environment will make the measurements fluctuate, but on the positive side it does give more realistic measurements for a normal usage scenario compared to a noise-free environment. We have done all the measurements with the phone being placed in the same spot in order to obtain the same level of background noise.

All measurements were performed a minimum of five times and for at least ten minutes. To make the included graphs easier to read, measurements were

[1]`http://www.metermantesttools.com/products/TMeters.asp`

done for the shortest period possible while still showing the important factors. For short experiments we have performed the experiment in succession and displayed all the experiments on the same graph, with a random wait in between arbitrarily chosen to be between 12 and 17 seconds.

All experiments have been done with a CSR-based Microsoft USB Bluetooth unit as the Bluetooth unit for the PC. The unit was placed approximately 50 centimeters from the mobile phone.

5.2 The Mobile Device

In this section we will analyze the energy usage of the mobile phone in general. This will reveal the characteristics of the mobile device, which are not otherwise available. These informations can be used to affirm our assumptions and hopefully validate our design decisions in the previous chapters.

There are no public energy specifications for the mobile phone. We have tried to contact Sony Ericsson, but have not been able to get any information. The only available information is presented in Section 4.1, which is the voltage of the battery (3.6V), the capacity of the battery (1000 mAh), best case life time during conversations (13 hours) and standby (400 hours). From this we can deduce that the battery should have 12960 Joule, standby should consume ∼9 mW and conversations ∼277 mW. For the rest of the functionalities there is no information. In this section we will measure the energy usage of the different operating modes and functionalities and build a cost model.

5.2.1 Operating Modes

We will start by measuring the cost of the different operating modes of the phone. First how much it costs to have the phone in standby or *idle mode*, and after that how much some of the functionalities cost, that is, *busy mode*.

Idle Mode

Before we can make any energy measurements we must find out how the phone behaves without being used and with Bluetooth disabled. We assume that the following three consumes almost all of the power:

- The CPU
- The GSM radio
- The display

The radio can be turned of by setting the mobile device in a *flight mode*, that turns of the radio part of the device to allow it to be used in an airline or other areas where a mobile phone is disallowed. The display can have the back-light turned on or off, and the phone has power-save modes that saves power by turning the display off after a specific amount of time. The power usage of the CPU cannot be influenced by user-available settings. According to the ARM website[2], ARM9 CPUs come in two varieties where the only difference is the cache size. Depending on the type of silicon used (0.18 or 0.13 μm) the maximum power consumption (including caches) is reported to be either $156MHz \times 0.8 \frac{mW}{MHz} = 124mW$ or $156MHz \times 0.36 \frac{mW}{MHz} = 56.16mW$. Whether this holds for the CPU in the phone is questionable as it may be a custom edition of the chip, but it does give a hint about the power usage.

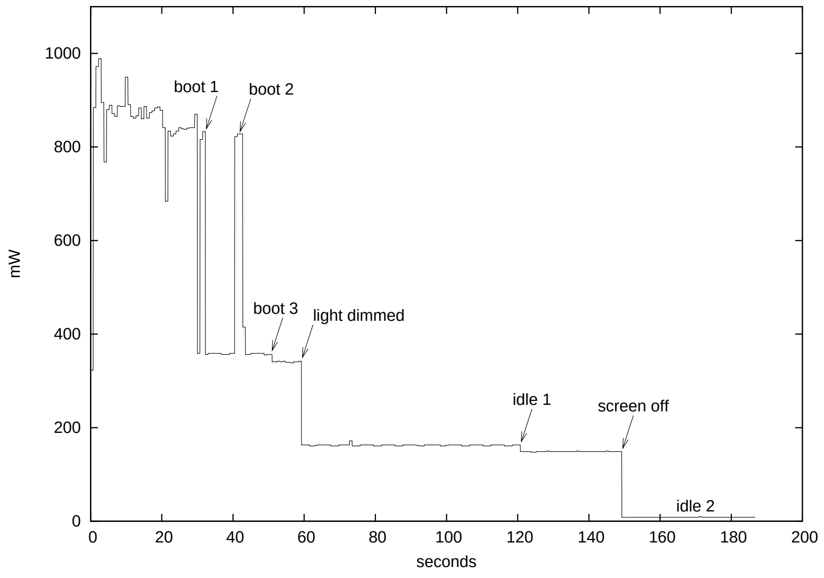


Figure 5.2: This graph shows six of the modes that the phone enters from booting until it reaches its final idle mode. Initially back-light is on, power-save is on and it is in flight mode.

To reveal the different power modes of the phone, measurements were made for all the possible combinations of the radio and back-light modes for three minutes from the phone is turned on (see Appendix C for the graphs). It takes 150 seconds before the phone reaches its final idle mode after being booted. The lowest power consumption is ∼9 mW, in flight mode with the screen turned off. The screen has four modes: On (∼320 mW), dimmed back-light (∼140 mW), no back light (∼20 mW), and off. The CPU seems to have two idle modes, idle 1 using 12 mW and idle 2 using 9 mW (entered after 120 seconds). We have pointed out some of the different modes in Figure 5.2.

We want to use the mode where the power consumption is as constant as possible for our measurements, to obtain a constant baseline. This seems to be flight mode with no back-light and no power-save. We measured the energy consumption of this mode over a couple of hours, to see the general behavior and to verify whether it kept constant. The results found in Figure 5.3 on the next page show that the power consumption seems to be rather constant at about 29 mW, which probably is the cost of the CPU running in idle mode 2

[2]`http://www.arm.com/armtech/ARM9_Thumb`.

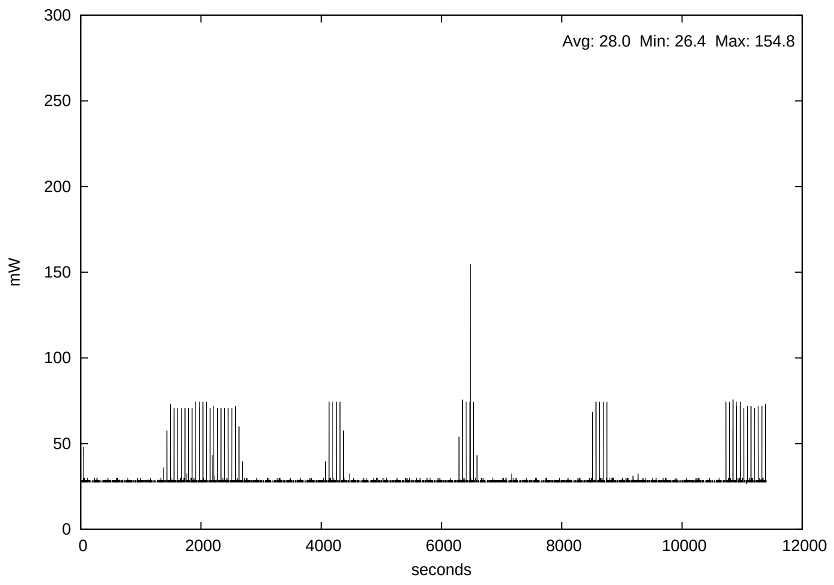
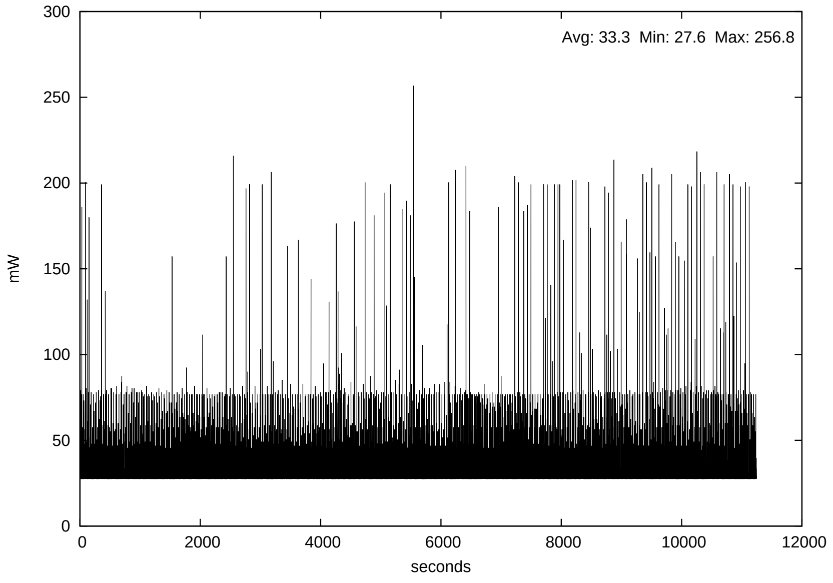


Figure 5.3: The energy consumption in flight mode. Back-light and power save are off.

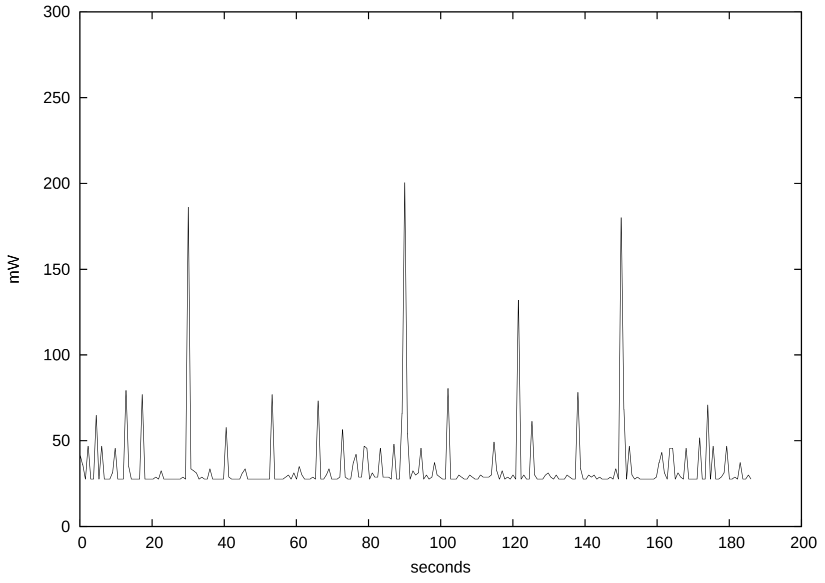
and the screen. Something also consumes power every 30–40 minutes, around 50 mW a couple of times over a couple of minutes. We stopped all possible programs, so whatever it is we cannot avoid it, and thus have to be aware of it in our measurements.

As we are measuring for such a long period, there is a chance that the voltage from the battery could change (see Section 5.1). Inspecting the graphs reveals that the baseline keeps at the same level which either means that the voltage is stable, or that the energy usage of the system is increasing at exactly the same speed as the voltage is dropping. We find the latter unlikely, and conclude that the voltage change is negligible.

For measuring anything that uses Bluetooth we need to use another mode though, because the flight mode also turns off Bluetooth. We have tried to find a way to have Bluetooth turned on without GSM also being on, but it is not supported on the phone. The phone refuses to boot without a SIM card so that is not a method to disable GSM. We have also tried to use invalid SIM-cards, but the phone just keeps on searching for a network. Even if it gives up on trying to find a valid network for the SIM-card, it will still keep a connection to one of the available networks since it always has to be possible to make an emergency call with the phone. The average energy consumption for the normal mode is ∼33 mW as seen in Figure 5.4(a) on the following page. The problem is that it is not a constant usage, it fluctuates, so it can be difficult to discern the GSM usage. Fortunately, the GSM power usage comes in peaks as (see Figure 5.4(b)), so as long as the functionality being tested has a stable power consumption it will be possible to disregard the GSM peaks.



(a) Long period.



(b) The first three minutes.

Figure 5.4: The energy consumption in normal mode. Back-light and power save are off.

Lastly there are the two idle modes of the CPU, which gives a difference around 4 mW. As far as our experiments show, idle mode 1 is only used during boot so it can be ignored.

Busy Modes

Having measured the idle modes of the phone, we will now measure the different busy modes. To be able to relate the power usage of our application to the power usage of the normal functions of the phone, we have tested the following: using the GUI, having a conversation, using the camera, and playing a game.

Having a phone conversation takes around 250–300 mW, not far from the specifications. Using the GUI or the camera costs 500–600 mW, while playing a game ranges from 800 mW to 1000 mW. Compared to these numbers it was a bit surprising to see the cost of just touching the screen. Holding the pointer still at the same position on the screen costs 500 mW, no matter whether the program has a GUI or not (a console mode program). This is probably caused by the endless processing of pointer events from the screen. For a GUI application this is unavoidable, but for a console program this is not necessary, and must be considered a design flaw[3]. Lastly, we tried to stress the phone maximally by playing a game with back-light on, full sound, a GSM connection, and Bluetooth connection on at the same time—this peaked around 1570 mW. The graphs are shown in Appendix C.

To find the cost of having the CPU running at full speed, we created a simple program with a busy loop increasing a simple counter. The cost of this is $\sim$508 mW as seen in Figure 5.5 on the next page. There are probably other hardware parts than the CPU that takes power in this mode, especially if the processor has the specifications found in Section 5.2.1, as it should then use a maximum of 124 mW. We will not go in to further speculations about what it is, but just conclude that the busy mode costs around 508 mW.

