Design of a cross-platform mobile application for sharing self-collected health data securely with health services

Joakim Kårstad Tran

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

June 2019

Western Norway
University of
Applied Sciences

Abstract

There is a need for sharing and integrating patients' self-collected health data with electronic health records used by clinicians.

A cross-platform mobile application has been developed in order to meet this need. It shares health data securely and is compatible with the Norwegian Centre for E-health Research's FullFlow architecture.

The application's design and its components are studied in order to find out which technologies are suited for this type of application to ensure usability, integration with the Norwegian healthcare infrastructure, and confidentiality and integrity of health data.

Acknowledgements

First, I want to thank my supervisors, Pål Ellingsen, and Alain Giordanengo for all the time they have spent guiding me. Both have been invaluable in the development phase and writing process.

Thanks to Astrid Grøttland and Eirik Åsand at the Norwegian Center for E-health Research for their cooperation and for including me in meetings pertaining to the FullFlow project.

I also want to thank Anders Steen Nilsen for including me in the process of finishing his master thesis and for helping me get started with mine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of mobile health applications and wearable monitoring devices is increasing [1]. Analyzing patients' self-collected data such as blood sugar, pulse, and physical activity can identify troublesome patterns such as elevated heart rate and blood sugar levels [94].

The Electronic health record (EHR) has become an important tool for clinicians, and it is predicted that there will be a greater need for integration of patient collected data and EHRs [80]. This can be beneficial for patients with chronic illnesses such as diabetes. EHR based clinical support systems have been shown to improve glucose and blood pressure control in diabetes patients [91]. By sharing self-collected data, patients can provide valuable information to these systems, and health care providers can be alerted if necessary. In addition, involving patients in their healthcare motivates them to follow health care recommendations better, leading to improved health outcomes [94].

While the benefits of sharing patient gathered data are clear, there are security and interoperability challenges that must be met. Due to the sensitive nature of health-related data, it is vital to keep it secure so that the privacy of patients can be protected [75]. Mobile applications that share health-related data must protect it in two areas [59]:

1. In storage
    - On the mobile device
    - In the cloud
2. Over the communication channel
    - Between sensors and the app
    - Between the app and the cloud

In addition, there should be transparency. Patients should know which data is collected and who it is shared with [61]. Most mobile health applications are exposing patient gathered data by failing to address security and privacy guidelines and regulation such as the General Data Protection Regulation (GDPR). An analysis of 20 popular health apps revealed that only half of them always transmitted sensitive health-related data over HTTPS [92]. Even when HTTPS was used, 30% had an insecure implementation of SSL. Static code analysis suggested that 85% of the apps may have stored sensitive data unencrypted. In addition, many apps share patient data with third parties without explicit consent.

Another issue is interoperability, which depends on the use of consistent standards so that the syntactic and semantic information of health data can be understood by the different systems involved in handling it. Unfortunately, widespread interoperability in health care systems has not been achieved [57]. Although there are many applications for managing health data, most are proprietary and follow different standards [55]. Because of this, it can be difficult to integrate data with health care systems.

1.1 Background

This section starts with an introduction to cloud computing and mobile security. This will give an overview of some of the challenges related to sharing self-collected health data securely. Then, we provide some context by introducing concrete examples of related systems that are planned to be or already are part of the Norwegian health care infrastructure. Finally, the problem description and research questions are presented.

1.1.1 Mobile cloud computing

The primary goal of mobile cloud computing is to give a better experience to users who have devices with limited computational power, storage, and battery capacity [96]. Although mobile devices have limited resources compared to desktop computers, they have improved rapidly over the years [71]. While cloud computing may not be necessary in all cases, it can certainly enhance the capabilities of apps on mobile devices.

For applications where mobile devices can handle computation and storage requirements, a more significant benefit is perhaps availability. Storing data in the cloud improves availability across devices. If data is to be shared and synchronized between various systems, they will not have to be directly linked to a mobile device that may not be online at all times. As an added boon, data in the cloud can be used as a backup in the event that a mobile device is stolen or broken.

That being said, the use of cloud computing can threaten security because data is transmitted over the internet [78]. In addition, data stored in the cloud can be vulnerable because of a greater attack surface. Despite this, there is research indicating that mobile cloud computing used in health care can be both secure and efficient [62] [82] [83].

1.1.2 Risks in mobile security

Mobile devices are used for social networking, shopping, emailing, banking, healthcare services and more [76]. While corporate devices may have restrictions that improve security, the number of personal smartphones that handle critical and sensitive corporate data has been increasing. Sensitive information can be found in SMS messages, photos, and applications. This makes personal mobile devices prime targets for attackers.

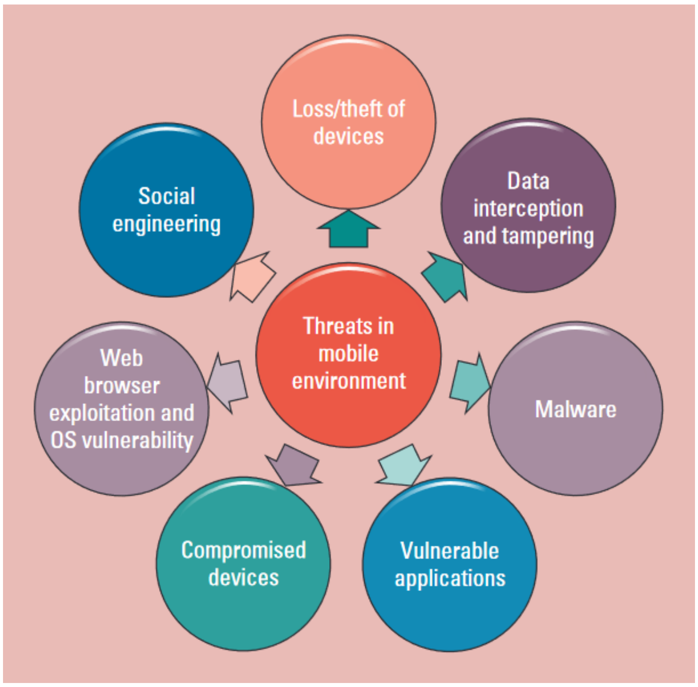

Loss/theft of devices
Social engineering
Data interception and tampering
Threats in mobile environment
Web browser exploitation and OS vulnerability
Malware
Compromised devices
Vulnerable applications

Figure 1.1: Mobile threats [76]

Mobile devices and computers have many common threats like web browser exploitation, OS vulnerability, and social engineering. Some of them, such as device loss or theft, and compromised devices become more prominent on mobile devices.

- Loss/theft of devices: Owners of mobile devices carry them wherever they go. While the portability is convenient, mobile devices are easily lost or stolen. A third of consumers in Canada and the U.S have had their phone lost or stolen [99].

- Data interception and tampering: Mobile devices typically communicate wirelessly, and many use public Wi-Fi hot spots that can be spoofed [67]. This makes mobile devices especially susceptible to data interception and tampering of data transmitted over the internet.

- Malware: Malicious software can be written for the purpose of collecting user information, sending premium-rate SMS messages, credential theft, and ransom [66]. The increasing amount of malware attacks makes this a serious threat.

- Compromised devices: Normally, iOS users cannot install 3rd party applications [72]. Android users may want to remove vendor-installed software or enhance the OS of their phones. Rooting an Android device or jailbreaking an iOS device solves these issues by giving its owner superuser privileges, but the same also goes for attackers. Compromised devices open for more powerful attacks by removing standard security mechanisms or allowing attackers to bypass them.

- Web browser exploitation and OS vulnerability: Mobile applications may use web technologies, thereby inheriting weaknesses such as cross-site scripting, SQL injection, and session fixation. While mobile operating systems provide security features, some of them have to be implemented by developers, which is something that can lead to issues if done incorrectly.

- Social engineering: Attackers may impersonate a trusted party or assume a role of authority in order to fool users into downloading malware or sharing sensitive information. For example, an attacker claiming to be working for the IT department of a user's organization can send an email asking for their passwords.

Another threat is vulnerable applications. It is closely related to some of the other threats that are listed because they target applications with insufficient security. Developers must take care to secure their apps, but there are many pitfalls. The Open Web Application Security Project (OWASP) lists the top 10 risks that mobile developers have to handle [50].

1. Improper platform usage: Misuse of security features such as TouchID, the Keychain, platform permissions and general violation of best practices can be exploited.

2. Insecure data storage: When developers do not encrypt data and assume that the file system is inaccessible to attackers, they expose data stored on the phone.

3. Insecure communication: Even if applications use HTTPS for secure communication, poor implementations can lead to leaked information.

4. Insecure authentication: Applications can fail to properly authenticate by using backend APIs that accept anonymous requests, using weak password policies, or by lacking other means of identifying users.

5. Insufficient cryptography: Even when cryptography is used, improper implementation can make it easy to break. Hardcoding cryptographic keys, relying on obfuscation and using custom algorithms are all examples of cryptography done incorrectly.

6. Insecure authorization: It is often not enough to just authenticate. All access to protected resources should be restricted unless users have been identified, and then authorized by checking whether or not they have the permissions that are required.

7. Client code quality: Missing or insufficient input validation can lead to buffer overflows and memory leaks.

8. Code tampering: Client-side applications such as mobile apps run in environments that are not under control of the developers. Attackers can modify the code of an application on their device in order to cheat in video games. They can also use social engineering to trick others into installing modified apps with malicious code that extracts sensitive information.

9. Reverse engineering: Bad practices such as hard coding secrets and relying on obfuscation can be exploited by attackers who reverse engineer applications, allowing them to inspect the code. It can also be used to find out what code to modify with code tampering.

10. Extraneous functionality: Developers may write code that is not suited for production, and only meant for assisting the development of an application. Examples include hidden administrator interfaces, backdoors that bypass authentication, passwords in comments and debug configurations that output log files.

1.1.3 Helsenorge

Helsenorge is a web portal for health services in Norway. It is a public service run by the Norwegian Directorate of eHealth (NDE), which is a subordinate institution of the Norwegian Ministry of Health and Care Services. Helsenorge governs a patient's kjernejournal (core journal) which is in strict compliance with Norwegian health and privacy regulations [27]. The kjernejournal gathers health data from several sources such as hospitals and national registers. Patients can view and add information like medical history and next of kin.