Summary

To summarize the above measurements, the cost of the phone functionalities are:

Function	**Average cost (mW)**
Idle mode, flight	29
Idle mode, normal	33
CPU in busy loop	508
Having a conversation	250–300
Using GUI / camera	500–600
Maximum measured	1570

[3]We also tried to enter the *lock mode* of the phone which disables all input except the unlocking keys, but the same phenomenon was observed here. As the screen is needed to unlock the phone some kind of polling is necessary, but we believe that less frequently would do.

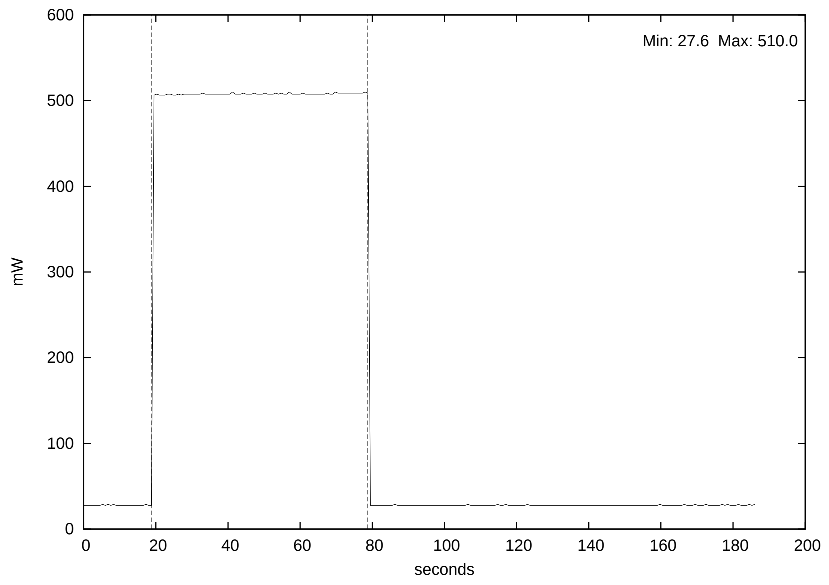


Figure 5.5: Energy usage when the phone executes a program with a busy loop for 60 seconds.

Thus, to find the power consumption of a program we can subtract 29 mW when the phone is in flight mode, and 33 mW when it is in normal mode.

5.2.2 Bluetooth

After having established some knowledge about the phone's general characteristics we can now begin to measure the energy consumption for Bluetooth. Bluetooth functionality can be divided into:

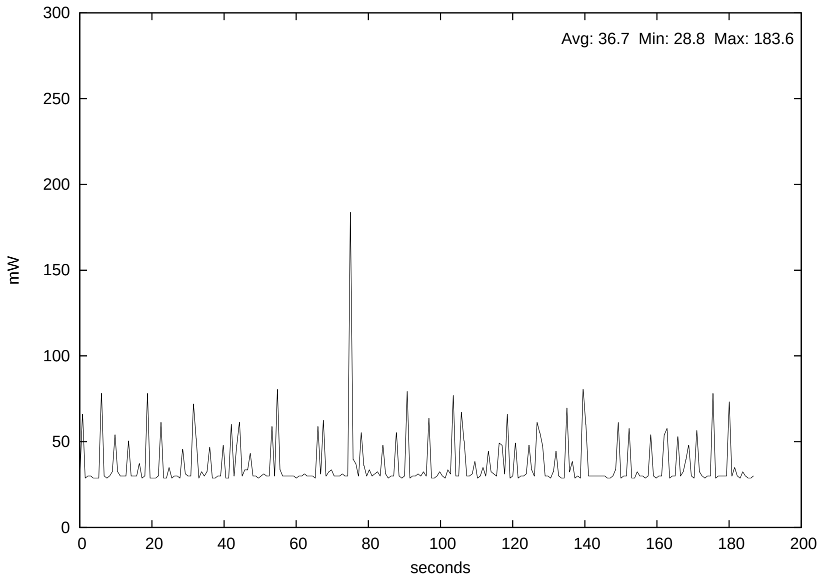
Idle: Bluetooth off, on (no inquiry or page scans), connectable (page scan), and discoverable and connectable (page and inquiry scan) [Figure 5.6]

Discovering: Answering an inquiry, and doing inquiry [Figure 5.7]

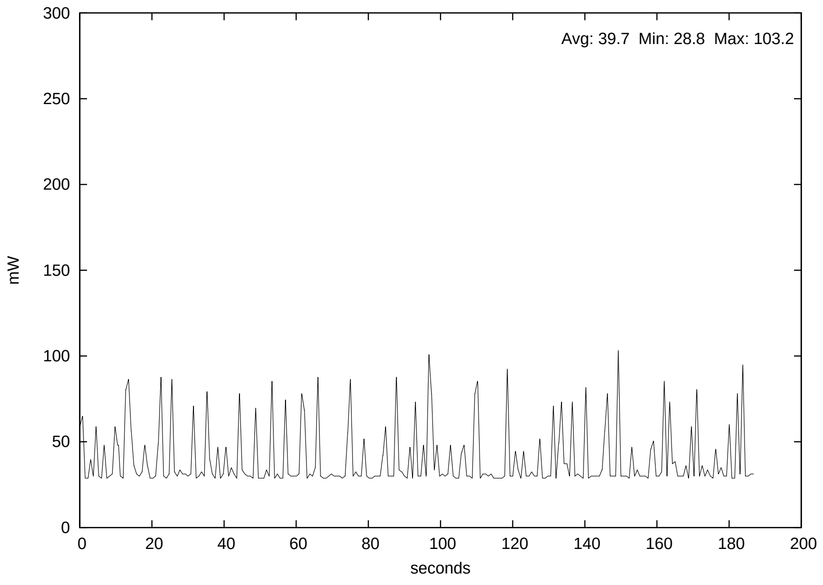
Connecting: Failed connect, and being connected [Figures 5.8 and 5.9]

Transmitting: Receiving data, and sending data [Figure 5.10]

We start by measuring the cost of having the Bluetooth radio turned on in its various modes. As we can see in Figure 5.6 on the facing page, the difference between the normal mode with Bluetooth off and being connectable or discoverable is noticeable, but it is difficult to say exactly how much with the GSM traffic going on. Looking at the average energy consumption, the cost of having Bluetooth turned on is around 4 mW and 7 mW accordingly. Having the Bluetooth radio turned on without being connectable was not directly accessible, and with the small cost of being connectable we did not investigate it further. All in all, having the Bluetooth radio turned on is cheap.



(a) Connectable (page scan).



(b) Discoverable and connectable (page and inquiry scan).

Figure 5.6: The energy usage of the phone with Bluetooth in two different *idle* states.

The cost of answering an inquiry is seen in Figure 5.7(a) on the next page, where the dotted vertical line pairs show the start of the inquiry procedure on the remote host, and when the answer is received. The only conclusion we can give is, that whatever amount of energy being used it is obscured by the GSM activity—the power used is negligible. Performing inquiry on the phone consumes a lot more energy, as seen on Figure 5.7(b) on the facing page. As reported in Section 4.2.2, we had troubles using the inquiry API. From the graphs it seems like the inquiry is being done, although the application layer malfunctions. Software behavior moreover indicates a caching mechanism, but energy usage does not. The dotted vertical line pairs on the figure denote the start and the end of the inquiry function call, but except for the first call, all calls return immediately and only every other call uses energy (seen on Figure 5.7(b) where the dotted line pairs, all except the first, looks like one line). Our interpretation from the energy usage is that inquiry is being done, and the inquiry lasts for ∼13 seconds. This coincidently is close to $10 \times 1.28s = 12.8s$[4], and uses approximately $160mW - 33mW = 127mW$. Note also the *slack-phase* after the inquiry, where the energy usage keeps at ∼57 mW for 17 seconds before returning to the normal idle state. Without counting in the slack, the cost for doing an inquiry is $12.8s \times 127mW = 1625.6mJ$, while answering an inquiry is unnoticeable.

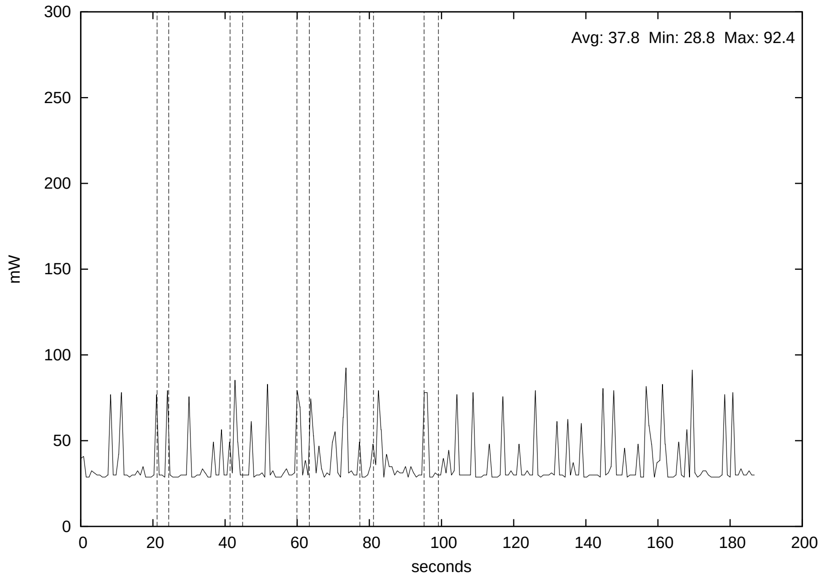
When it comes to a Bluetooth connection, the energy consumption rises considerably. As seen in Figure 5.8 on page 70, an incoming connection takes around $590mW - 33mW = 557mW$. This rather expensive cost probably also covers the cost of the busy mode at 508 mW, so the cost of Bluetooth itself is $590mW - 508mW = 82mW$. After 180 seconds the phone changes its power consumption and the connection takes 100 mW. Subtracting the idle cost gives 77 mW which is not far from the 82 mW, so around 80 mW is probably the true cost of maintaining an incoming Bluetooth connection. Handling a failed connection takes 3.5 seconds, costing $3.5s \times (580 - 33)mW = 1914.5mJ$.

The fact that it takes energy to make a connection to a *closed port* (that is, a port that no program is listening on), may seem surprising at first, but the phone has to receive and process the incoming request before it can determine what port it is for. A strange thing is that whenever an incoming Bluetooth connection is detected, no matter if it fails or is successful, the phone awakens itself from any power-save mode—even though there is no action on the screen. That means that any Bluetooth connection also has the cost of bringing the phone away from idle mode (including eventual back-light, etc.). We can only conclude that this is a design flaw.

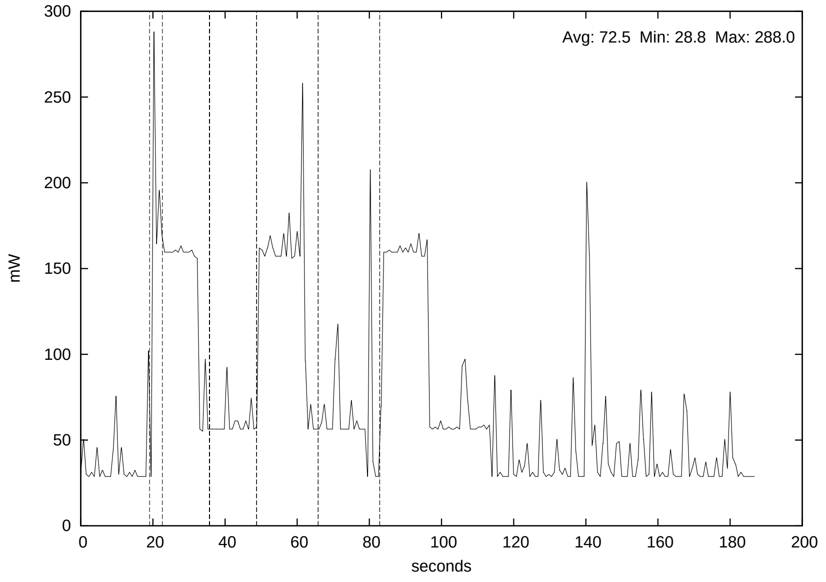
If the phone is the initiating part (see Figure 5.9 on page 71), doing a connection costs around $565mW - 33mW = 532mW$, and the phone also powers down after 180 seconds to 104 mW. Hence, for the first 180 seconds, initiating a connection is cheaper than handling an incoming connection. Failed connection attempts cost around 160 mW for 10 seconds giving a total of $10s \times (160 - 33)mW = 1270mJ$, a lot cheaper than receiving failed connections.

We also experienced the same slack phenomenon we experienced during inquiry, varying from 20 seconds in Figure 5.8(b) to 28 seconds in Figure 5.8(a).

[4]Inquiry in Bluetooth is performed in 1.28 second intervals.

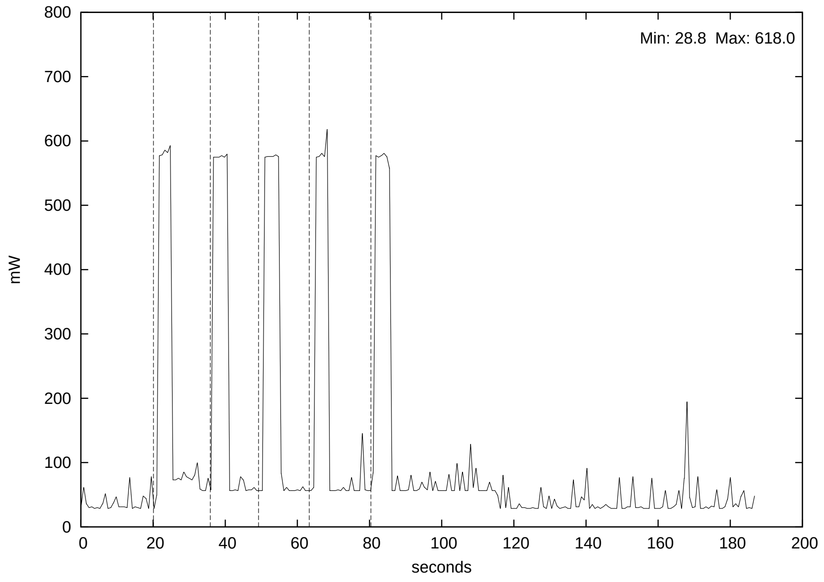


(a) Answering inquiries.

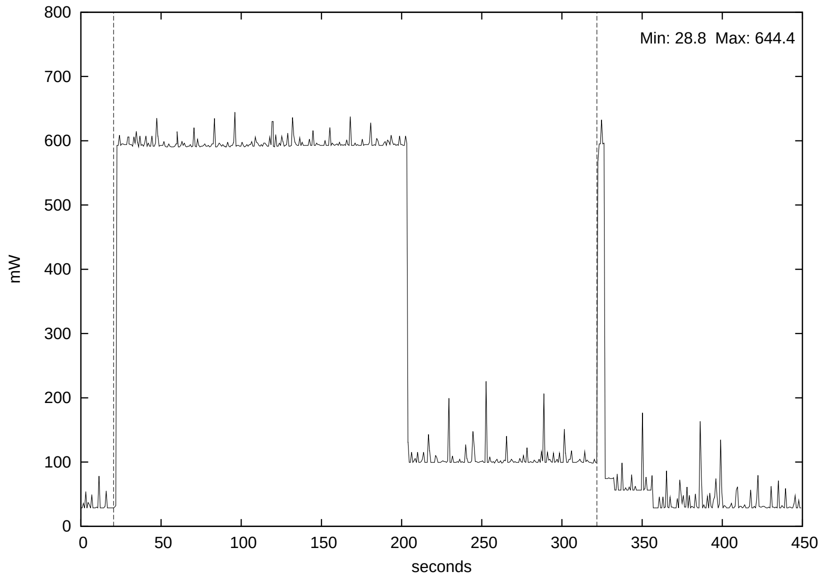


(b) Doing inquiries.

Figure 5.7: The energy usage of the phone when answering and doing Bluetooth inquiries. The dotted lines shows in pairs the start and end of the inquiry commands. In Figure 5.7(b) the pairs are only visible in the first call, the rest looks like one line.

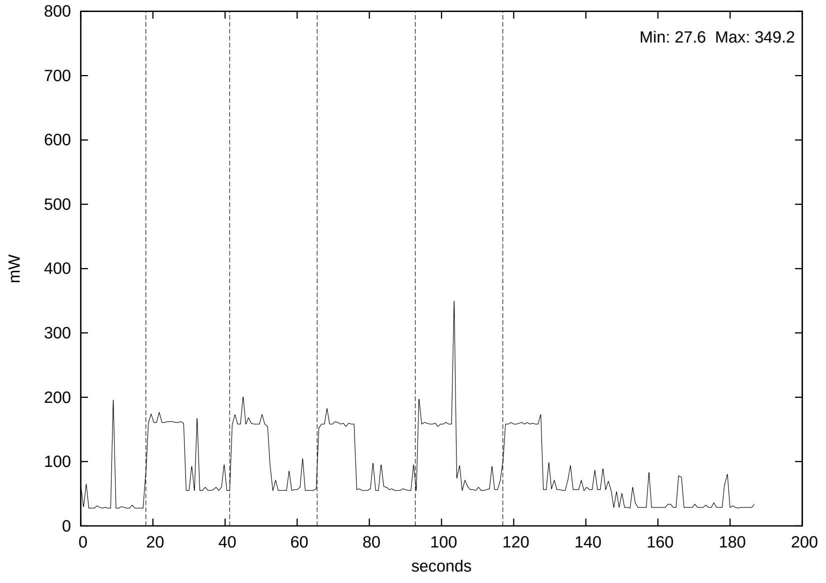


(a) Receiving connections to closed port.

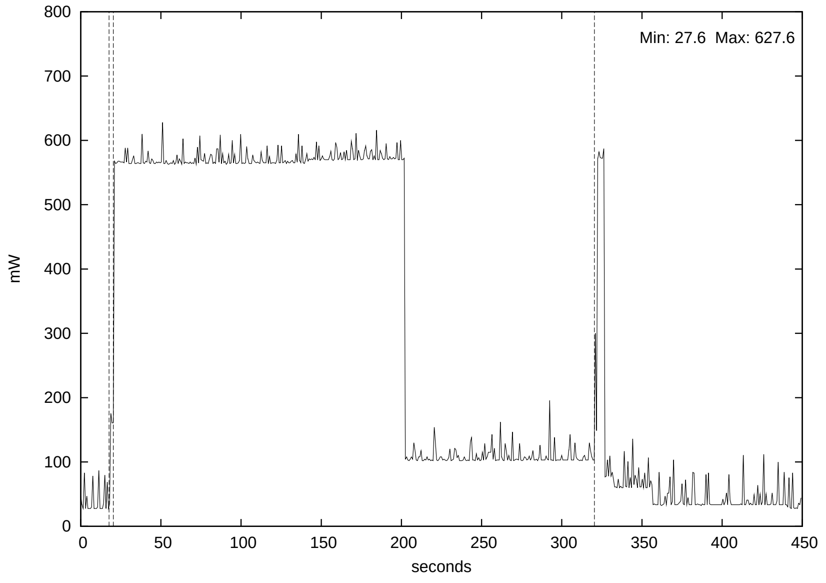


(b) Receiving connection, open for 300 seconds.

Figure 5.8: The energy usage of the phone when receiving Bluetooth connections to a closed and an open port.



(a) Initiating connections to closed port.



(b) Initiating connection, open for 300 seconds.

Figure 5.9: The energy usage of the phone when initiating Bluetooth connections to a closed and an open port.

The same 3.5 seconds experienced after a failed connection is also experienced after successful connections.

We then send and receive data over the connection, as seen in Figure 5.10 on the next page. Except for the power-down stage after 180 seconds, the graphs are almost exactly the same. Even though data needs to be transmitted and processed, it does not cost more compared to having the connection open. After 180 seconds there is a difference: receiving data on the mobile takes on average $324mW - 33mW = 291mW$ and sending takes $364mW - 33mW = 331mW$. Whether this is the cost of transferring data or the program itself is hard to say. If the communication takes more than 180 seconds in our scenario something is wrong, so we choose not to investigate this further.

The slack is also present again and takes around 30 seconds, so except for answering an inquiry every Bluetooth action on the phone has this slack for some time ranging from 17 seconds to 30 seconds always taking around 57 mW, or 24 mW more than being idle. Put differently every Bluetooth action on the phone takes an extra $17s \times 24mW = 408mJ$ to $30s \times 24mW = 720mJ$.

To summarize, when idle mode is subtracted Bluetooth costs:

Action	**Cost**
Being discoverable	4 mW
Being discoverable and connectable	7 mW
Answering an inquiry	< 1 mW
Doing inquiry	127 mW
Maintaining a connection or transferring data	557 mW / 532 mW[5]
Transferring data, > 180 seconds	291 mW / 331 mW[5]
Slack	408–720 mJ
Extra cost on connections (3.5s)	1914.5 mJ

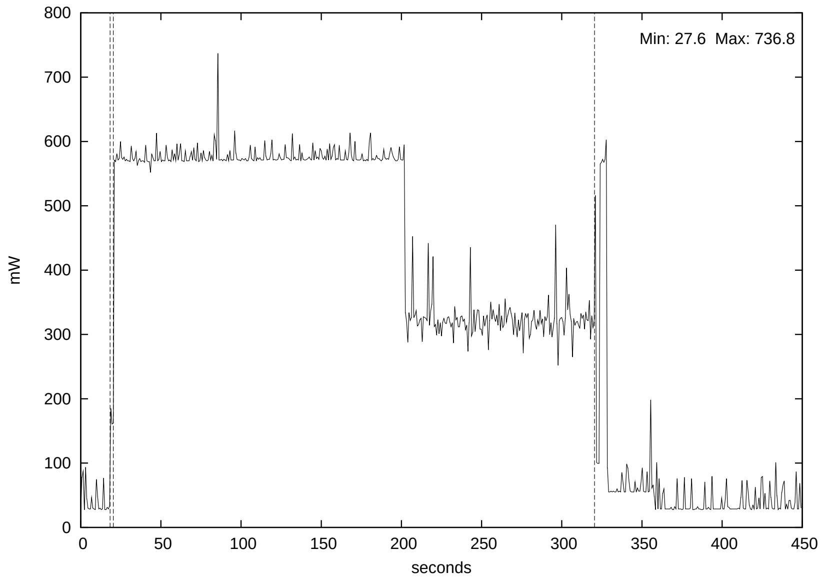
Compared to having a conversation, using Bluetooth is quite expensive. Especially when adding the slack and extra cost.

5.2.3 Cryptography

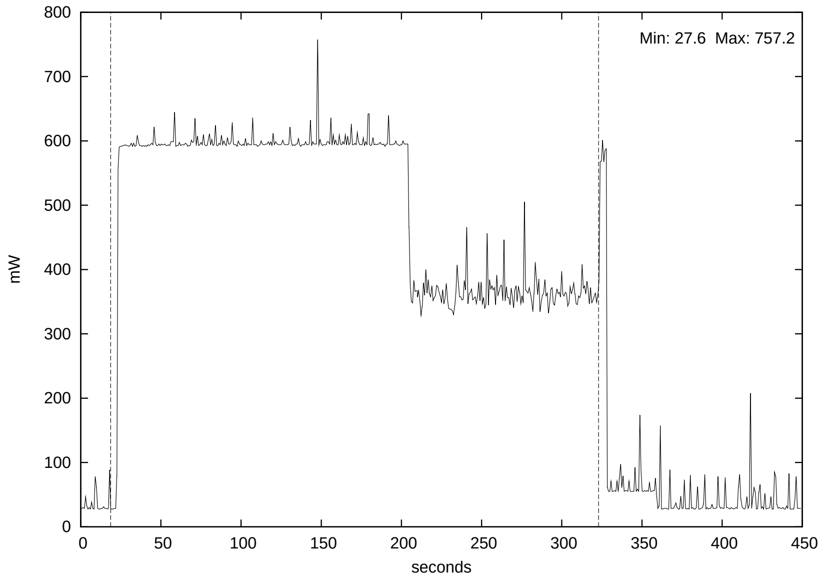
In this section we will test the RSA Reference Implementation, the cryptographic library we used. We will test signing and verification with both 512 and 1024 bit keys, higher keys are unfortunately not available (see Section 4.2.2).

We ran 200 iterations for signature measurements, but had to run 20,000 iterations of verifications to 1024 bit keys and 200,000 iterations for 512 bit keys to be able to measure the cost. All operations were done on 2048 bytes of text, which was chosen to be big enough to cover our needs. The exact size is not important, as the expensive function is the RSA step and not the hashing step. The tests were run on the same key pairs. A more thorough test on an assortment of key pairs would give more qualitative results, but we believe that these tests are sufficient to give general measures for both energy cost and tim-

[5] When phone initiates or sends.

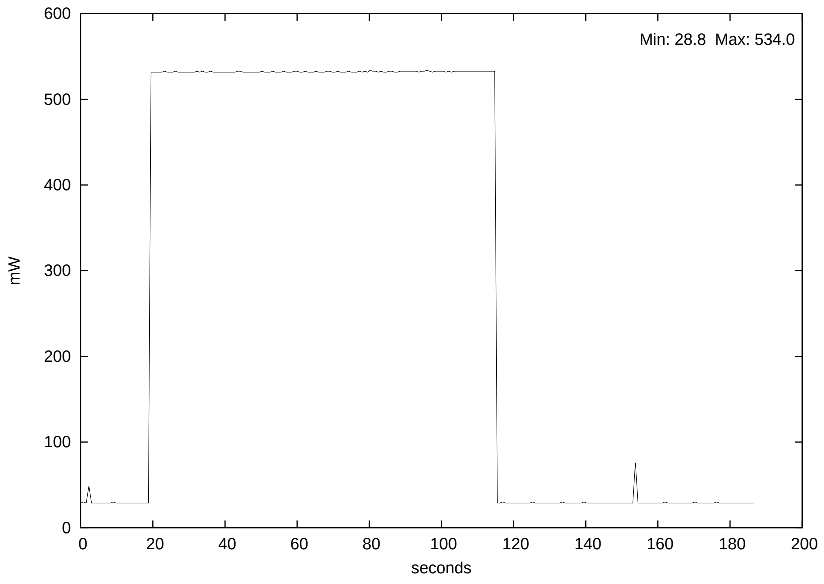


(a) Receiving data for 300 seconds.