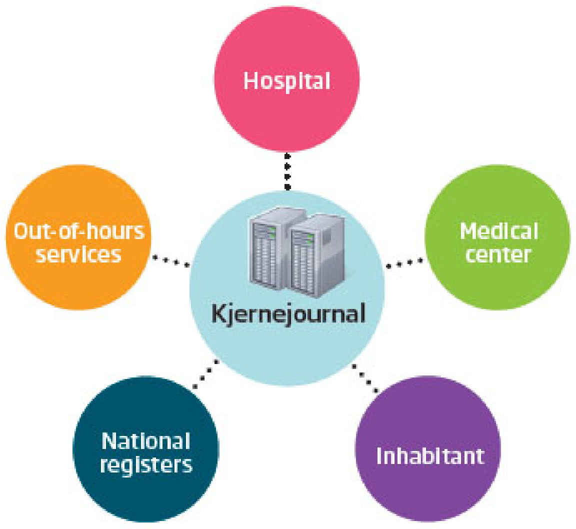


Figure 1.2: Kjernejournal gathers health information from several sources

The kjernejournal functions as personal cloud storage that is independent of any health care provider, and enables medical workers across different organizations to quickly access patient data that is not stored in their EHR. The portal also provides general health advice and information about patients' rights.

1.1.4 Full Flow

Norwegian EHRs do not currently support integration of patient gathered data, and while EHR suppliers are working on it [26], most of them lack semantic interoperability. In order to remedy this situation, the Norwegian Centre for E-Health Research initiated the project "FullFlow of Health Data Between Patients and Health Care Systems", or FullFlow for short. Partners include universities such as UiT The Arctic University of Norway and Aalborg University as well as EHR providers such as Infodoc and Dips.

The goal of the project is to increase the knowledge of how secure technological solutions can contribute to better communication between patients and health services. FullFlow is investigating the medical and financial impacts of full flow between patients, primary health care EHRs and secondary health care EHRs [7].

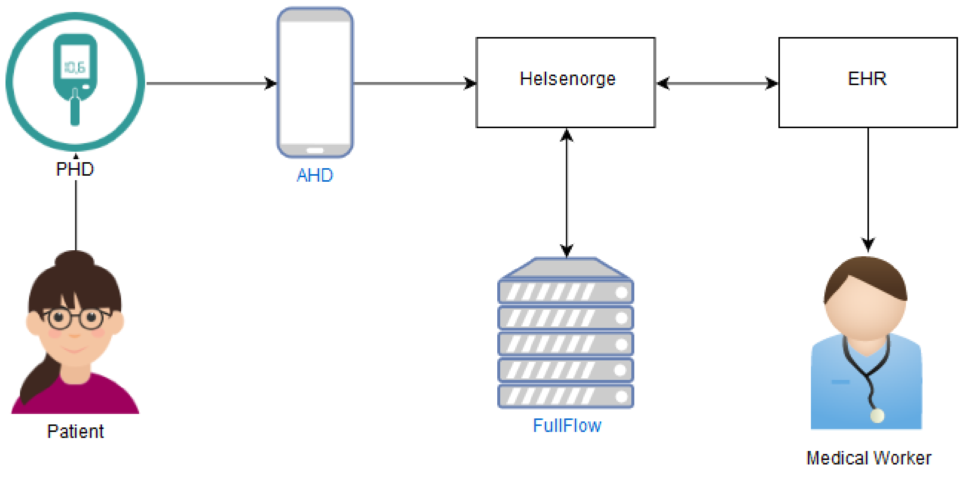


Figure 1.3: Simplified data flow.

Figure 1.3 illustrates a simplified data flow that is part of FullFlow's planned architecture. To gather health data, the patient uses a personal health device (PHD), such as a glucose meter that measures the concentration of glucose in the blood. The data is collected in application hosting devices (AHD) such as smartphones, tablets, or laptops. The AHDs have an application that sends patient data to Helsenorge.

FullFlow pulls data from Helsenorge and processes it, creating visual representations of health data that highlight important information, as seen in Figure 1.4. Medical workers can then request to access the data from their EHR.

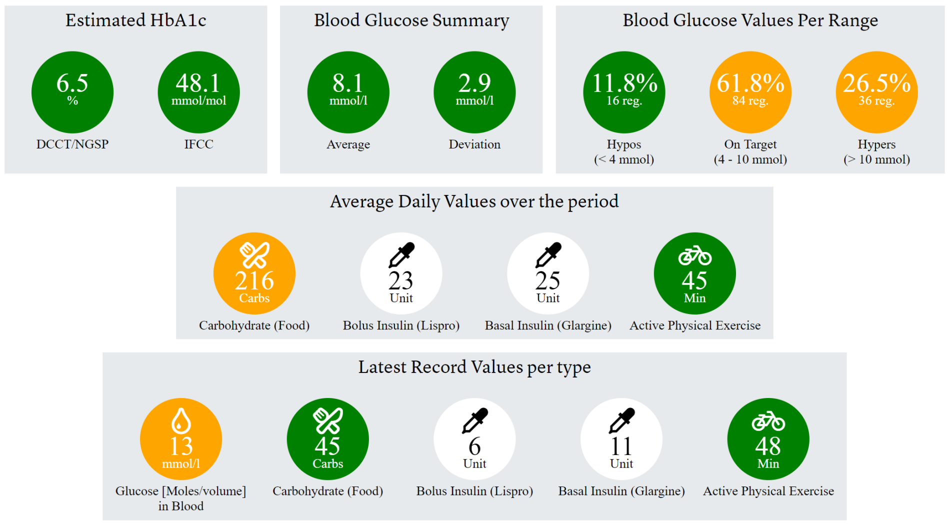


Figure 1.4: Visual report of glucose data

Before a medical worker can access the data, the patient must grant permission by logging into Helsenorge with ID-Porten, which is used for authentication against public services in Norway.

FullFlow is using diabetes as a use case and cooperates with doctors and diabetes patients. The patients use Diabetes Diary, a proprietary Android application that collects self-gathered health data. Diabetes Share Live is used to share the data with clinicians.

1.2 Problem Description

There is clearly a need for a way of sharing patient gathered health data with clinicians. The FullFlow project already has the applications Diabetes Diary and Diabetes Share Live, which can do this, but they are not secure. In addition, they are not interoperable with clinical systems due to proprietary data models. Also, because they are Android only, patients with iOS devices are missing out.

Therefore, we have addressed these issues by developing a cross-platform mobile application with focus on security. The application will support FullFlow by sharing self-collected health data from diabetes patients. In order to make it interoperable with health care systems, we use Fast Healthcare Interoperability Resources (FHIR), a standard format for sharing health data. We also conform to Norwegian health and privacy regulations.

Preserving the confidentiality and integrity of health data is a priority, but the application should also provide good user experience. Even if patients know that the app can benefit their health, they are less likely to use it if they find it inefficient or unintuitive. By focusing on usability in addition to privacy, the app secures not only patients' data, but their health as well.

1.3 Research questions

- Main question: Which technologies are suited for the development of a secure, cross-platform mobile application for managing and sharing health data?

  R1: How can the application preserve the confidentiality and integrity of health data?

  R2: How can one integrate this app into the Norwegian healthcare infrastructure?

  R3: How can one ensure usability?

The three research questions are all related to the main question. To answer these questions, we will review different technologies in order to assess their suitability. In addition to supporting the functional requirements captured by various scenarios, the technologies should secure the application (R1) and facilitate integration with other systems (R2) without discouraging patients from using the app (R3).

1.4 Thesis Outline

The *Introduction* provides context by introducing important concepts and systems. It also presents the problem description and research questions. *Theoretical Background* starts off by giving an overview of related work. Then, the methods used to gather results and answer the research questions are discussed. This will allow the one to evaluate the validity and reliability of the work that is presented in the following chapters. In *Technologies* we describe, compare, and evaluate technologies used in the application. This will give the reader some insight into why certain technologies are suitable for the application. In *Design and Implementation*, we explain how the application solves problems introduced in the first chapter. The high-level architecture, flow of the application, and individual components are also described. In *Analysis and Assessment*, we evaluate the application in line with the methodology described in the second chapter. The *Conclusion* sums up the thesis and relates the analysis and assessment to the research questions.

Chapter 2

Theorethical Background

This chapter begins with a summary of related work that describes other attempts to solve the problem of sharing and integrating self-collected health data. The methodology section will then explain why certain methods have been used to gather and analyze results.

2.1 Related Work

According to Kumar et al. [80], there are not many who have successfully integrated patient gathered glucose data with EHRs, and the existing solutions require custom interfaces, which limits replication. Their paper concludes that it is possible to integrate patient gathered data with EHRs by using existing technologies.

A continuous glucose monitor from Dexcom gathers data and sends it to a smartphone application developed by the same company. An application that directly accesses the database used by an EHR is also used on the mobile device. Both of these applications are compatible with Apple's HealthKit, which enables health data interoperability. The only new software developed was web-based visualization integrated into the EHR.

The authors found that the solution "enabled secure communication, timely access to information, and enhanced interpretation of large volumes of patient device data". It also resulted in better health outcomes for several patients as insulin doses were corrected after analyzing health data.

A drawback of this solution is that the application that accesses the EHR database is provided by that specific EHR. Because of this coupling, if one wants to share patient data with different EHRs, the patient would need several such applications. Another drawback is that the solution can only be used with iPhones, as HealthKit is not available for other smartphones.

The Mobiguide project aims to develop a decision-support system that can be used by patients through their smartphone [94]. The patients wear sensors that monitor and transmit data to the system so that it can provide recommendations regarding actions that should be taken. These recommendations are shared with medical workers.

The system integrates patient gathered data with a personal health record (PHR), and EHR. However, rather than sending patient data to EHRs so that medical workers can access it, data is sent from EHRs to the PHR so that the data can be used for making recommendations.