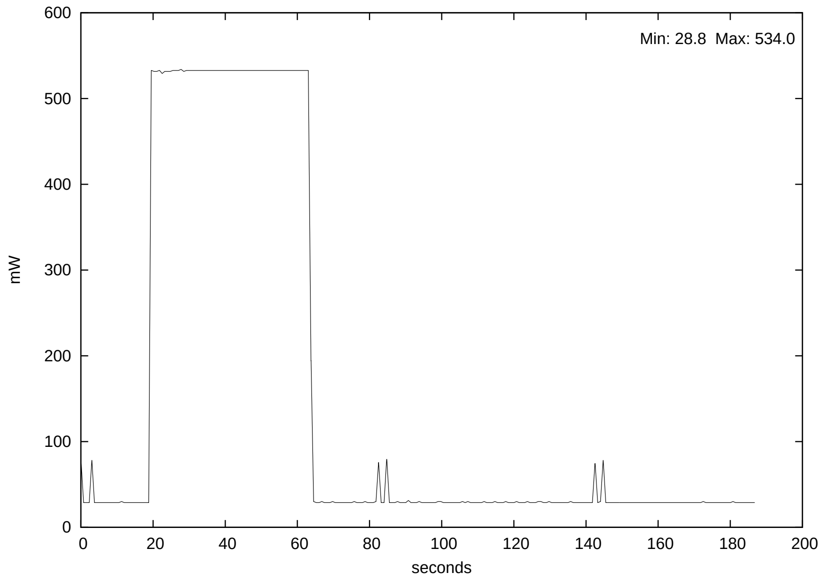


(b) Sending data for 300 seconds.

Figure 5.10: The energy usage of the phone when sending and receiving data with Bluetooth.



(a) Signing 1024 bits, 200 iterations.



(b) Verifying 1024 bits, 20,000 iterations.

Figure 5.11: These graphs shows the energy usage during RSA signature generation and signature verification with 1024 bit keys. Please note the substantial difference in the number of iterations done.

ing since the difference between different key pairs are insignificant compared to the total cost of the operation.

The results show that that the energy cost for all operations is approximately 532 mW, the only difference being the speed. The energy usage is in fact 24 mW higher than our previously measured busy mode cost. The reason must be the simplicity of our busy mode program, which was just a simple loop, compared to the RSA algorithm which somehow takes more power from the CPU. The results are shown in Figures 5.11 on the facing page and 5.12 on the next page, and we have calculated the cost of one iteration of each operation here:

Operation	Time (s)	Cost (mJ)[6]
Sign, 1024	0.48	241.00
Verify, 1024	0.02	10.08
Sign, 512	0.08	40.32
Verify, 512	< 0.01	0.01

The only expensive operation is signing with a 1024 bit key, the rest are almost unnoticeable. There is a factor six speed difference between signatures with 512 and 1024 bit keys. The same factor or slightly larger is also seen in [15] both between 512/1024 and 1024/2048, so this should also be valid for our implementation. So even though we are not able to validate this in practice, a signature with a 2048 bit key will probably take at least 3 seconds on the mobile phone. This makes 1024 bit keys the largest usable, given our time constraints.

The phone handles 1024 bit keys well; half a second should be fast enough for our purpose. If it turns out not to be, performance might be improved through more optimized code. It is also possible to optimize RSA with techniques described in [13] and gain a factor two speed up.

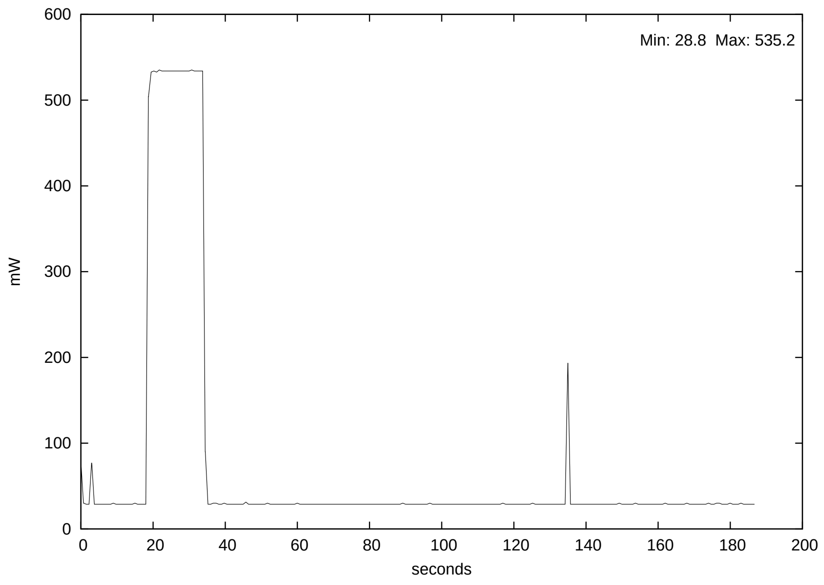
5.2.4 Conclusion

One of our first assumptions was that inquiry was expensive compared to inquiry scan, and it certainly is. Doing inquiry scans cost 7 mW while doing inquiry costs 127 mW. The cost of answering an inquiry was so low that it disappeared in the GSM background noise. Doing inquiries constantly would drain the battery 14 times faster than doing inquiry scans, so the choice of the lock device as the inquiring party seems correct.

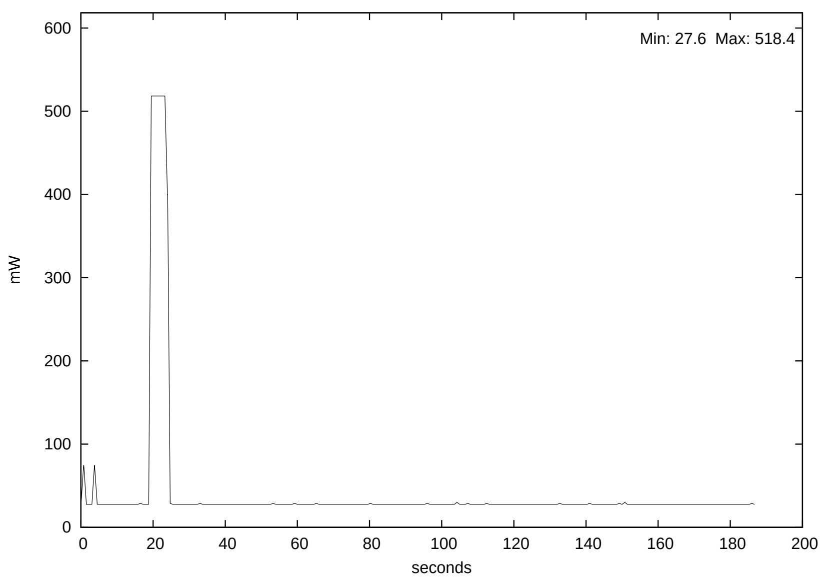
When it comes to the actual communication cost Bluetooth is much more expensive than having a GSM conversation, but after 180 seconds the energy level almost halves. We do not know what makes the energy consumption fall that dramatically, but it seems natural to assume that it could be kept at that level for all communication with a better design. We also experience both some slack after all Bluetooth commands, and a 3.5 seconds slack after all Bluetooth connections. Again, both of these could probably be optimized away.

Regarding cryptography, the device performs well. Generating a level 1 signature takes 0.5 seconds using the non-optimized code—only 7.5 times slower

[6]With the base cost of 29 mW subtracted.



(a) Signing 512 bits, 200 iterations.



(b) Verifying 512 bits, 200,000 iterations.

Figure 5.12: These graphs shows the energy usage during RSA signature generation and signature verification with 512 bit keys. Please note the substantial difference in the number of iterations done.

than the PC used in [15]. Extrapolating these results, an ECDSA signature would take 0.015 seconds for level 1 and 0.030 for level 2. This would practically eliminate cryptography as a time factor for the application.

5.3 The Application

We will now analyze and measure the actual cost of our application prototype. Based on the measurements in the previous sections we can deduce what discovery and connection from the lock device should cost. We do not know how much time the actual data transfer takes, but with the conservative assumption that it takes less than 0.5 seconds, the cost calculation is:

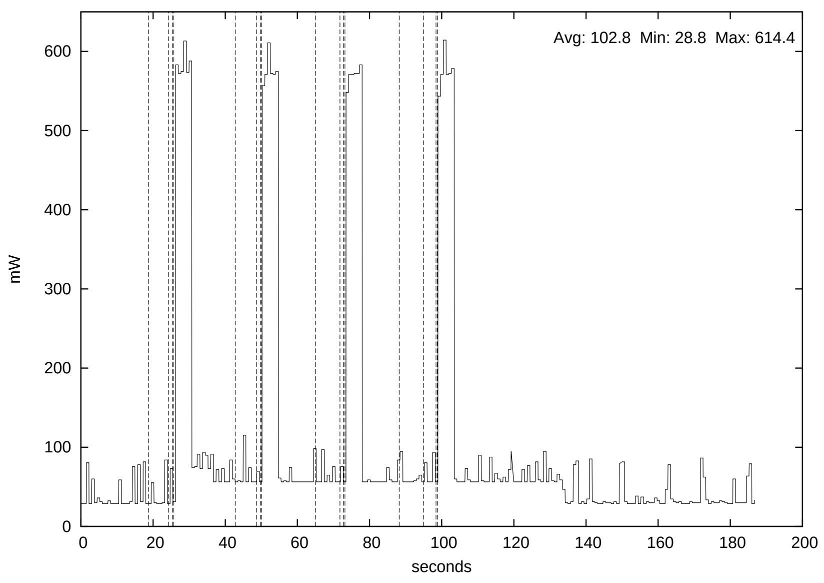
Function	**Cost (mJ)**
Answering inquiry	0
Signature (1024 bit)	241
Bluetooth connection (0.5 s)	278.5
3.5 s connection slack	1914.5
Bluetooth slack	408–720
Total	2842–3154

Around 3000 mJ for the whole application run, the most costly part being the 3.5 seconds slack after each Bluetooth connection. The idle mode cost and the cost of Bluetooth (40 mW all in all) should be added for the entire period to obtain the full cost.

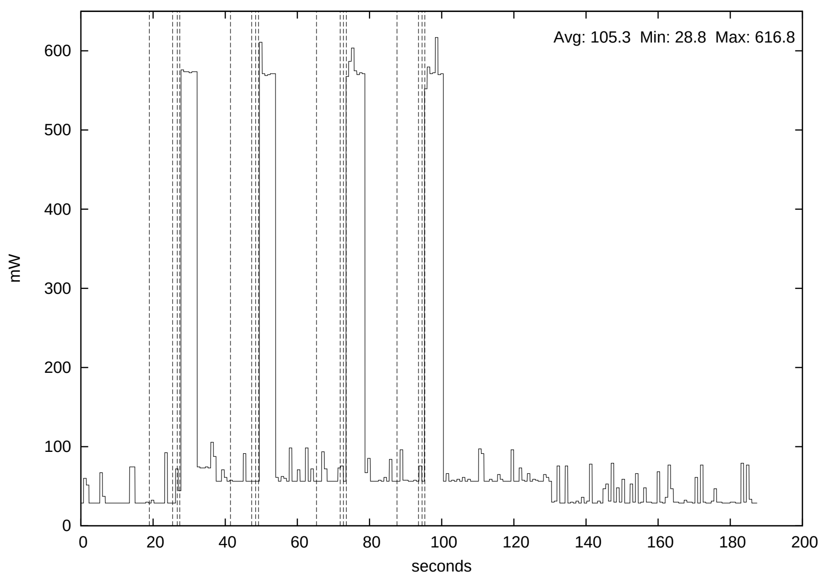
We measured the actual energy cost of the application for both 512 and 1024 bit keys, and the results are shown in Figure 5.13. The dominating parts are the 3.5 seconds extra connection time and the extra slack, whereas the connection is ended before any energy consumption is measured. With the multimeter we have access to, we are not able to measure whether the connection and signature generation for the full application matches our calculations. However, since the rest of the calculations matches the measurements we are confident that these are also valid.

We also timed the application 1000 times with full `MobileCred` packets. These tests revealed that average inquiry time was 3.6 seconds, which is more than one second slower than previously reported. The procedure was stable though, the minimum inquiry time was 2.8 seconds, and the maximum was 4.4 seconds. The reason for inquiry being slower may be found in the interoperability between the specific Bluetooth devices we used, or the specific noise conditions in our office environment. It is beyond our scope to investigate this further, but the results confirm that the Bluetooth discovery process is influenced by the specific equipment and environment.

Average connection time was 1.2 seconds which was a lot faster than we expected, but it was not as stable as inquiry. Even though most connects took around 1 second there were 23 connections that took over 10 seconds, the maximum being 18 seconds. It is unclear for us how this can happen. First, there has to be a design flaw somewhere, when connection establishment can take



(a) Application with 512 bit key.



(b) Application with 1024 bit key.

Figure 5.13: These two graphs show the lock device connection to the mobile device, with mobile device key being 512 and 1024 bits. The vertical lines denote: Inquiry start, connection start, connection succeeded, and connection closed.

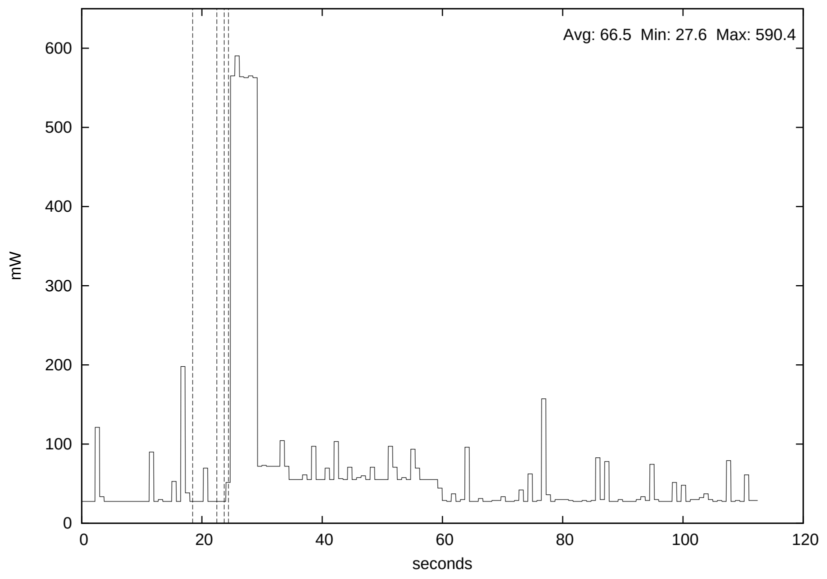


Figure 5.14: This graph show a single connection from the lock device to the mobile device, with the mobile device key being 1024 bits. The vertical lines denote: Inquiry start, connection start, connection succeeded, and connection closed.

18 seconds. Second, the standard *page timeout* for a Bluetooth device is 5.12 seconds and the Bluetooth device we used is also set to this (see [10, Part H, 4.7.16]). So the connection establishment should timeout after this period. The same phenomenon was experienced in [7], but unfortunately we do not have an explanation for this. Our conclusion is that there must be something wrong with the connection establishment procedure.

Apart from discovery and connection establishment the rest of the application takes 0.7 seconds on average, where 0.48 seconds are from the signature generation, so the rest of the application including communication takes around 0.2 seconds. All in all, the total average unlocking time is 5.5 seconds, which is shown in Figure 5.15—Bluetooth discovery and connection establishment accounts for 87%. This is still almost a second faster than the expected 6.33 seconds (see Section 2.3). The reasons are the faster connection time and the application taking 0.7 seconds instead of the assumed 1 second.

We also tested the application with `MobileCred` packets without the certificate and the access key, but the differences in application timing compared to the full packets were minuscule. In order to save transfer time, we conclude that the caching of the certificate and access key is an unnecessary complication. In general we must conclude that protocol optimizations of that kind have limited effect on the transfer time, when the full communication and application cost is 0.2 seconds.

Whether caching the certificate or splitting `MobileCred` into two packets as suggested in Section 3.3.3 improves overall application time significantly

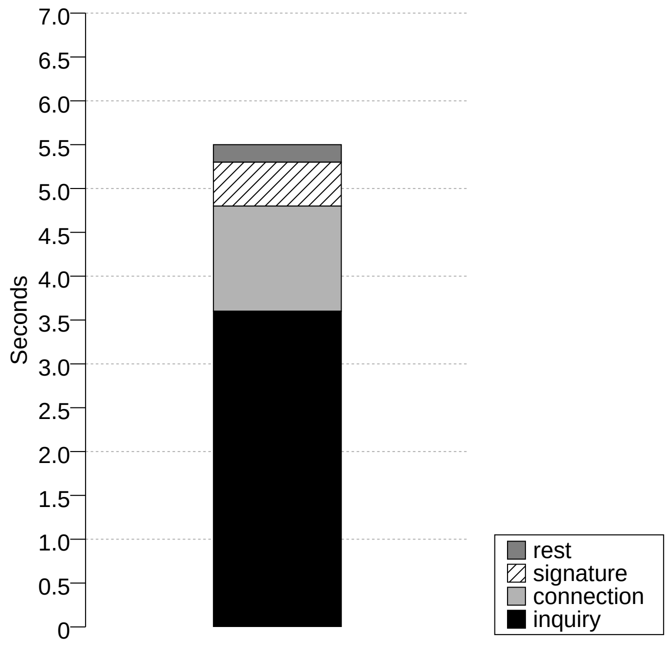


Figure 5.15: The average time usage of the different parts of the application.

will depend on the lock device hardware. As we have focused on the mobile device until now, we have not looked into that aspect. We will touch the subject of lock device hardware in Chapter 6.

5.3.1 The Infamous 3.5 Seconds

Late in the process it occurred to us what might cause the 3.5 extra seconds on each Bluetooth connection to the phone: The connection is not closed even though the Bluetooth stack reports it.

The explanation is as follows: to close a L2CAP connection, on either Symbian OS or Linux, we use a high-level *close* function that informs the communicating party that the connection is being closed and closes the connection. This works fine and the L2CAP-connection is closed immediately on both sides. The problem is that one of the Bluetooth stacks keeps the underlying ACL-connection alive for a period after the L2CAP connection is closed. This timeout causes the extra 3.5 seconds. We do not know for sure which of the protocol stacks cause this, but bringing down the Bluetooth interface completely in Affix directly after closing the connection, makes the power consumption fall immediately. Hence our guess is that Affix is the culprit, which is confirmed by at least one other person on the *Affix developers mailing-list*[7].

We did not have time to redo all experiments, but our measurements show-

[7]`http://lists.sourceforge.net/lists/listinfo/affix-devel`

ed that bringing down the Affix interface totally eliminated the 3.5 seconds after each connection. The 17–30 seconds general Bluetooth slack remained though. We conclude that the problem is in the Affix stack, and can ignore the 3.5 seconds in our cost measurements.

The rationale for the ACL-connection timeout in the Affix stack may be an attempt to optimize multiple connection attempts. The establishment of an L2CAP connection when the ACL-connection is already in place is done with no noticeable delay, the devices are already connected and synchronized. So any program that needs to create two subsequent connections to the same device saves 1.3 seconds on average for the second connection establishment. This is frequently needed as the first connection is used to query the SDP-database, and the second is to the actual service.

To our knowledge there is no indication of such an optimization in the standard, and we do not know whether the same phenomenon is found in for example the BlueZ stack.

5.4 Conclusion

In this chapter we set up a test environment that let us construct a cost model for the functionalities of the P800 mobile phone, and test the prototype we developed.

The tests confirmed our assumptions about the time and energy performance of Bluetooth and showed that our prototype implementation performed better than expected and allows a door to be unlocked in 5.5 seconds. Assuming that the application has full control of the underlying Bluetooth stack, the 3.5 seconds extra connection time can be avoided and the program cost is 241 mJ for the signature and $0.2s \times 557mW = 111.4mJ$ for the connection, totaling 352.4 mJ. On top of that comes the Bluetooth slack, so all in all the maximal cost is only 1072.4 mJ. This is the same cost as holding the pointer still at the screen for 2 seconds or having a phone conversation for 4 seconds. Or put differently, with a battery of 12960 J and the phone in normal mode with Bluetooth on (using 40 mW on average) it can do more than 10000 consecutive unlockings.

These figures allows the application to run on the mobile phone with little influence on the lifetime of the battery, and allows the range of the lock device to be two meters or less as we needed. All in all the application fulfills the goals set.

There are still issues that need to be tested. Even though the application performs well using RSA, we still believe that ECDSA is the optimal solution for the scenario and it should be tested on the mobile phone hardware. Moreover it would be interesting to test the shared secret optimization and compare it to the figures found during these tests. Testing the application in a real-life scenario with multiple users and doors is also a topic for further research.

CHAPTER 6

The Optimal Device

Everything that can be invented has been invented.
– Charles H. Duell, Commissioner, U.S. Office of Patents, 1899

In the previous chapters we have designed a system that fulfills its goals in the given context and with the available hardware. However, to make the system optimal there are two hardware-based issues that could be improved:

1. Discovery and connection establishment speed

2. Context awareness

The first issue concerns Bluetooth and the second the generic mobile phone hardware. Based on our experiences in previous chapters and looking beyond these two factors we envision the optimal mobile device to be:

> A small and light device that is aware of when it is in front of a door and exactly which door it is. It is also aware of whether the user carrying it wants to open a door or not. Discovery and connection time is less than a second and its energy resources are virtually unlimited. It also has fast cryptography support, is tamper-resistant, and has room for hundreds of keys.

With this device we believe the system would be optimal, although it is beyond the capabilities of a standard mobile phone equipped with Bluetooth. As we see it, it could either be done with a custom device or by augmenting the mobile phone. These approaches are discussed in the following.

6.1 Augmenting the Mobile Phone

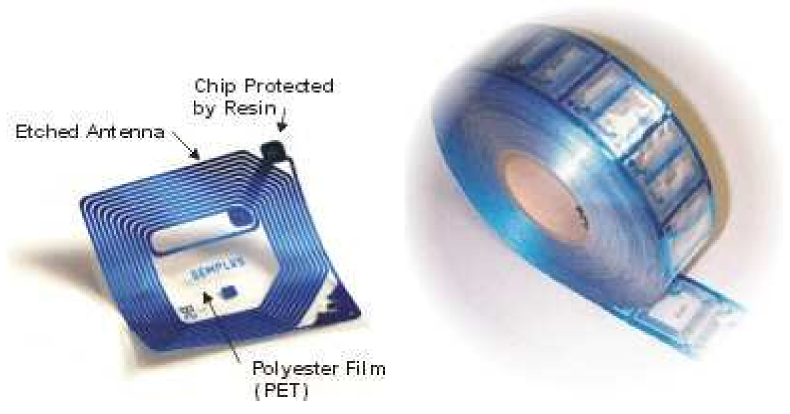
The first problem for the system is the discovery and connection establishment of Bluetooth. Even with careful analysis of how to perform this as fast as possible, it still consumes 87% of the total application time of 5.5 seconds (see Figure 5.15 on page 80). Hence Bluetooth may not be the correct technology for the optimal device.

But instead of altogether replacing Bluetooth it may be beneficial to focus on the speed of the discovery mechanism. Bluetooth inquiry procedure takes on average 3.6 seconds, while connection establishment on average takes 1.3 seconds. We propose to exchange the discovery procedure with a faster solution but still use Bluetooth for the communication. The purpose of the discovery mechanism is to detect the presence of another device and obtain its Bluetooth address[1], and in our scenario this can be achieved faster without using Bluetooth.

Hall et al. [30] proposes a scheme they call *RF Rendez-Blue* that incorporates a RFID-based discovery mechanism for Bluetooth. *Radio frequency identification* [35] (RFID) is used for identification of objects: goods, locations, animals, people, etc. RFID are is not standardized, but the basic idea is that objects are equipped with a *RF tag* which contains information, that can be read by an electronic reader from a short distance. RF tags are small, cheap, and can be designed such that the power needed to operate them comes from the reader (*passive tags*, as opposed to *active tags*). The idea in *RF Rendez-Blue* is to install a passive RF tag on devices that needs to be discoverable, containing the Bluetooth device address. Discovery is then done by scanning for RF tags which is very fast compared to Bluetooth inquiry times, and then connecting to the Bluetooth address obtained from the RF tag. Discovery times for RF tags are not mentioned directly, but their graphs assume zero seconds. This is of course not entirely correct, but to our knowledge it only takes a couple of milliseconds to discover and read information from a RF tag. With *RF Rendez-Blue* the discovery time is in practice eliminated.

In our scenario we can equip each mobile phone with a RF tag and the lock device with a RFID reader. RF tags have many different form factors, even like adhesive stickers measuring only $4 \times 4 \times 0.1cm$ (see Figure 6.1 on the next page) with room for minimum 73 bits (a Bluetooth address is 48 bits). Such a tag can be placed inside the mobile phone without any modifications to the hardware (f.x. next to the battery) and being passive it requires no power from the mobile device's power source. This only needs to be done once, and can also be used by other applications. We do not have any information about the price, but as RFID is expected to be incorporated into clothes and normal grocery products the cost will eventually be low. Moreover, the security of the system is not affected as the RF tag functions like the Bluetooth inquiry procedure, only supplying the Bluetooth address. It does make a relay attack more difficult as the attackers also need a custom RF tag to be discovered by the lock device.