A major challenge was interoperability. Mobiguide used openEHR, which aims to provide universal interoperability between all forms of electronic health data. It was found that "the use of post-coordinated terms was necessary in order to capture detailed semantics of concepts used (e.g., after lunch (postprandial) blood glucose measurement) and in some cases, certain semantics could not be provided even by post-coordination". However, it seems like this was only an issue when dealing with data not usually found in EHRs.

Infodoc Plenario is an EHR that can be customized to fit the needs of general practitioners, health clinics, and specialists like eye doctors and dermatologists [49]. It relies on an on-site server, but this is being phased out as modules are replaced by cloud services.

Infodoc is a partner in the FullFlow project, and they have worked on integrating FullFlow in Infodoc Plenario with the help of a master student [90]. A message queue which is planned to come from Helsenorge is used to retrieve self-collected health data. Because the data comes in the form of HTML files, the open source browser Chromium is embedded in Infodoc Plenario and used to present the patient data.

2.2 Methodology

This thesis is a case study that investigates how one can develop a secure cross-platform mobile application that shares self-collected health data that will be integrated with EHRs. By building a prototype, we have gathered quantitative data from both the development process and testing of the application, which has given a better understanding of how such an application should be developed (RQ1, RQ2). Quantitative data consists of results from performance tests, which answer RQ3.

Scenarios that capture the core requirements of the application are used for several purposes. Overall, they serve as a focal point and provide structure to both the thesis itself and the work it is based on. This unifies the development and analysis of the application. The scenarios help us identify important elements and make the arguments of the thesis more coherent. The result is a thesis in which background, design, implementation, analysis, and assessment all complement each other.

2.2.1 Functional Requirements

The requirements of the application were defined in cooperation with representatives from the Norwegian Centre for E-Health Research, who is overseeing the FullFlow project. It was decided that the application should have two main functional requirements. 1) collect patient data. 2) send patient data to a server. As the requirements became more understood, they grew more fine-grained:

1. Collect patient data
2. Persist patient data
3. Authenticate patient
4. Share patient data

Because FullFlow is focusing on health data from diabetes patients at the moment, the mobile application must be able to handle this type of data. Specifically, the app handles blood glucose measurements in the FHIR format.

The collection of patient gathered health data was deemed out of scope because of time constraints. The rest of the requirements were used as scenarios.

Situation	A patient has health data collected from a PHD such as a glucometer. The patient is able to import the data into the application, either automatically by integrating the app and PHD, or by entering it manually.
Actors	Patient
Goals	The patient wants to have access to his health data at all times, even when offline. He also wants to minimize the risk of losing data that has not yet been shared with the cloud. Therefore, the patient wants the application to persist his health data after importing it. This must be done securely.
Events	1. A patient imports health data. 2. The app stores encrypted health data.

Table 2.1: Persisting patient data scenario (S1)

Situation	When the Helsenorge server receives patient data from the app, it must store it in the correct kjernejournal. The data itself does not contain enough information for Helsenorge to determine the identity of the patient who sent it.
Actors	Patient
Goals	The server has to find out which patient the data belongs to, and verify that it was the patient in question who sent the data.
Events	1. Before data is shared, the app prompts authentication. 2. The patient enters credentials. 3. The app receives a patient identifier. 4. The app sends the identifier to the Helsenorge server. 5. The Helsenorge server verifies the authenticity of the identifier and uses it to find out which kjernejournal to store the patient data in.

Table 2.2: Authentication scenario (S2)

Situation	A patient has health data on the app and wants to store it in the cloud and share it with clinicians.
Actors	Patient
Goals	The patient uploads data securely to Helsenorge
Events	1. The app retrieves stored patient data that will be shared. 2. The patient authenticates. 3. The app encrypts data. 4. The app sends the patient data to Helsenorge.

Table 2.3: Sending patient data scenario (S3)

The scenarios assisted development in several ways. First off, they described the functionalities clearly, so that all stakeholders were on the same page. Further, they served as subjects of discussion, which aided the discovery of more detailed requirements, and helped us validate our understanding of the domain when in discussion with representatives from the FullFlow project. In addition, they assisted in the development of the architecture, as it was constructed by figuring out which components would be needed to support the scenarios. By splitting the high-level requirements of the application into distinct scenarios, we were able to build the app part by part. After each component had been developed, we validated its design by assessing whether or not it fulfilled the goals of the corresponding scenario.

An agile development process was used to develop the app. First, a high-level architecture containing key components and data flow was sketched. Then, the main technologies, the cross-platform framework, and the database were chosen to fit the architecture and requirements of the application. Each core component, such as persistence, encryption, and authentication was then developed one by one with its own development cycle consisting of analysis, design, implementation, and testing.

2.2.2 Security

In addition to the functional requirements, a key aspect of the application is security. One can not be certain that a non-trivial system is completely secure, as one would have to imagine and account for an infinite amount of possibilities for compromising it. It is, however, possible to gain confidence by means of thorough testing and analysis. There are several approaches that can be used.

Threat modeling is a structured approach that starts off by identifying threats. The threats are then categorized and prioritized before countermeasures are determined. This approach involves looking at the system from the point of view of an attacker as one identifies entry points that can be used to gain access to assets. By doing so, one can uncover architectural weaknesses such as missing authentication in parts of the system accessible to unauthorized users. The threats and attackers described are based on a threat model for mHealth apps [79].

Threats:

T1. Unauthorized learning of health data: Someone gets unauthorized access to health-related data.

T2. Tampering with health data: Attacker modifies data that is stored or transmitted.

T3. Reporting invalid health data: App reports wrong information

Together, the three points cover threats to confidentiality (T1) and integrity (T2), and by extension, privacy (T1) and safety (T2, T3). We will assume that T2 leads to T3. T3 by itself can be influenced by bugs in the app or patients who do not want to report their actual health data. This is out of scope for the thesis, so T3 will only be considered in relation to T2.

Attackers :

A1. Eavesdropper: Captures unprotected network traffic.

A2. Active attacker on the network: Deletes, modifies and redirects data sent over networks. Also attempts to authenticate by brute force.

A3. Man in the middle: Impersonates other actors on the network by taking advantage of improper SSL implementations. The attacker will then be able to read data encrypted with HTTPS in cleartext

A4. Malware developer: Injects malware into the mobile device and uses it to gain access to data from other apps and send it over the internet.

A5. Third parties: If a third party cloud service is used to store health data, it could be exposed.

A6. Attacker with physical access to smartphone: Can extract unprotected data stored on the mobile device.

A7. The user: Can unknowingly put his own health data at risk. The app can be used in ways the developer did not foresee. The user could follow bad practices such as using short common passwords. Powerful, but potentially dangerous features available for advanced users can be misused without the user being aware of the consequences.

The following attackers are not as relevant as the others, and will not be addressed:

1. App developer: May make mistakes that leak information or include malicious code that violates the user's privacy
2. App show owners: Users may not want others to know that they are using a health app. App show owners could potentially expose users through public app reviews.

While mistakes done by the app developer is a concern, this is something that will be addressed implicitly by testing the application. It does not need to be referenced like the other attackers. Dealing with app show owners is outside the scope of this thesis.

The list of threats and attackers have been used to justify design choices throughout the thesis. The threat model is also used in conjunction with the scenarios in order create and classify test cases for penetration testing, which involves taking the role of an attacker, and trying out different attacks on a running system. This gives an overview of how well the app deals with different issues and makes it clear that potential threats to the scenarios have been considered.

2.2.3 Performance

Performance testing has been used for assessing the user experience of the application. Research suggests that it is important to gather quantitative data when measuring the quality of experience [60]. User satisfaction can be assessed by recording it for known levels of performance and comparing these levels to the performance of a concrete application.

There are several things that can be measured when testing the performance of a mobile application. Execution time, memory usage and battery usage are parameters that often give useful values for performance assessment [64].

All of these parameters are relevant, but we have focused on execution time. Compared to the other factors, this will give a higher degree of validity because it directly impacts response time. This is an aspect of performance for which research has given concrete numbers that we can use in order to measure quality the of experience [86]. Based on this, we can set a response time limits for actions taken by a user, and find out how much data the application can handle before reaching them. The results of the performance tests can, therefore, be used to determine how often patient data should be sent from the application for the user experience to be satisfactory. If the frequency is high enough, the inconvenience of having to authenticate too often may outweigh patients' perceived health benefits.

The three scenarios that capture the core functionality of the app rely primarily on database operations, cryptography, and authentication, so these tasks are performed often and will have the most impact on the performance of the application.

The performance of relevant tasks was tested by isolating functions like database insertion and encryption, and measuring their run times multiple times with different amounts of blood glucose measurements in the FHIR format. Real devices were used, and all other applications were closed during testing. Functions were executed with amounts of health data varying from one day to one week's worth of blood glucose measurements. In order to estimate the amount of health data collected in a day, an expert at the Norwegian Centre of E-Health Research was consulted. Real health data collected from diabetes patients involved in the FullFlow project was used.

Chapter 3

Technologies

This chapter describes the technologies that have been used for developing the app as well as the motivation for using them. The technologies support the requirements specified in previous chapters. Together, they enable cross-platform development with authentication, as well as secure data storage and transmission of data in a standardized format. A selection of databases for S1 are compared and evaluated. The authentication technologies OIDC, ID-Porten, and IdentityServer4 support S2. The encryption standard OpenPGP is used in S3. Finally, we describe different cross-platform frameworks that can potentially support all three scenarios and choose a suitable one. Basic concepts related to the technologies, such as symmetric/asymmetric cryptography and authentication schemes are also explained.

3.1 OpenID Connect

OpenID Connect (OIDC) is an authentication and authorization protocol [30]. We use it because it is compatible with ID-Porten, which will be used for authentication (S2). An alternative to OIDC is Security Assertion Markup Language (SAML), which is an XML oriented framework for exchanging authentication and authorization information [31]. SAML is an older technology specifically designed for web browsers, and it has limited support for mobile devices [87]. In contrast, OIDC was developed after smartphones had become prevalent, and it is designed to work with mobile devices.

There are four parties in OIDC:

1. End User (U), a human participant who wishes to authenticate.
2. User Agent (UA), typically a web browser used by the end user to enter and transmit credentials to the OP.
3. OpenID Connect Provider (OP), a server capable of authenticating the end user.