[1]Inquiry also exchanges other information but the address is all that is needed to establish a connection.



(a) Single. (b) Roll.

Figure 6.1: Example of a RF tag, available on peel-and-stick rolls with each tag measuring 4 × 4 × 0.1 cm.
(from: `http://www.ie-oem.com/rfid/rfid-tags-folio.htm`)

Incorporating this into our system would enable our prototype application time to be decreased to only 1.9 seconds on average. The RFID reader on the lock device can be decreased to only 40–50 centimeters, and the door could still be unlocked in less than 2 seconds. Moreover it would save the inquiry scan cost on the mobile device—although minimal on the P800, it may be different on other hardware.

We see RFID as an ideal solution for decreasing the application time in our scenario, without modifications to the mobile phone hardware and with limited cost. There are still some issues that can be handled by a custom device but RFID enables a mobile phone to work very well as a mobile device.

6.2 Building a Custom Device

Although we believe the mobile phone is a good mobile device, a custom designed device may still have advantages over a mobile phone. The mobile phone as a mobile device has the following issues:

- It may be expensive
- It may be too big
- It has limited energy resources

First, the user may not have a mobile phone and may not be interested in having one. To obtain a mobile phone only to be able to unlock doors may be too expensive. Second, depending on the mobile phone hardware, a mobile phone may not be practical to carry around because of its size. Third, the mobile phone is a generic device used for other purposes which also takes valuable energy. This could create a situation where the user cannot unlock a door because the battery was drained by having a phone conversation. A custom device could amend these issues. In this section we will briefly explain some of the approaches that could be taken.

A custom device could also use a more suitable communication technology, as Bluetooth may be too slow or expensive to incorporate in devices. A technology which may in the future be more suited is *ZigBee*[2]. ZigBee is presented as a complementary technology to Bluetooth, especially for small and hardware-constrained devices that are incapable of using Bluetooth. It supports 20–250 Kbps, range from 10–75 meters, and is optimized for slave power consumption—similar to Bluetooth except for the lower maximum transfer rate. The mission is different, as ZigBee promises to work on devices with more than two years of lifetime on a normal battery (like the ones used in a Walkmans, etc.). It furthermore promises discovery times around 30 ms which will make it more ideal in our scenario. The three lowest layers in the ZigBee protocol stack is an IEEE standard, 802.15.4[3]. This standard has recently been approved (May 2003), but the rest of ZigBee is still a work-in-progress. Hence it will take some time before ZigBee can be used, but when building a custom device ZigBee seems interesting.

The processor on the device could either be a general purpose processor fast enough to support cryptography or a low-performance low-cost processor with additional hardware cryptography support. The general purpose processor has to be fast enough to handle software-based cryptography and use as little energy as possible, it is out of our scope to give a general recommendation for this. The other approach is to choose a low-performance processor and equip the device with hardware cryptography support. An example is the *M-Systems SuperMap RSA CoProcessor*[4] that, contrary to its name, is a generic cryptographic co-processor that supports both RSA and ECDSA. The specifications lists a maximum of 190ms for 1024-bit operations. The product can be purchased as Intellectual Property (IP) and incorporated directly in the device. Other similar products are produced by f.x. Analog Devices[5] and National Semiconductor Corporation[6]. Another approach is to use general purpose cryptographic smart-cards. Handschuh and Paillier [31] gives a review of the available cards in 1999, and all smart-cards perform level 1 operations in less than 0.5 seconds. There are three major advantages to incorporating a smart-card reader in the mobile device and use exchangeable smart-cards. First, they are mass-produced which decreases the price. Second, some are tamper-resistant, incorporating features described in Section 3.3.6 to lessen the damage on theft. Third, users never have to handle private keys, as they can be installed by the manufacturer or the external authority.

[2]`http://www.zigbee.org`
[3]`http://www.ieee802.org/15/pub/TG4.html`.
[4]`http://www.m-sys.com/`.
[5]`http://www.analog.com/`.
[6]`http://www.national.com/`.

Regarding the power for the device, the energy source should either be abundant or easily replenished. As size is an important issue, making sure there is enough power is a question of economizing the power there is. For this, Hall et al. [30] further enhances the use of *RF Rendez-Blue* by integrating the RF tag with the rest of the hardware, and proposes to have the device enter sleep mode until awaken by an RFID query (a device *rendezvous*, see [76]). The authors note that depending on the hardware it can take some time for the Bluetooth radio to be ready after sleep mode. With their specific hardware it takes 1.28 seconds but results in only 1 μA power consumption during sleep mode. Combining this with an accelerometer in the custom device, will also avoid unnecessary power usage when the user is not moving (see Section 2.2.3). With these two features incorporated in a custom device, energy will only be used when interacting with a lock device. Still, the battery can be drained though and we propose to use standard type batteries that can be bought in a normal grocery store to enable easy replenishment. This is not our field of expertise, but we imagine that it may be possible to use inductive power in the custom device. Not as the main power source, but if the batteries are drained it could generate enough power to enable one unlocking.

The last issue is how the device obtains the keys. When the mobile device is a mobile phone there are communication possibilities with the external authority through SMS, GSM, etc. The custom device will also need such a communication path to the external authority. This can be done through Bluetooth (or ZigBee), using a device installed at a suitable location (a *base station*) also equipped with Bluetooth and with a communication path to the external authority. This allows the user to retrieve the latest keys using the base station. The base station could be any desktop PC equipped with Bluetooth and the necessary software, allowing the user to use any existing and available PC to retrieve keys from. However, not many desktop PCs are equipped with Bluetooth so the custom device could pose as a generic storage medium through the use of USB or acting like a *Sony Memory Stick*[7] or a similar technology.

Figure 6.2: This picture shows the *Frontel MP301* MP3 player, measuring 8.5 × 3.2 × 1.8 cm and weighing 37 grams.
(from: `http://frontelinc.com.tw/product/mp200.htm`)

[7] `http://www.ita.sel.sony.com/memorystick/`

Looking for examples of existing devices that are portable, has a small form factor, processing power, memory, etc., we find the combined USB-drive/MP3-players. A good example of these types of devices is shown in Figure 6.2 on the preceding page. This device measures 8.5 × 3.2 × 1.8 cm, weighs 37 grams, has 64 Mbytes of storage, and is capable of playing music for 15 hours on a standard 1.5 volts AAA-type battery—we have seen the device being sold for $98.95. Removing most of the storage and equipping the device with Bluetooth, or ZigBee, possibly hardware cryptography support, and other of the above features may be possible at the same price depending on the amount of devices manufactured. The USB-support enables transfer of keys to the device. The form factor is also good, and when the device can play music for that long on a standard battery, it should be capable of unlocking a high number of doors.

We are not engineers, nor product designers, so we will not venture further into this area. It is a topic for further research and there seems to be plenty of possibilities for creating a cheap and well-functioning custom mobile device for our scenario.

6.3 Conclusion

In this chapter we have investigated how we envision an optimal device for our scenario, and have given two approaches to make this. In the first approach we propose to augment an existing mobile phone with a RF tag, in practice eliminating the discovery time in Bluetooth making the total average application time decrease to 1.9 seconds. In the second approach we discuss using other communication technologies, and discuss various techniques that can be incorporated into a custom built mobile device.

We believe that the mobile phone will become a generic mobile platform finding many different usages. With the RF tag the application time is almost negligible, and becomes an even better candidate for the mobile device. However, we do not see the approaches as being complementary, as the most suited device depends on the given scenario and the given user.

CHAPTER 7

Conclusion

In this thesis we have designed a system that allows a Bluetooth-enabled mobile device to unlock a door without user interaction. The design fulfills the requirements of supporting autonomous lock devices, easing key distribution compared to physical keys, having minimal requirements for the hardware, and supporting personalized keys.

To accomplish this we have designed a fully automated discovery procedure using Bluetooth and a secure protocol. To demonstrate and evaluate the design, a prototype using a readily available mobile phone was developed. The evaluation shows that the design consumes minimal power from the hardware and is able to unlock a door on five seconds in average. The prototype fulfills the goals set, and demonstrates that it is possible to use an existing mobile phone as a door-unlocking device.

The system is primarily designed for mail or goods delivery companies, but throughout the thesis, suggestions have been given on how to extend the system to a wider usage. We have also given suggestions on how a simple augmentation of the mobile phone results in a dramatic decrease in the time needed to unlock a door, and suggestions for the design of a custom mobile device.

7.1 Future Work

The following are interesting areas for further research:

- *Mobile device to mobile device communication*

  As sketched out in Section 3.7.2, extending the scenario to incorporate mobile device to mobile device communication, could create new and interesting usage scenarios, for example allowing a mobile device to create access keys and transmit it directly to the intended user's mobile device.

- *Key distribution*

Investigating the issues of key distribution between all three entities in the system. As an example, a lock device could be used as a central distribution point, obtaining keys from mobile devices and distributing them to other mobile devices. Keys could also be disseminated more generally between both lock devices and mobile devices using the approach in [7].

- *Broadening unlocking*

  Instead of unlocking a door, the system could be used to "unlock" other items: a refreshment from a vending machine, a seat in a theater, a parking area, etc.

- *Further implementation of the system*

  Enhancing the prototype to work on other types of mobile phones, possibly using Java with Bluetooth support [51] for platform independence. ECDSA and the shared secret scheme should also be investigated.

- *Experimenting in a real-life scenario*

  Testing the system in a real-life scenario with multiple mobile devices and one or more doors to reveal eventual practical problems with the system.

- *Context awareness*

  The concept of context awareness has only briefly been touched in this thesis. We believe that the use of context could be developed to further decrease energy consumption and increase usability.

Bibliography

[1] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 1996.

[2] Smart Card Allicance. Contactless technology for secure physical access: Technology and standards choices. Technical report, Smart Card Alliance, 2002. URL `http://www.smartcardalliance.org/`.

[3] R. Anderson and M. Kuhn. Tamper Resistance – a Cautionary Note. In *Proceedings of the Second Usenix Workshop on Electronic Commerce*, 1996.

[4] Ross Anderson. *Security Engineering*. Wiley Computer Publishing, 2001.

[5] Ross Anderson and Roger Needham. Robustness principles for public key protocols. In *International Conference on Advances in Cryptology (CRYPTO 95)*, 1995.

[6] Daniel J. Barrett and Richard Silverman. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly & Associate, 1st edition, 2001.

[7] Allan Beaufour, Martin Leopold, and Philippe Bonnet. Smart-tag based data dissemination. In *First ACM International Workshop on Wireless Sensor Networks and Applications WSNA02*, June 2002.

[8] Daniel J. Bernstein. UTC, TAI, and UNIX time, 2001. URL `http://cr.yp.to/proto/utctai.html`.

[9] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. RFC2704 – The KeyNote trust-management system version 2. Technical report, RFC Editor, September 1999.

[10] *Specification of the Bluetooth System – Core*. Bluetooth SIG, 1.1 edition, February 2001.

[11] *Specification of the Bluetooth System – Profiles*. Bluetooth SIG, 1.1 edition, February 2001.

[12] Christian Boesgaard and Allan Beaufour Larsen. Authentication infrastructure for wireless key-door lock system. Classified, 2003.

[13] Dan Boneh and Honav Shacham. Fast variants of RSA. *RSA CryptoBytes*, 5(1), 2002.

[14] Jennifer Bray and Charles F. Sturman. *Bluetooth 1.1 – Connect Without Cables*. Prentice Hall PTR, 2. edition, 2002.

[15] Michael Brown, Donny Cheung, Darrel Hankerson, Julio Lopez Hernandez, Michael Kirkup, and Alfred Menezes. PGP in constrained wireless devices. In *Security Symposium*. USENIX, 2000.

[16] Lars Cederquist. Facts about positioning services, 2002. URL `http://www.ericsson.com/mobilityworld/`. Ericsson Mobility World.

[17] T. Dierks and C. Allen. RFC2246 – the TLS protocol version 1.0. Technical report, RFC Editor, Januar 1999.

[18] Kevin Dixon. Symbian OS version 7.0s – functional description. Technical report, Symbian Ltd., 2003. Revision 2.0.

[19] Johnathan B. Postel (ed.). RFC793 – transmission control protocol. Technical report, RFC Editor, September 1981.

[20] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4), 1985.

[21] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. Simple public key certificate. Internet Draft, January 2000.

[22] Sony Ericsson. P800/P802 white paper. Technical report, Sony Ericsson, 2003.

[23] ETSI. Alphabets and language-specific information. Technical report, European Telecommunications Standards Institute, 1998. ETSI TS 100 900 V7.2.0.

[24] ETSI. Technical realisation of the short message service (SMS). Technical report, European Telecommunications Standards Institute, 1998. ETSI TS 100 901 V7.2.0.

[25] Simon L. Garfinkel. *PGP: Pretty Good Privacy*. O'Reilly & Associates, Inc., 1994.

[26] H.W. Gellersen, A. Schmidt, and M. Beigl. Multi-sensor context-awareness in mobile devices and smart artefacts. *Mobile Networks and Applications*, 2001.

[27] Peter Gutmann. cryptlib security toolkit. URL `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`.

[28] Peter Gutmann. Lessons learned in implementing and deploying crypto software. In *Security '02*. USENIX, 2002.

[29] Peter Gutmann. Encryption and security tutorial, 2003. URL `http://www.cs.auckland.ac.nz/~pgut001/tutorial/`.

[30] Eric S. Hall, David K. Vawdrey, and Charles D. Knutson. RF Rendez-Blue: Reducing power and inquiry costs in Bluetooth-enabled mobile systems. In *IEEE International Conference on Computer Communications and Networks (ICCCN)*, 2002.

[31] Helena Handschuh and Pascal Paillier. Smart card crypto-coprocessors for public-key cryptography. *RSA CryptoBytes*, 4(1), 1998.

[32] Gunnar Heine. *GSM Networks: Protocols, Terminology and Implementation*. Artech House, 1999.

[33] Bernhard Hofmann-Wellenhof, Herbert Lichtenegger, and James Collins. *Global Positioning System: Theory and Practice*. Springer-Verlag, 5th edition, 2000.

[34] R. Housley, W. Polk, W. Ford, and D. Solo. RFC3280 – internet x.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Technical report, RFC Editor, April 2002.

[35] AIM Inc. Radio frequency identification – RFID: A basic primer. White Paper, 1999.

[36] ITU-T. Information technology - ASN.1 encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER) and distinguished encoding rules (DER). Recommendation, ITU-T, 2002.

[37] Markus Jakobsson and Susanne Wetzel. Security weaknesses in bluetooth. *Lecture Notes in Computer Science*, 2001.

[38] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for ”smart dust”. In *International Conference on Mobile Computing and Networking (MOBICOM)*, pages 271–278, 1999.

[39] Dmitry Kasatkin. Affix Bluetooth protocol stack for Linux. URL `http://affix.sourceforge.net`.

[40] O. Kasten and M. Langheinrich. First experiences with bluetooth in the smart-its distributed sensor network. In *Workshop on Ubiquitous Computing and Communications, PACT 2001*, 2001.

[41] John Kelsey, Bruce Schneier, and David Wagner. Key schedule weaknesses in SAFER+. In *The Second Advanced Encryption Standard Candidate Conference*, 1999.

[42] H. Krawczyk, M. Bellare, and R. Canetti. RFC2104 – HMAC: Keyed-hashing for message authentication. Technical report, RFC Editor, February 1997.

[43] Martin Leopold. Evaluation of bluetooth communication: Simulation and experiments. Technical Report 02/03, Institute of Computer Science, University of Copenhagen, 2002.

[44] J. Linn. RFC1421 – privacy enhancement for internet electronic mail: Part i: Message encryption and authentication procedures. Technical report, RFC Editor, February 1993.

[45] Sue Long, Rob Kooper, Gregory D. Abowd, and Christopher G. Atkeson. Rapid prototyping of mobile context-aware applications: The cyberguide case study. In *Mobile Computing and Networking*, 1996.

[46] J. Massey, G. Khachatrian, and M. Kuregian. Nomination of SAFER+ as candidate algorithm for the advanced encryption standard AES. Technical report, National Institute of Standards and Technology, 1998.

[47] Pat Megowan, Dave Suvak, and Doug Kogan. *IrDA Object Exchange Protocol OBEX*. Infrared Data Association, 1.3 edition, January 2003.

[48] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997.

[49] Microchip. KEELOQ authentication products, 2002. URL `http://www.microchip.com/1010/pline/security/index.htm`.

[50] David L. Mills. RFC1305 – network time protocol (version 3). Technical report, RFC Editor, March 1992.

[51] *Java APIs for Bluetooth Wireless Technology (JSR-82) – Specification*. Motorola, version 1.0a edition, April 2003.

[52] Srilekha S. Mudumbai, Mary R. Thompson, Gary Hoo, Abdeliah Essiari, Keith Jackson, and William Johnston. Akenti – a distributed access control system. In *Supercomputing*, 1998.

[53] NIST. Secure hash standard. Technical Report FIPS 180-1, National Institute of Standards and Technology, 1995.

[54] NIST. Digital sinature standard (DSS). Technical Report fips186-2, National Institute of Standards and Technology, 2000.

[55] NIST. Advanced encryption statndard AES. Technical Report fips197, National Institute of Standards and Technology, 2001.

[56] NIST. Recommendation for key management. part 1: General guideline. Special Publication 800-57, National Institute of Standards and Technology, January 2003. URL `http://csrc.nist.gov/CryptoToolkit/tkkeymgmt.html`. Draft.

[57] The Mozilla Organization. Network security services (NSS), 2003. URL `http://www.mozilla.org/projects/security/pki/nss/`.

[58] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J. D. Tygar. Spins: security protocols for sensor networks. In *Proceedings of the seventh annual international conference on Mobile computing and networking*, 2001.

[59] Jonathan B. Postel. RFC821 – simple mail transport protocol. Technical report, RFC Editor, August 1982.

[60] BlueZ project. BlueZ – official Linux bluetooth protocol stack. URL `http://bluez.sourceforge.net/`.

[61] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS, 2003. URL `http://www.openssl.org/`.

[62] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 1978.

[63] B. Schiele and S. Antifakos. Beyond position awareness. In *Proceedings of the Workshop on Location Modeling, UBICOMP 2001*, 2001.

[64] Albrecht Schmidt. *Ubiquitous Computing – Computing in Context*. PhD thesis, Computing Department, Lancaster University, UK, 2002.

[65] Albrecht Schmidt, Michael Beigl, and Hans-W. Gellersen. There is more to context than location. *Computers and Graphics*, 1999.

[66] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons Ltd., 1996.

[67] Bruce Schneier. *Secrets & Lies*. John Wiley & Sons Ltd., 2000.

[68] Adi Shamir and Eran Tromer. Factoring large numbers with the TWIRL device. URL `http://psifertex.com/download/twirl.pdf`. Preliminary Draft, January 2003.

[69] Bluetooth SIG. Bluetooth assigned numbers. URL `https://www.bluetooth.org/`.

[70] Robert D. Silverman. A cost-based security analysis of symmetric and asymmetric key lengths. Technical report, RSA Laboratories, 2001. URL `http://www.rsasecurity.com/rsalabs/bulletins/`.

[71] R. Srinivasan. RFC1833 – binding protocols for ONC RPC version 2. Technical report, RFC Editor, August 1995.

[72] William Stallings. *Cryptography and Network Security*. Pearson Education, 3rd edition, 2003.

[73] *UIQ SDK for Symbian OS v7.0*. Symbian Ltd., 2003.

[74] Microchip Technologies. Remote keyless entry and convenience center reference design with LIN bus interface, 2002. URL `http://www.microchip.com/`.

[75] R. Thayer, N.Doraswami, and R. Glenn. RFC2411 –IP security: Document roadmap. Technical report, RFC Editor, November 1998.

[76] T. Todd, F. Bennett, and A. Jones. Low power rendezvous in embedded wireless networks. In *Proceedings of 1st Workshop on Mobile Ad Hoc Networking and Computing MobiHOC*, 2000.

[77] R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. An overview of the PARCTAB ubiquitous computing experiment. *IEEE Personal Communications*, 1995.

[78] Roy Want. New horizons for mobile computing. Keynote speech at PerCom 2003, 2003. URL `http://www.percom.org/percom_2003/slides/`.

APPENDIX A

Protocol Description

The protocol consists of three different packet types:

ID	Name
1	LockGreet
2	MobileCred
3	Message

All packets share a common header where each packet is identified by **ID**. We start by describing th data types, the common header and the structure for the access key, and then the three packets types.

Data Types

Integers Integers are unsigned if nothing else is stated, and multi-byte integers are transfered in common network byte order (*big-endian*).

Booleans 1-byte integer. False when contents = 0.

Variable Sized Data The first two bytes of every variable sized field is the length of the field, excluding the two bytes themselves.

Time stamps 4-byte integer, seconds since January 1, 1970, 00:00:00 GMT (POSIX time[1]).

[1]There are small errors in the POSIX standard time (as noted in [8]), but with the precision needed in our context it does not matter.

Access Key

Field	Description	Size (B)
Doors	List of door identities	2–
Identities	List of mobile device identities	2–
TS_start	Not valid before this	4
TS_end	Not valid after this	4
Creator	Identity of creator	1–

`Doors` and `Identities` are both lists of variable sized data. The end of the list is denoted by `00`.

Common Header

Field	Description	Size (B)
Version	Protocol Version	1
Type	Unique packet type identifier	1
Size	Size of the packet	2

The `Size` is the size of the packet (excluding the header) in bytes.

LockGreet Packet

Field	Description	Size (B)
ID	Unique ID of the lock	1– ?
Nonce	Unique value	20
SymKey	Allow shared key (boolean)	1
CachedCert	Available certificate for mobile	20
CachedKey	Available access key for mobile	20

`CachedCert` and `CachedKey` are a SHA-1 of a cached identity certificate or access key, and all necessary certificates if multiple authorities are used.

MobileCred Packet

Field	**Description**	**Size (B)**
SymKey	Shared key used (boolean)	1
LockID	Identity of lock device	1– ?
MobID	Identity of mobile device	1– ?
Nonce	Nonce sent in `LockGreet`	20
Cert (optional)	Identity certificate	0– ?
Key (optional)	Access key	0– ?
TS	Current time on mobile device	4
Signature	Signature for packet	1– ?

The certificate and/or key is only sent if the lock device does not have the information already (indicated via `LockGreet`, A). The signature is generated on a byte-wise concatenation of the entire packet except the `Signature` field itself, the same way as done in f.x. TLS (see [17, Section 4.1]).

Message Packet

Field	**Description**	**Size (B)**
TS	Time stamp of last usage of mobile device key pair	4
Num	Message number	1
S (optional)	Encrypted shared secret	0 - ?

`Num` is interpreted as a signed integer, where values below zero is error messages and values above zero is non-error messages. The special case where `Num = 0` is not used. `S` may only be used by the lock device and only when `Num = 1`. The messages sent by the lock device are:

Num	**Description**
1	Authentication Successful
-1	Authentication failed, access key
-2	Authentication failed, certificate
-3	Authentication failed, signature

And the messages sent by the mobile device are:

Num	**Description**
-1	No Key Available
-2	Currently Not Interested in Unlocking

APPENDIX B

Developing for the Symbian OS using Linux

In this appendix we give a short description of how we have developed for the Symbian OS using Linux. We start by describing the installation and configuration of the software development kit, then how to transfer files and applications to the mobile phone, and finally how to debug the applications.

The Software Development Kit

To program for the Symbian OS you will need to get the Windows-based *Symbian OS Software Development Kit*[1] (SDK) for the specific phone you are developing for[2], which can be downloaded freely from the mobile phone companies homepages. The SDK consists of a tool-chain that allows you to build for various platforms with the same source files. The supported targets are the mobile phone and the windows-based emulator. The actual compilation is done with GNU GCC for the mobile phone, and a Windows compiler (Microsoft Visual C++ or Metrowerks CodeWarrior) for the emulator.

Since our normal working environment is Linux, we chose to find a way to make the SDK work with this. There are two ways to make it work: The *GnuPoc* project[3] and *sdk2unix*[4]. They are both a collection of patches and utilities that are applied to the official SDK, to make it work with Linux. They both consist of patches to solve case- and path-problems, but take two different approaches to the actual build process. GnuPoc makes the SDK work the same way with Linux as it does with Windows, using Symbian's own build-tools. sdk2unix

[1]`http://www.symbian.com/developer/SDKs.html`

[2]Which for the moment has a one-to-one mapping to Symbian OS versions: Nokia 9210 = Symbian OS 6.0, Nokia 7650 = Symbian OS 6.1, and Sony Ericsson P800/802 = Symbian OS 7.0).

[3]`http://gnupoc.sourceforge.net`.

[4]In fact this project doesn't seem to have a name, it is just a collection of tools that maintained by Rudolf Knig. But for simplicity we named it sdk2unix. See `http://www.koeniglich.de/symbian_sdk_7.0_on_linux.html`.

makes the SDK behave more like a traditional Unix tool-chain, by supplying a set of GNU Make-rules that allows usage of standard Makefiles.