4. Relaying Party (RP), a client application that requires authentication from the end user.

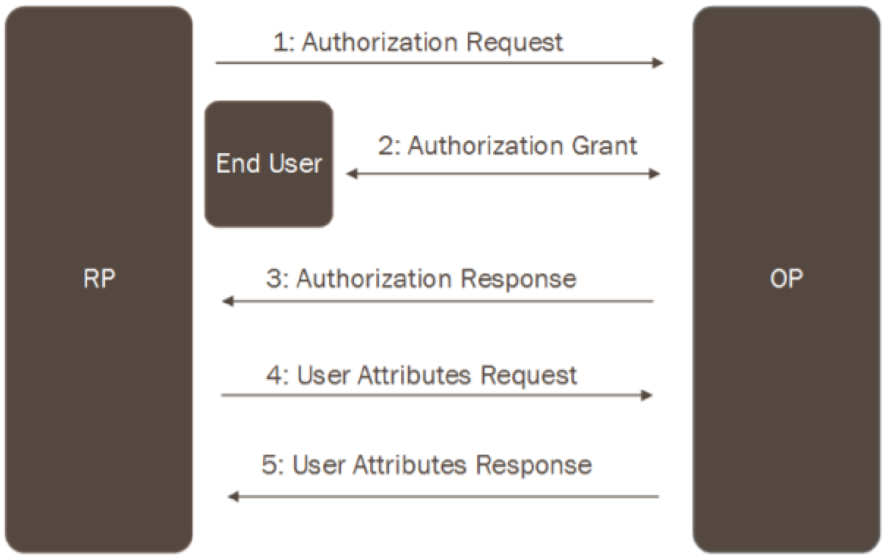

RP
1: Authorization Request
End User
2: Authorization Grant
3: Authorization Response
4: User Attributes Request
5: User Attributes Response
OP

Figure 3.1: OpenID Connect Protocol Overview [81]

In our case, the mobile application (RP) needs the patient (U) to authenticate with ID-Porten (OP) using a web browser (UA).

OIDC specifies several "authentication flows", but they share the same main steps. The RP sends a request to the OP (1). The end user then authenticates via the user agent (2), and the RP is given an access token by the OP (3). With the access token, the RP can request and receive information about the authenticated user (4, 5).

3.2 ID-Porten

ID-porten is an authentication solution operated by the Agency for Public Management and eGovernment (Difi). It used for authentication with public services such as the Norwegian Tax Administration and Helsenorge. ID-Porten allows users to log in with two-factor authentication on mobile devices.

There are other alternatives for authentication, but the FullFlow architecture specifies that patients have to authenticate with ID-Porten. Therefore, our application has to support it.

ID-Porten is useful for several reasons. It is user-friendly because most Norwegians already have an electronic ID, which is essentially a user account for ID-Porten. A consequence of this is that patients will not have to create new user accounts. This will cause them to suffer less from password fatigue, which can cause users to forget passwords and resort to reusing them as a coping mechanism (A7) [56].

Users in our application can be mapped to patients in Helsenorge's system by looking at the electronic ID because we both use ID-Porten as our OIDC-Provider. If this had not been the case, the patient would have to enter his social security number or something equivalent in order to be identified as a Norwegian citizen. This would make the system more vulnerable to identity theft, as Helsenorge would have to trust that the user entered his own social security number. Lastly, because ID-Porten is used for important services such as banking, it has high requirements for security and is thoroughly tested.

ID-Porten supports SAML and OIDC. For mobile applications, the authorization code flow is used with Proof Key of Code Exchange (PKCE).

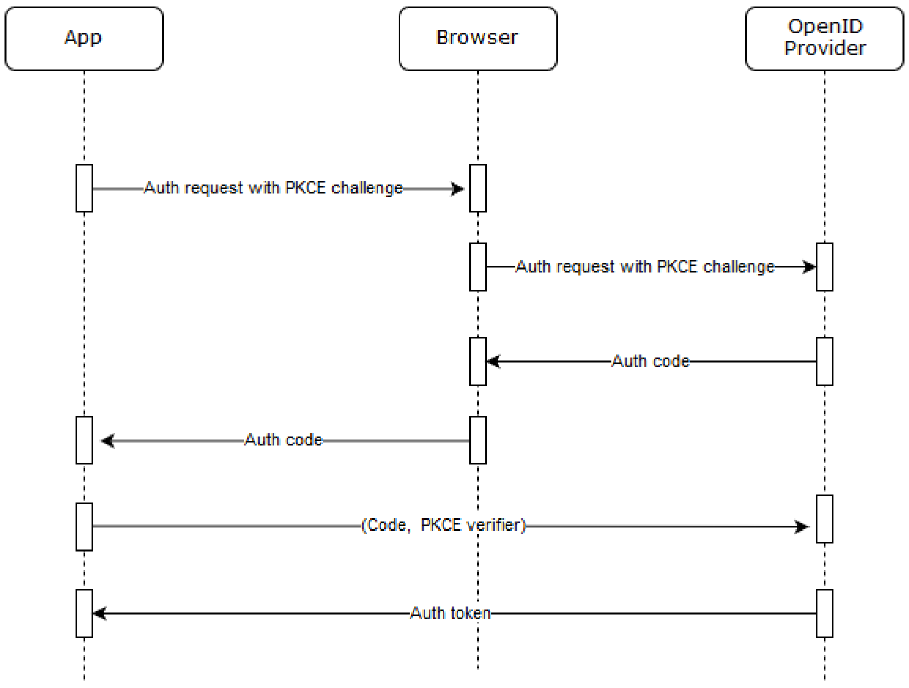


Figure 3.2: Code authorization flow with PKCE

In this scheme, the application generates a code verifier, and hashes it, creating a code challenge which is sent with an authorization code request. When requesting a token, the code verifier is sent so that the OpenID provider can hash it and verify that the token request came from the same client that sent the authorization code request. This prevents other applications on the mobile device from stealing tokens because only the app that sent the initial request has the code verifier.

3.3 IdentityServer4

IdentityServer4 is an open source OpenID Connect and OAuth 2.0 framework for ASP.NET Core 2. It is used as a stand-in for ID-Porten. Integration with ID-Porten requires relatively extensive planning in cooperation with Difi, and they did not want to do this for just a prototype. Because ID-Porten and IdentityServer4 both use the OIDC protocol one only has to do a few minor changes on the client in order to integrate with ID-Porten when that time comes.

There are many OIDC implementations [28]. IdentityServer4 was used for another part of the FullFlow project with success [90]. It also allowed development with ASP.NET Core and deployment on Microsoft Azure, which was helpful for testing because it automatically set up HTTPS with a proper certificate. If we had used another OIDC implementation, we would have to get a signed certificate from a certificate authority. This requires ownership of a registered domain, which we did not have.

3.4 FHIR

Fast Healthcare Interoperability Resources (FHIR) is a standard for exchange of healthcare information. It is based on "Resources", which are representations of healthcare entities such as patients, measurements, and appointments. Resources can be represented in JavaScript Object Notation (JSON) or Extensible Markup Language (XML).

FHIR's intended scope is broad. It's meant to be used globally in many different architectures and scenarios. Because of this, it is infeasible to explicitly include every thinkable property for each resource in the specification, so FHIR supports extensibility. Extensions are optional properties of resources. The way they are structured has an impact on how FHIR data is stored in a database.

FHIR is recommended by The Norwegian Directorate of eHealth [29]. The standard is used in FullFlow, so the app must also use it in order to be compatible with the FullFlow architecture.

3.5 OpenPGP

In order to defend against man in the middle attackers (A3), we use end-to-end encryption. OpenPGP is a non-proprietary encryption standard commonly used for emails [51]. It based on the Pretty Good Privacy (PGP) software. OpenPGP provides authentication, confidentiality, and integrity for messages with the help of digital signatures and encryption [24]. The standard combines two forms of cryptography.

- Symmetric cryptography: A shared key is used for both encryption and decryption.

- Asymmetric cryptography: Every participant has a key pair consisting of a public key and a private key. A message encrypted with the private key can only be decrypted with the public key. Messages encrypted with the public key can only be decrypted with the private key. As the names suggest, private keys are kept secret, while public keys are shared.

If Alice wants to send a message to Bob that only he can read, she can encrypt it with Bob's public key. Because only Bob has his private key, no one else can decrypt the message. Symmetric encryption is more efficient than asymmetric encryption, so Alice will actually encrypt the message with a symmetric key. The symmetric key is normally much smaller than the message and will be encrypted with Alice's private key. Both the encrypted message and the symmetric key is sent to Bob. He can then use Alice's public key to decrypt the symmetric key, which in turn will be used to decrypt the message.