We have successfully used both GnuPoc[5] and sdk2unix with the Nokia 7650, but have only managed to get sdk2unix to work with the Sony Ericsson P800. For that reason we have chosen to use sdk2unix, which is also easiest for us to use at it uses traditional Unix toolchain. Here is an example of a typical Makefile for sdk2unix:

```
ARCH=p800
EPOC=/usr/local/symbian/7.0

LIBS=   $(EPOCTRGREL)/euser.lib \
        $(EPOCTRGREL)/apparc.lib \
        $(EPOCTRGREL)/cone.lib \
        $(EPOCTRGREL)/eikcore.lib \
        $(EPOCTRGREL)/eikcoctl.lib \

NAME=btecho
OBJECTS=btecho.o btechoapplication.o messageserver.o \
        btechoappview.o btechodocument.o btechoappui.o
TARGET=$(NAME).app
MAJOR=1
MINOR=0
PKGVERS=$(MAJOR),$(MINOR)

PKGFILES=$(NAME).app $(NAME).rsc

U1 = 1000007a
U2 = 100039ce
U3 = 10BE5C34

CFLAGS = -O -I. -I$(EPOC)/include/libc -DUID3=0x$(U3) \
         -DDEBUG -DARCH_${ARCH}
CPPFLAGS += -DARCH_${ARCH}

all:$(PKGFILES) $(NAME).sis
    mv $(NAME).sis $(NAME)-$(MAJOR).$(MINOR)-$(ARCH).sis
$(TARGET):$(OBJECTS)

$(NAME).o: $(NAME).rsc $(NAME).h

clean:
      rm -f $(GENERATED)
```

File Transfer

The actual transfer of files from the host can be done in various ways, where we successfully have tried two: Transfer via Bluetooth with the OBEX ObjectPush

[5]It did require quite a lot of patching though, but the package should be updated with my patches now.

protocol or via copying directly to the filesystem on the phone. OBEX doesn't seem to work very well with the official Linux Bluetooth stack *BlueZ*[6], so we changed to the *Affix*[7] stack which has worked perfectly for us. With the Affix stack installed it works as simply as this:

```
$ btctl push 00:0a:d9:17:6f:0f btecho-1.0-p800.sis
Transfer complete.
5474 bytes sent in 1.0 secs (5474.00 B/s)
```

The phone then receives the file, shows a message about the transfer that let one install it or store it in the Inbox.

Another approach is to use *p3nfs*[8]. which consists of software for both Linux, Sony Ericsson P800 and Nokia 9210/7650. It makes it possible to mount the filesystem of the phone directly with Linux, which makes it possible to copy files directly to the phone. This saves a lot of time, since installed with the above needs a lot of manual input on the phone.

Debugging

The SDK and Symbian web-site advertises on-target debugging using the GNU Debugger GDB[9] or MetroWerks MetroTRK[10]. There is even a patched version of the GDB source code available, but the P800 does not support GDB. And for MetroWerks the following is reported:

```
MW Ron <mwron@metrowerks.com>
Newsgroups:   discussion.epoc.C++
Date:         Tue, 18 Mar 2003 16:58:44 -0500


  On-device debugging support for the P800 is currently being finalized by
Sony Ericsson, who have a working version in their lab. We do not have
control over when this comes to market. Once their final testing is
complete, you will need to have your phone re-flashed, by Sony Ericsson,
in order to use this functionality. As soon as this is available, we
will notify our customers via our website. You can expect that other
Symbian devices, from other manufacturers, will include MetroTRK
functionality in late 2003 and throughout 2004 to support on-device
debugging.
```

At the time of writing, we have not found more specific informations about the status of debugging support. This means that there is no proper debugging support available, and we had to resort to instrumenting our code with statements that printed debug information to some device.

We experimented with three different approaches for the output device. The first simple one was to write debugging to a file, and read the file via an

[6]http://bluez.sf.net/
[7]http://affix.sf.net/
[8]http://www.koeniglich.de/p3nfs.html
[9]http://www.gnu.org/software/gdb/
[10]http://www.metrowerks.com/

application on the phone or on the PC via *p3nfs*. Viewing the file on the phone was slow using the built-in applications and needed swapping between the program being debugged and the viewer. Using *p3nfs* was not working either as it took some time before the PC recognized updates or they only came block-wise. We do not know which is the case but we did not find it feasible to use this method.

Instead of using a local file we investigated how to get the debugging information shown on another device. Our first attempt was to use the serial port, but found that it was not directly accessible. It was locked by an application on the phone used for synchronization with other devices. Instead we used the same approach but using the infrared port, printing debug information to a Palm Pilot, which was the only other device we had with an infrared port. This was however not optimal. First, the location of the infrared ports meant that the devices had different reading angles, making it very difficult and tedious to watch both screens at the same time. Second, we needed to correlate the time of some of the informations with the time on the PC to use for the graphs.

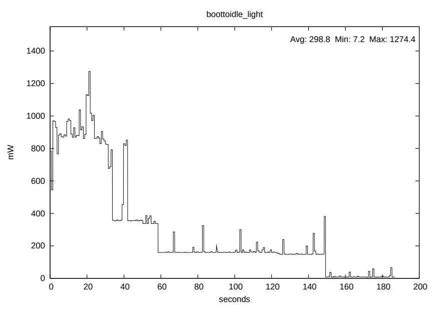
Finally we found the (undocumented) `RDebug`-class (in `e32svr.h`) which has a `Print` function used to print arbitrary data to the serial port. This makes it possible to read the debugging informations on the PC, using it for timing, recording the information, etc. This is the best debugging solution we have found, and we have had no problems using it.

Conclusion

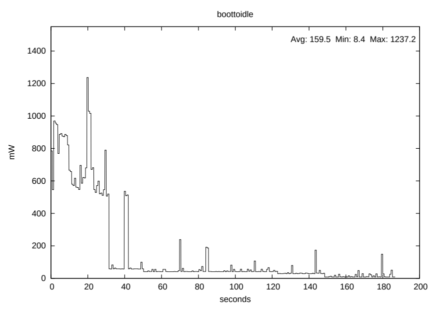
Programming for Symbian OS using Linux is possible using the standard GNU tools. The only real problem is the lack of debugging. The SDK environment for Windows has a Windows-based emulator that has the debugging facilities that we seek. The problem with the emulator is that it emulates the Symbian OS, not the phone hardware, and programs need to be compiled with either Microsoft Visual C++ or MetroWerks CodeWarrior. Besides the lack of an emulator everything works, and we have not encountered problems originating from the fact that we are developing with Linux.

APPENDIX C

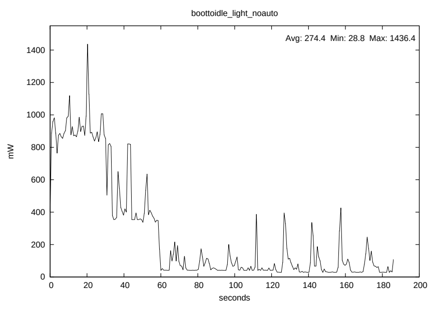
Additional Measurements



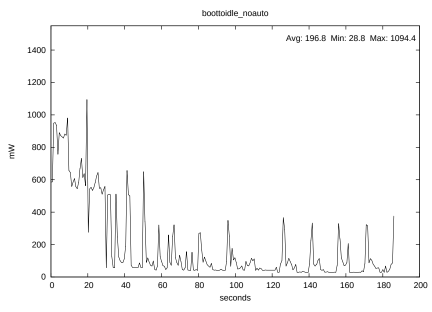
(a) Back-light on, normal mode, power-save.



(b) Back-light off, normal mode, power-save.

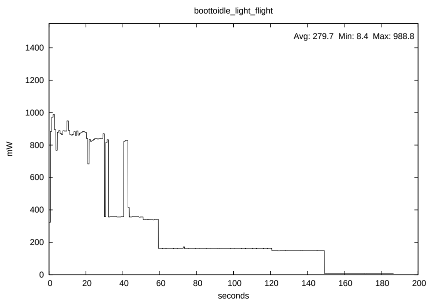


(c) Back-light on, normal mode.

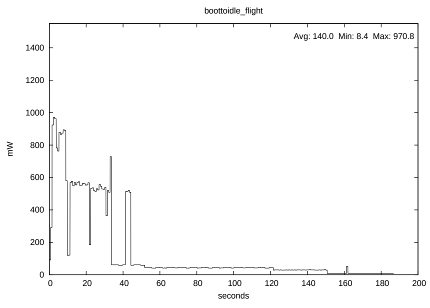


(d) Back-light off, normal mode.

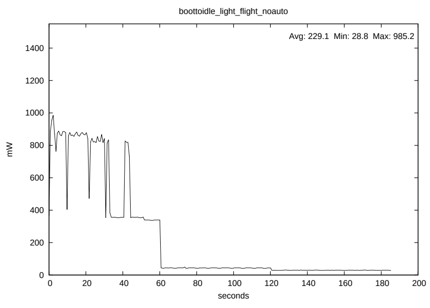
Figure C.1: The energy usage when booting the phone until it goes idle, in normal mode. It is measured with and without back-light, and with or without 2 minutes screen power-save.



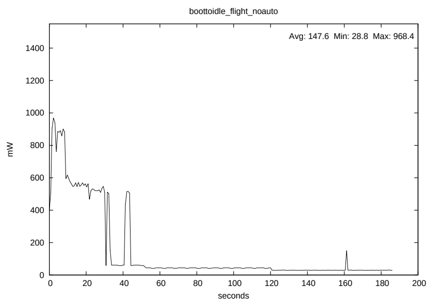
(a) Back-light on, flight mode, power-save.



(b) Back-light off, flight mode, power-save.

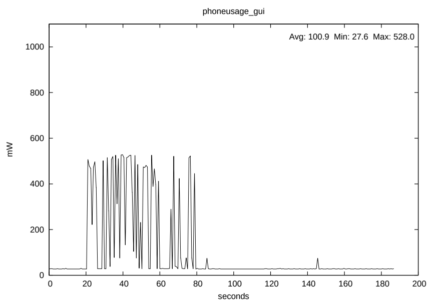


(c) Back-light on, flight mode.

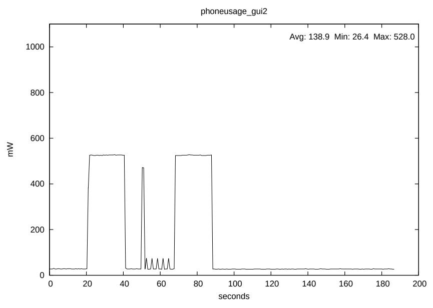


(d) Back-light off, flight mode.

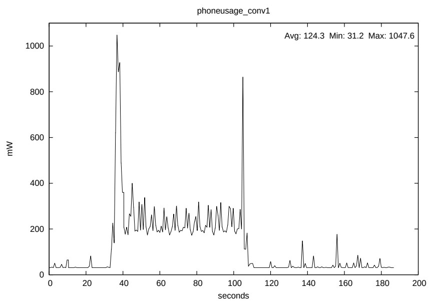
Figure C.2: The energy usage when booting the phone until it goes idle, in flight mode. It is measured with and without back-light, and with or without 2 minutes screen power-save.



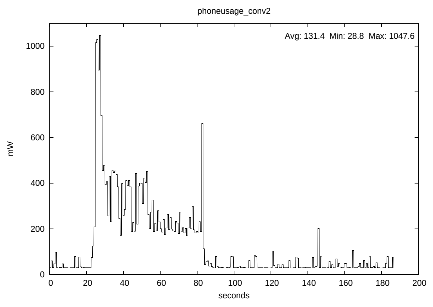
(a) Playing around with the GUI.



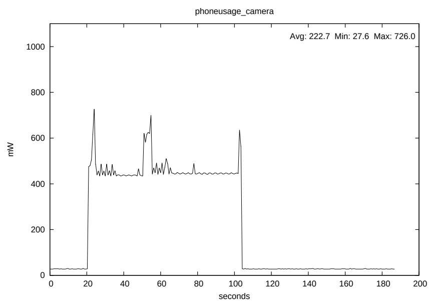
(b) Touching the same place of the screen in a GUI and a console mode program.



(c) A GSM connection, with no conversation

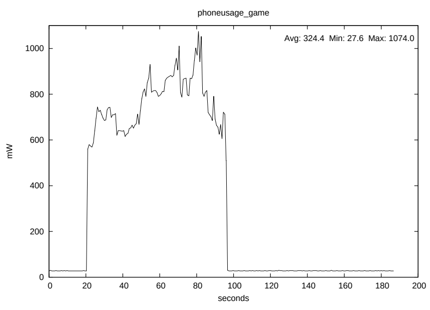


(d) A GSM connection, with conversation

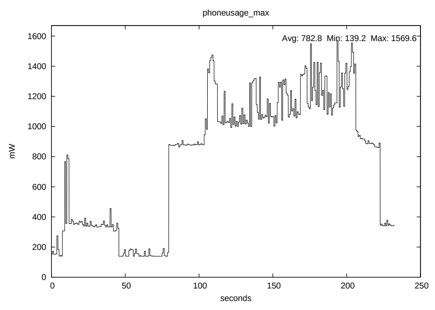


(e) Turning on the camera, wait 20 secs., take a picture.

Figure C.3: The energy usage when using different phone functionalities.



(a) Playing a game



(b) GSM connection, Bluetooth connection, back-light on, and game at same time

Figure C.4: The energy usage when trying to stress the phone. Note the differences in scale.