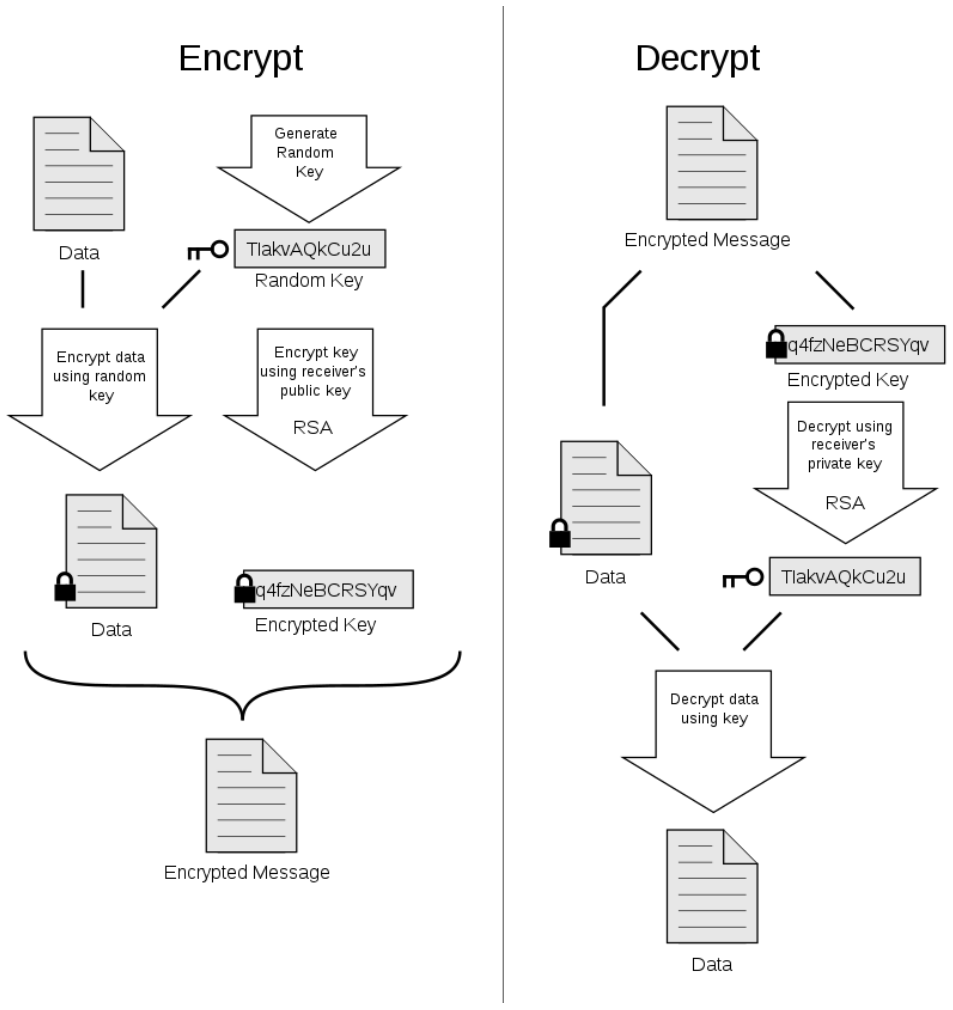


Figure 3.3: PGP encryption [52]

If Bob wants to make sure that a message is sent from Alice, she can sign it by hashing and encrypting it with her private key. She can then attach the signature to the message and send it. Bob can then decrypt the signature with Alice's public key and compare it to the hash of the message. If an attacker modifies the message, the two hashes will not match. An attacker will also not be able to forge a signature without Alice's private key.

3.6 Databases

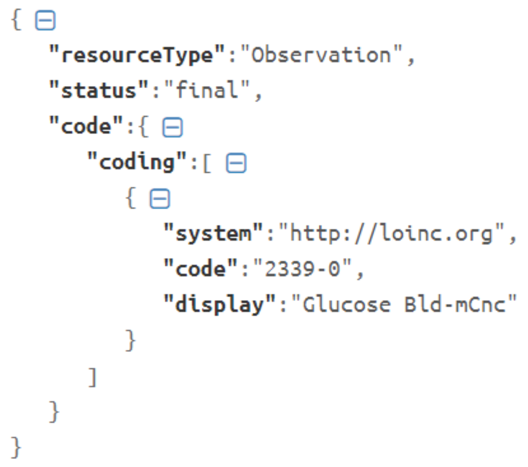
A database is needed for persisting data securely (S1). This section starts with a description of requirements for the database, which will ensure that it is secure, compliant with Norwegian regulations, and suitable for storing health data. Different databases are then compared, and one of them is selected.

3.6.1 Requirements

1. Compatible with mobile applications
2. Document store
3. Local
4. Supports encryption

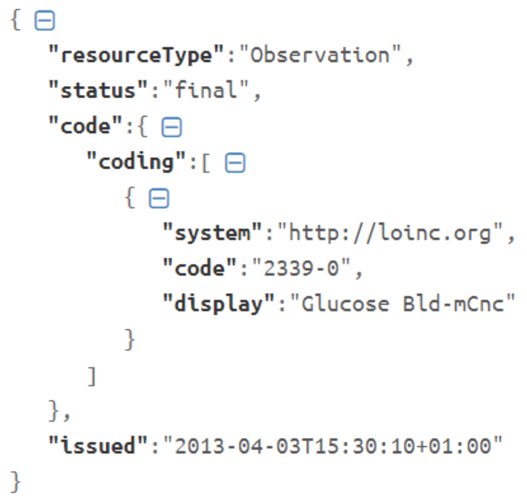
FHIR is a specification that has a lot of optional fields, which is something that needs to be taken into consideration when choosing a database. The Observation resource, for example, can have 21 fields, but only two of them are required [3]. The FHIR specification also supports extensions, which allows anyone to extend FHIR with new resources [2]. We essentially have objects of the same class with varying and unpredictable properties. In order to solve this in a relational database, one would have to use the entity property value (EAV) model, which is an anti-pattern than can lower performance [77].

The solution is to use document stores, a class of non-relational databases (NoSQL) that store data as documents encoded in JSON, XML, or BSON (Binary JSON) [95]. Encoding documents in JSON is convenient because it is one of the supported formats of FHIR. A document is comparable to a row in a relational database where the number of columns can vary. For example, two documents representing observations could be:



```json
{
   "resourceType":"Observation",
   "status":"final",
   "code":{
      "coding":[
         {
            "system":"http://loinc.org",
            "code":"2339-0",
            "display":"Glucose Bld-mCnc"
         }
      ]
   }
}
```

Figure 3.4: Observation example



```json
{
   "resourceType":"Observation",
   "status":"final",
   "code":{
      "coding":[
         {
            "system":"http://loinc.org",
            "code":"2339-0",
            "display":"Glucose Bld-mCnc"
         }
      ]
   },
   "issued":"2013-04-03T15:30:10+01:00"
}
```

Figure 3.5: Observation example with an extra field

The first observation contains only "resourceType" and the two required "status" and "code" fields, while the second also includes the optional "issued" field. The first observation would have a null value for the "issued" field in a relational database, but in a document database, the "column" does not need to be present if there is no value for it because the database is schema-less. There is no rule saying that you need an "issued" column, or conversely, that you cannot have one. Without a schema that constrains what can be stored, one can simply add any JSON object to the database. Consequently, document databases are useful for storing irregular data that would require a lot of null values in a relational database [85] [88].

Flexibility and scalability are the two biggest reasons people have started using NoSQL databases in favor of the traditional RDBMS [85]. Relational databases scale vertically by adding more processors, memory, and storage to a single server where the database is located. NoSQL databases are typically cloud-based and scale horizontally by adding more servers.

Because of this, we have to choose and use NoSQL databases carefully. There are strict Norwegian regulations regarding the storage of health data outside of the country [47]. If the database is cloud-based, one should make sure that the data does not cross any borders. The safest option is to go for a database that only stores data locally on the mobile device. This will protect against A5. In addition, it will enable one to know exactly where the data is located at all times, and how it is secured.

The database also has to support encryption in order to protect its contents from A4 and A6.

3.6.2 Comparison

There are many non-relational databases other than the ones presented here. Examples are BerkeleyDB, Realm, and SQLite. However, this discussion is limited to document stores for mobile applications. PouchDB has a plugin for encryption, but its dependencies are only available in a browser/Node environment, so it is not compatible with certain cross-platform mobile frameworks. MongoDB Mobile does not support encryption out of the box, but requires you to implement it yourself. Amazon DynamoDB and Microsoft Azure Cosmos DB are cloud-based only.

Database	Mobile	Document store	Local	Encrypted
Couchbase Lite	✓	✓	✓	✓
PouchDB	✓	✓	✓	✓/ ✗[1]
MongoDB Mobile	✓	✓	✓	✗
Amazon DynamoDB	✓	✓	✗	✓
Microsoft Azure Cosmos DB	✓	✓	✗	✓

Table 3.1: Database comparison

Couchbase Lite was chosen because it is the only database that fulfills all the requirements on React Native, the cross-platform mobile framework that was selected for the application.

[1]PouchDB's encryption is only compatible with web and hybrid applications

3.7 Cross platform frameworks

It is desirable to make the application available to as many patients as possible. A cross-platform approach was chosen because it allows one to create an application for both Android and iOS without developing on two completely separate code bases, thus reducing development time.

This section compares a selection of cross-platform mobile frameworks. The scenarios in chapter 2 were used for identifying important requirements that the framework had to support. Ease of use and was also considered because it allows one to spend more time on constructive work, which should lead to better results.

There are four types of cross-platform applications: web, hybrid, interpreted and cross-compiled [53]. Web applications run on a browser and hybrid applications run in a web container, limiting native capabilities and performance. This is not the case for interpreted and cross-compiled apps, which are rendered with native components. These two approaches are more suitable for standalone applications where data is processed on the mobile device rather than a server. While we have a client-server architecture, the app may have to process large amounts of data, so interpreted or cross-compiled frameworks are preferred.

3.7.1 Requirements

Primary

- Good support for cryptography and FHIR: Cryptography is used in all scenarios. FHIR is handled in S1 and S3.
- Access to native APIs or option to write native code: If native features or libraries are not available for the framework out of the box, one should be able to develop native modules to ensure that the app will fulfill all requirements.

Secondary

- Good performance: It is important that the framework facilitates good performance, and by extension, satisfactory usability. However, performance can be sacrificed if necessary in order to fulfill the primary requirements.
- Good documentation and/or support from the developer community: Saves time and reduces the risk of not being able to figure out how to solve problems.
- Ease of development: Less time spent fighting the framework leads to more time for constructive work.

3.7.2 Comparison

This section compares the most popular cross-platform frameworks [48] [46]. All of them support native code to some extent. Popular does not necessarily mean good, but it is desirable to have a large developer community. This will lead to more results when searching for solutions to issues, as more people will be discussing it. This reduces the time spent troubleshooting and minimizes the risk of not being able to solve problems.

Xamarin

Xamarin is owned by Microsoft. It uses a shared C# codebase and has two versions. The first is Xamarin Native which includes Xamarin iOS and Xamarin Android, where business logic, data access, and network communication are shared, but UI is coded natively in different codebases. The second is Xamarin Forms, where UI is also shared. Xamarin supports iOS, Android and Windows Phone.

Pros:

- Access to .NET libraries
- A good amount of code reuse with Xamarin Forms.
- Xamarin Native is cross-compiled and Xamarin Forms is interpreted.

Cons:

- Difficult to integrate custom native Android and iOS components.
- Rated as the second most dreaded cross-platform mobile framework in Stack Overflow's 2019 survey.

React Native

React Native is developed by Facebook. It uses the React JavaScript library for building native user interfaces. React Native supports iOS and Android.

Pros:

- React Native was rated the second most loved cross-platform mobile framework in Stack Overflow's 2019 survey.
- Hot reloading
- Interpreted

Cons:

- Certain components are specific to each platform, so some code must be written twice
- Can be difficult to get started with if you are unfamiliar with the React ecosystem

Ionic

Ionic uses Javascript, HTML, and CSS. It is wrapped in a web browser and uses plugins to connect to native APIs. Ionic is best suited for applications that do not use a lot of native features. Supports iOS, Android, Windows Phone and BlackBerry.

Pros:

- A good amount of code reuse
- Supports more than just Android, iOS and Windows Phone.

Cons:

- Hybrid.
- Lacking libraries for secure storage

3.7.3 Comparison

We have not tested all of these frameworks, so parts of this comparison are based on the general impression gained by browsing a variety of blogs, official documentation, and discussion sites. Because some of the information about factors such as ease of development is based on anecdotal evidence, the comparison may not be completely accurate.

	Xamarin Forms	Xamarin Native	React Native	Ionic
Crytography support	⋆⋆⋆	⋆⋆⋆	⋆⋆	⋆
FHIR support	⋆⋆⋆	⋆⋆⋆	⋆⋆	⋆⋆
Support for relevant databases	⋆⋆⋆	⋆⋆⋆	⋆	⋆⋆
Code reuse	⋆⋆⋆	⋆⋆	⋆⋆	⋆⋆⋆
Performance	⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆
Ease of development	⋆	⋆	⋆⋆⋆	⋆⋆⋆
Supported Platforms	⋆⋆⋆	⋆⋆⋆	⋆⋆	⋆⋆⋆

Both Xamarin and Ionic have clear weaknesses. Xamarin is difficult to work with, and Ionic has bad performance as well as poor support for cryptographic libraries. A good reason to select Ionic would be that it is fully compatible with PouchDB but the weaknesses of the framework are too great to overlook. The general consensus seems to be that Xamarin is buggy and difficult to work with. The opposite is true for React Native. In the end, React Native was chosen for its ease of development, which keeps the development time short.

Initially, it was thought that React Native had equally good support for cryptographic libraries as Xamarin. When we found out that the cross-platform OpenPGP libraries for React Native were lacking, we were so far into development that it was too late to reconsider frameworks. A drawback of using React Native with Couchbase Lite is that JavaScript is not a supported language for the database, so it has to be implemented in Swift for iOS and Java for Android. Xamarin has a C# library for Couchbase Lite, which better facilitates cross-platform development. The application initially used Realm as the database, so this was not considered when choosing the framework. It was later discovered that Realm was unsuitable as the requirements of the application became better understood. It is possible that Xamarin would have been chosen if this had been known beforehand.

Chapter 4

Design and Implementation

This chapter presents the design and implementation of the application. The main components are described first. Libraries used by the components are named, and alternatives are discussed. This is followed by an overview of 12 steps, starting with patients collecting data and ending with clinicians accessing it. Then a detailed sequence diagram reveals more of the implementation. References to scenarios, threats, and attackers are used to explain the reasoning behind the design.

4.1 Architecture

The architecture contains three main components: 1) a cross-platform mobile hosting app, used for collecting and centralizing self-collected health data from multiple PHDs and AHDs, 2) an OIDC provider (ID-Porten) for authenticating patients and 3) a server application for analyzing and displaying the data.

Patients interact with the application by directly entering data or through external PHDs or third-party apps, such as Diabetes Diary. In addition, patients use the app to authenticate with ID-Porten. On the other side, clinicians interact with the Helsenorge server in order to consult the data collected by the patients

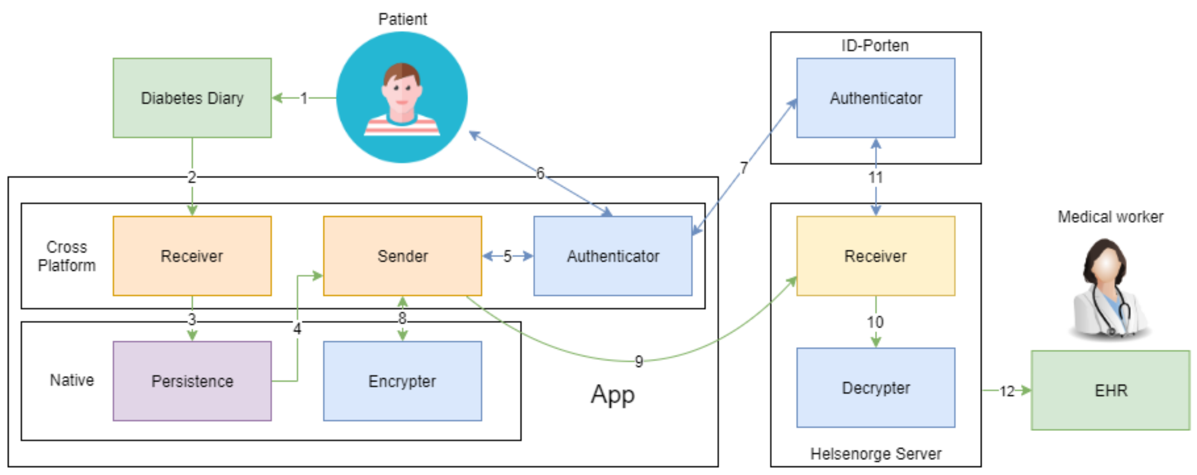


Figure 4.1: Architecture

The app contains multiple modules divided into two categories: cross-platform and native. The cross-platform module contains the code shared between all platforms while the native module consists of the custom implementation for each different platform.

The native module contains two sub-modules: Persistence and Encrypter. Persistence focuses on securely storing and retrieving FHIR artifacts representing medical data in the application (S1). We rely on Couchbase Lite for these actions. CouchBase Lite's encryption deals with attackers A4 and A6 who threaten T1 and T2. There is no cross-platform Couchbase Lite library for React Native, so it was implemented natively in two separate codebases. One for Android, and one for iOS. The password of the database is encrypted in Keychain for iOS and Keystore for Android using *react-native-keychain* [39]. There are other options such as *rn-secure-storage*[41], but *react-native-keychain* has options for biometry, which could be useful in the future. *react-native-securerandom* [40] is used for generating the password.

The Encrypter is an essential component for S3 that encrypts patient data using OpenPGP, thereby preserving confidentiality and integrity by dealing with A1, A2, and A3. A practical solution would be to use an existing OpenPGP library for React-Native. *React-Native-OpenPGP* [25] was tested, but it has a bug which makes it impossible to use a debugger on a React-Native application. It is also much slower than native encryption, as one can see in the performance tests in chapter 5. We opted to implement encryption natively because of this.

In Android, OpenPGP is implemented with Spongy Castle [17], a repackage of Bouncy Castle [13] for Android. The alternative to Bouncy Castle is OpenKeychain API [15], which works by connecting to a remote service. This is not ideal because it requires significant changes to our architecture. In iOS, encryption is implemented with ObjectivePGP, which has undergone a complete security audit from Cure53 [14]. Swift-PGP was considered, but it does not support encryption yet [18].

PGP implementations may support different algorithms. We use RSA with 2048-bit keys for asymmetric encryption, AES with 256-bit keys for symmetric encryption with CFB mode, and SHA-1 for making Modification Detection Codes (MDC). ZLIB is used for compression on iOS, while ZIP is utilized on Android.

The cross-platform module contains 3 sub-modules: receiver, sender, and authenticator. The purpose of the Receiver is to import health data into the app.

The Sender is a central component that is responsible for all the events of S3. It retrieves patient data from the database and utilizes the security features provided by the Authenticator and Encrypter so that it can send data securely.

The Authenticator which uses the OIDC protocol to authenticate users takes care of S2, which is needed for stopping A2 from threatening T2. It communicates with ID-Porten, which allows the app and the Helsenorge server to verify the identity of a user.

The Authenticator uses *AppAuth*, an SDK that implements the authorization code flow with PKCE. *Appauth* is implemented natively, but we consider it cross-platform because we do not have to write the native code ourselves in order to use it. This is because we use a React-Native bridge, which maps cross-platform code to native code that already exists.

Other alternatives are *react-native-oidc-client* [20] and *react-native-oidc* [19]. *react-native-oidc-client*'s documentation does not specify how to configure the library for iOS, and it does not mention PKCE support. *react-native-oidc* lacks options that are useful for development such as allowing/disallowing insecure HTTP requests and specifying token and authorization endpoint separately.

The data sharing process contains 12 steps. The first steps involve S1 and start with data collection. Our app provides two different ways for collecting the data: either by 1) manually registering the data using the application as an AHD directly, or by 2) extracting/receiving data from third-party AHDs or apps. In Figure 4.1, we illustrate the second possibility by collecting the data via the Diabetes Diary app (1-2). In the third step, the collected data is stored in an encrypted database (3).

The next steps occur after sharing of patient data has been initiated, and cover S2 and parts of S3. First, the Sender module retrieves data from the encrypted database (4). Then the patient must authenticate (5). The app opens a browser and a login page provided by ID-Porten appears (6). When the patient has entered his credentials, ID-Porten sends an access token to the app, which can be presented to the Helsenorge server as evidence that the data it receives originates from the authenticated patient(7).

The final steps describe the remaining events of S3. Before data is sent, it must be encrypted. The sender uses the Encrypter module to encrypt the patient data and access token with OpenPGP (8). The data is sent over HTTPS, but with OpenPGP's end to end encryption, only the Helsenorge server can decrypt it even if the security provided by HTTPS is compromised.

The Helsenorge server has a simple REST API with two endpoints. Our implementation runs on a Java EE server. This enabled us to reuse the OpenPGP Java code written for Android. The first endpoint is used by the app to get the information necessary to encrypt data, while the other is used to deliver it (9). When the server receives data, it is first decrypted (10). Then, as part of S2, the Helsenorge server asks ID-Porten to determine whether or not the app's access token is valid (11). If it is the data is analyzed, and medical workers can access it through an EHR (12). HTTPS is used in all steps that make use of the internet in order to protect against A1, A2 and A3.

When persisted locally, patient data is vulnerable to attackers A4 and A6. Even though there are security mechanisms in place to protect the application's database, it is more secure to send data directly to the Helsenorge server without storing it locally. It is recommended to not store sensitive data locally [93]. More traditional client server architectures have been proposed [69] where data is not stored locally, but retrieved from the server when needed.

there are evidently more secure designs than the one we are proposing, but they can come at the cost of convenience and loss of data. When data is only stored on a server, issues will occur when an internet connection is not available. Even though the patient owns the data according to Norwegian health regulations, he will not be able to access it.

Sending patient data can also be hindered. Data can be buffered in memory, but it will be lost if the application somehow closes. If health data contains critical information such as dangerously high blood sugar levels, health personnel will not be made aware of this through the FullFlow system. This can have serious consequences for a patient's health.

Gejibo [68] makes use of the Android Keystore to protect the database key as we do, but it is also protected by a password. The use of a password with sufficient entropy will strengthen security in cases where an advanced attacker attempts to bypass security mechanisms by rooting, but this is inconvenient for the user, [63]. Protecting data with a strong password is no guarantee either, as an attacker can obtain the password with a key logger or social engineering. The patient must already authenticate with ID-Porten. Adding another barrier will make the application cumbersome to use. Patients may not want to use the app at all if the user experience is lacking. Thus, they will not benefit from the full flow of health data between patients and healthcare systems.

Although it is cumbersome for users to enter a password, other solutions could be considered in the future. With biometrics, one can simply scan the user's fingerprint or eye. Not all devices support biometry, but the number is rising [65], so this will become more relevant in the future. It is also possible for the patient to authenticate with ID-Porten and derive a key with a secret from the Helsenorge server. One has to make tradeoffs between security and other requirements such as performance, availability, and usability. The most secure system is not necessarily the best one.

4.1.1 Detailed communication

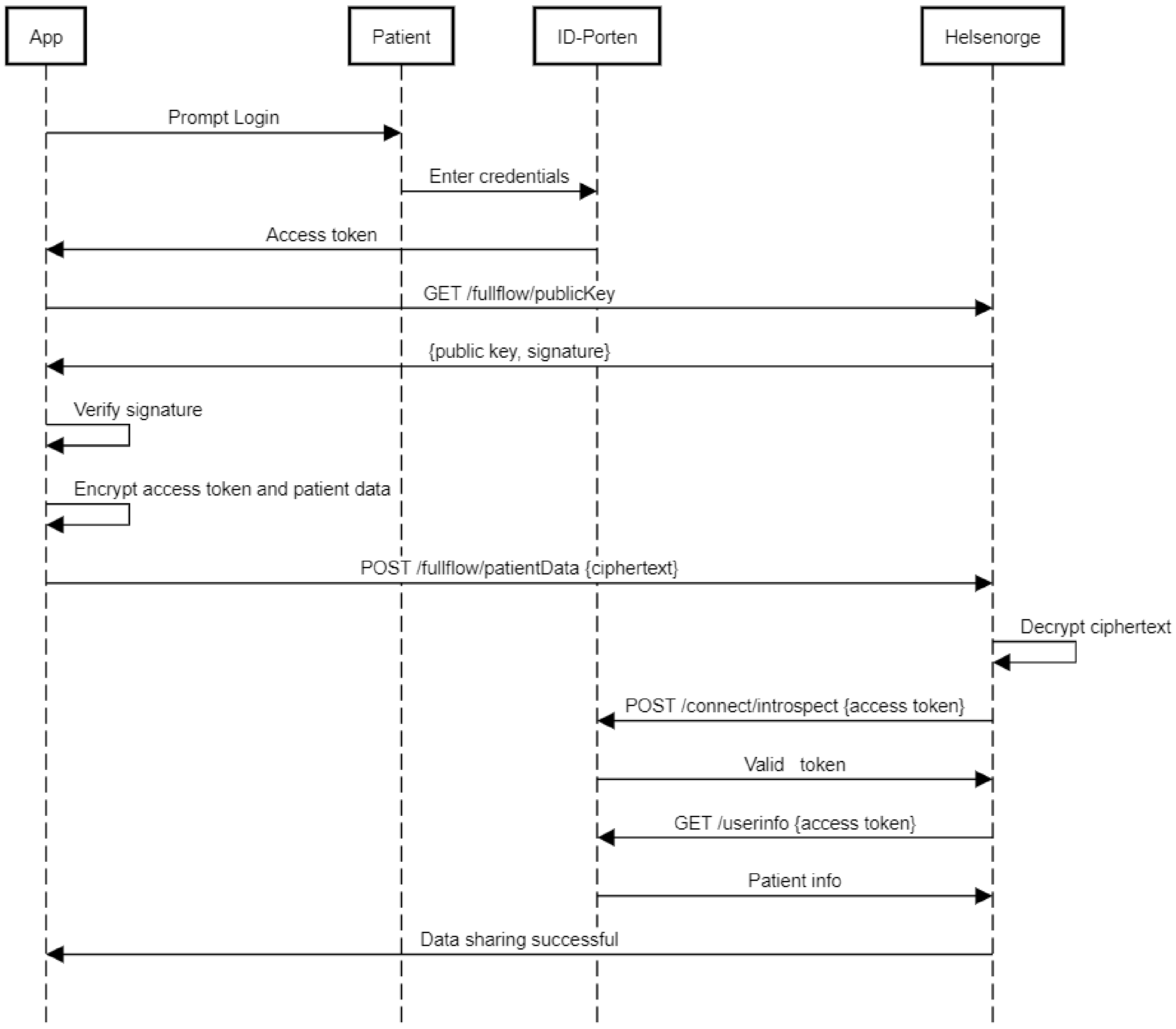


Figure 4.2: Sending data to Helsenorge

Figure 4.2 shows communication between the app, patient, ID-Porten, and Helsenorge when health data is sent successfully. The app needs two things in order to send data to the Helsenorge server. An access token and a public key for encrypting an OpenPGP message. The patient starts by authenticating with ID-Porten, and the app receives an access token. The app will then ask Helsenorge for a public key and a signature that is used to verify the key. When this has been done, the access token and patient data is encrypted with the public key so that only Helsenorge, who has the corresponding private key, can decrypt the message. Helsenorge decrypts the message and asks ID-Porten whether or not the access token is valid. Then, the access token is used to identify the patient so that the data can be stored in the correct kjernejournal. The flow is interrupted if the patient enters incorrect credentials, signature verification fails, decryption fails, or if the access token is invalid.

Chapter 5

Analysis and Assessment

This chapter begins by examining potential security threats. Issues with technologies and libraries are also discussed. The second part of the chapter presents the results and interpretations of the performance tests. Finally, we go over a number of scenario tests that have been used in attempts to find vulnerabilities.

5.1 Encryption of data on the device

The first time the application is opened, a 64 byte array is generated by a cryptographically secure random number generator. The array is converted to a string and used as the database password. As with all other secrets on the device, the key itself must also be encrypted. One can use a password to generate a key, but this is inconvenient for the user. Fortunately. Android and iOS provide encrypted storage, Keychain for iOS and Keystore for Android, that uses a key derived from hardware [22][21]. The security of the database, and by extension, Keychain and Keystore, is essential for S1 because it protects the app against A4 and A6.

An attack that requires root privileges has been found on the Android Keystore, but is limited to software-based implementations [97]. This can be an issue for some users, as not all Android devices support hardware binding. Keystores that are backed hardware-backed are not affected.

Cooijmans T. et al. [63] found that Keystore provides device binding, but not app binding on Android devices with versions 4.1.2 (Jelly Bean) - 4.4.2 (KitKat). This means that keys cannot be exported from the device, but they can be used by A4 or A6 attackers with root access. The latest Android version is 9 (Pie) [36]. Android's Keystore has received updates on every version starting from version 6 (Marshmallow) up until the latest version [37]. *react-native-keychain* only supports Keystore on devices with version 6 or higher. This amounts to 75% of all Android devices [38]. The vulnerabilities were reported to Google, so one can assume that the issues in version 4 were fixed in version

6, but further research should be done in order to confirm this.

The security of iOS's Keychain depends on its configuration. There are four main "accessibility values" which specify when the data in the Keychain can be accessed [4]. In order from most to least restrictive they are:

- AccessibleWhenPasscodeSet: The data in the Keychain can only be accessed when the device is unlocked. Only available if a passcode is set on the device
- AccessibleWhenUnlocked: The data in the Keychain item can be accessed only while the device is unlocked by the user
- AccessibleAfterFirstUnlock: The data in the keychain item cannot be accessed after a restart until the device has been unlocked once by the user
- AccessibleAlways: The data in the Keychain item can always be accessed regardless of whether the device is locked

The three last accessibility values have "ThisDeviceOnly" counterparts, which prevent items in the Keystore from migrating to a new device. This means that data in the Keystore will be lost when restoring from a backup of a different device. [5]

A weakness that can compromise Keychain-protected data has been found in the Keychain on iOS 6 and lower [74]. It requires the device to be jailbroken, and the minimum supported iOS version that React Native supports is iOS 9 [6], but it is still useful to look at the recommendations for protecting against the attack [73]. In order to ensure that the iOS keychain is secure, two requirements must be met:

- The items must be protected with an accessibility value that requires the device to be unlocked for items in the Keychain to be made accessible.
- A passcode that consists of at least 6 alphanumeric symbols is used

In practice, this means that AccessibleAlways should not be chosen as the accessibility value. For maximum security, AccessibleWhenPasscodeSetThisDeviceOnly should be chosen, but this in itself is not good enough to ensure that data in the Keystore is protected if the device is jailbroken because the user may not use a sufficiently strong passcode. Whether or not the device is protected by a passcode is up to the user, but AccessibleWhenPasscodeSetThisDeviceOnly was chosen in order to protect against A7. If a passcode is not used, the application should inform the user that this must be done in order to secure his data. An option to set the accessibility value to AccessibleWhenUnlockedThisDeviceOnly may be considered if patients are willing to take the risk. Even though AccessibleWhenPasscodeSetThisDeviceOnly is the default value, the Keystore has been configured to use this accessibility value explicitly in case the default value changes at some point in the future

Although Couchbase Lite is a NoSQL database, it is built on top of SQLite. The data is encrypted using SQLite Encryption Extension (SEE). The extension encrypts all database content on disk.

When decrypting the database, it is first opened the same way an unencrypted database would be, and one gets a handle, which can be used to query the database. Before any queries can be made, a decryption function must be called with the handle and encryption key as parameters.[9]

If our application has decrypted a database, and an attacker manages to establish a connection to the same database, then the attacker's database connection will obviously not be the same as ours, so it will also have to be decrypted.

In React-Native, the database module is loaded as a singleton object [11], and a database connection remains open until the application shuts down. The application is sandboxed, so its files are not accessible by other applications without root access. [10][12]

All metadata is encrypted [8]. This stops attacks that recover plaintext using the information found in log files and diagnostic tables. [70]

Many databases run on a virtual machine, which makes them vulnerable to virtual machine (VM) image leak attacks, which enable attackers to read in-memory data[70]. SQLite runs bytecode in a virtual machine [23], but as long as the device is not rooted, the VM is protected by the application sandbox.

It is possible to use the Android Debug Bridge, a command line tool, to read data in memory after it has been decrypted [54]. However, this requires USB debugging to be enabled. This is disabled by default and intended to be used by developers only. Because of this, the option to enable it is hidden, and normal users are unlikely to activate USB debugging by accident.

5.2 PGP

PGP is an essential component of S3 that protects patient data against man in the middle attackers (A3), who threaten confidentiality and integrity (T1, T2). The reason for this is that SSL relies on a trusted third party, while our OpenPGP implementation does not. When the app uses SSL, it depends on certificate authorities (CA) to verify the identity of the servers it is communicating with. When two parties are establishing a secure connection with SSL, they exchange certificates, which are public keys that have been signed with the public key of a CA either directly, or through a chain of certifications originating from a CA. Devices have preinstalled root certificates that identify CAs. If an attacker manages to get the private key of a CA, he will be able to impersonate any server that has its certificate signed by that CA. Therefore, we have to trust that the root certificates have been configured properly on the device. We also require the CAs to keep their private keys secure. In addition, CAs must only sign public keys after confirming the identity of the owner. This is a real threat, seeing as there have been several incidents where man-in-the-middle attacks have been possible because a trusted third party could not be trusted [58]

We are protected against these types of attacks because we have a fallback with OpenPGP signatures and encryption. Even if an attacker manages to decrypt data sent over HTTPS, they still need the private key required to decrypt the OpenPGP message in order to obtain the patient data plaintext. If an attacker attempts to impersonate the Helsenorge server and have the app encrypt OpenPGP messages that he can decrypt with his own key, it will fail. This is because we have our own OpenPGP "root certificates" hardcoded in the app. We elaborate on this in the key verification subsection.

We stress that OpenPGP is only used in S3, not S2. This means that a man in the middle attack on S2 could allow an A3 attacker to obtain an access token and send incorrect health data to Helsenorge.

When encrypting with OpenPGP, data is split into 64 or 128-bit blocks. A cipher like AES is used repeatedly to encryp the blocks. This can be done in several ways, specified by a block cipher mode of operation. OpenPGP uses a variation of cipher feedback (CFB) mode that can allow an attacker to determine the first 16 bits of any block [84]. It exploits that OpenPGPs variation of CFB includes an integrity check that can leak information.

The attacker targets a specific block by modifying a ciphertext with a two-byte value D. The attacker sends the modified ciphertext to the person or server that can decrypt it. If the recipient attempts to decrypt the message, and the value D is equal to the two first bytes of the targeted block, the integrity check succeeds. The check fails otherwise. If an attacker can determine whether or not the integrity check fails, he can do a brute force attack by checking a maximum of 2^{16} combinations of D until the integrity check succeeds. If the attacker can figure out that the integrity check succeeds for a value D, he knows that it is equal to the 16 first bits of the targeted block. This means that an attacker can obtain up to 25% of the plaintext depending on the block size. That being said, we use compression, and 25% of compressed plaintext does not necessarily contain enough information for an attacker to determine the uncompressed plaintext. However, there are things we can and should do to fend off the attack.

The key to doing this is to prevent the attacker from knowing whether or not the integrity check has succeeded. Certain implementations of OpenPGP will print an error indicating that the integrity check failed. If the attacker gains access to this error message, the attack will work. In our case, the Helsenorge server will only send a general error message if the encryption fails, so an attacker will not know whether or not the integrity check failed based on the contents of the response.

However, if the integrity check fails, decryption is aborted, and the attacker will get a quicker response from the server. This makes timing attacks possible. The attacker could keep sending ciphertexts to the server with different values for D until one of the responses take much longer than the others and conclude that the D value for the late response corresponds to the first 16 plaintext bits of the targeted block. Whether or not this will work in real life depends how much the internet latency varies, and how long a response from a successful integrity check takes in comparison to a failed one. Regardless, the Helsenorge server can add artificial random latency that is large enough to make the attack infeasible in order to protect against this attack.

SHA-1 is the hashing algorithm that is used in OpenPGP. It is not collision resistant, but it is sufficient for modification detection codes. This is because the plaintext of the message is hashed, not the ciphertext [24]. Therefore, an attacker has no way of verifying that a collision has been found before sending a modified message. In 2017, a collision was found on two PDF files in $2^{61.1}$ operations [98]. An attacker would have to send a tremendous amount of modified messages for one of them to get past MDC. This adds a significant overhead, which makes an attack impractical. In our case, since we also encrypt the access token together with the patient data, the modified message would also have to be decrypted such that it happens to contain a valid token, which is very unlikely.

5.2.1 Key verification

In order to encrypt data with OpenPGP, a public key is needed. Although the application gets the encryption key from a trusted endpoint, an A3 attacker can send his own key. Therefore, the application has hardcoded public keys that can be used for verification. Every public key used for encryption of patient data must be signed with one of the hardcoded keys. Because the verification keys are hardcoded, one cannot modify them in an attempt to inject a key that verifies an attacker's encryption key.

Of course, an A6 attacker can reverse engineer and modify the application in order to insert his own public keys. This would enable a server controlled by the attacker to decrypt patient data sent from the app. However, if he can do this, it easier for the attacker to simply skip the key verification and send data unencrypted to himself. Even if the attacker manages to do this, modifying the code of the app would require a reinstall. This wipes the data stored in the app, so patient data stored prior to the attack cannot be retrieved. For such an attack to work, the patient would have to store data in the modified app.

In the future, it may be possible for patients to download health data backed up in Helsenorge on a new device. This would require authentication, but an attacker could trick the patient into doing this on a modified app. In this case, the attacker would be able to retrieve health data stored prior to app modification. This attack could be handled by giving the patient a warning, stating that he should only download data from the cloud after a fresh reinstall done from the app store. Of course, the warning cannot come from the app, since the attacker could remove the code for it. The warning has to be presented in the browser when the patient is authenticating. Different scopes should also be used for authentication when sharing and downloading data. A scope is an attribute of an access token that states which resources one is allowed to access with the token. If different scopes are not used, an attacker could store the access token retrieved when the patient authenticates before sending data, and then use that token to download data without triggering the warning for the user.

Objective-PGP, the library used for OpenPGP encryption and verification on iOS does not support embedded public key signatures. However, it is possible to sign and verify arbitrary data using a detached signature.

Data with line breaks are not verified correctly by the library. Because valid OpenPGP keys must have line breaks, they have to be pre-processed before signing. This should be done in a way such that different systems and people in working in various environments produce the same output given the same input. If pre-processing is not done in exactly the same way before both signing and verification, the verification fails. One could edit the key manually, but this is prone to errors. Hashing is an easy to use method that Windows, Mac and Linux have built-in. One could argue that this makes signature verification less secure because it introduces the possibility of collisions, but this is not the case, as the data is hashed in the verification algorithm anyways [24]. Therefore, keys are hashed with SHA-256 before they are signed.

5.3 OIDC

It is possible to retrieve a refresh token when authenticating with ID-Porten. The refresh token can be used to get new access tokens without user interaction. This is convenient for the patient, but makes the application less secure. One may question why the app does not store a refresh token in the same manner as the database password in order to secure it. The answer is that having a database is important enough to warrant the increased security risk of storing a secret on the device. The use of refresh tokens, however, is not vital for the application. In the event that an A6 attacker manages to steal a phone and extract data from the Keychain/Keystore, where the database password is located, only T1 is endangered. An A4 attacker could possibly threaten T2 by modifying data in the database before it is sent to Helsenorge. If we store refresh tokens, an A6 attacker would also be able to violate T2 by getting an access token and sending arbitrary data to Helsenorge on behalf of a victim. Therefore, the app will not ask for a refresh token, and the patient must always authenticate manually when sending data.

5.4 Performance

Android tests have been done on an LG K10 (2017). iOS tests were done on iPhone 6, except for the one which compares different methods of measuring time. This was done on an emulator (iPhone 6) running on iMac 17,1 (Intel Core i5, 3.2 GHz, 4 cores). Tests were done on a release build, as recommended in the React-Native documentation [16].

Some tests have a number of observations as a parameter. We assume that the app will collect 288 observations per day, corresponding to one measurement every 5 minutes.

In order to determine how many times the operations should be run, the number of iterations was increased until the difference in median run time between tests consistently deviated by less than 5%. Example: In order to determine the number of iterations needed for measuring native encryption accurately, 2016 observations were encrypted 200, 400, 800, and 1600 times. The median run times were 113, 116, 113, and 112 milliseconds respectively. 400 iterations are good enough, but 800 was chosen for good measure.

performance.now() is the preferred method of measuring time in JavaScript, but it is only available with a debugger attached, which slows down the app. One can use the Date object, but it is not made for the purpose of measuring time accurately.

Encrypting with Date results in an increase of about 5ms for small values when compared to performance.now(). Even though the difference between performance.now() and date.getTime() increases for large values, using Date will give us a more accurate result because it can be used with a release build, which drastically reduces runtime. This can be seen by comparing the run times of Figure 5.1 and 5.2

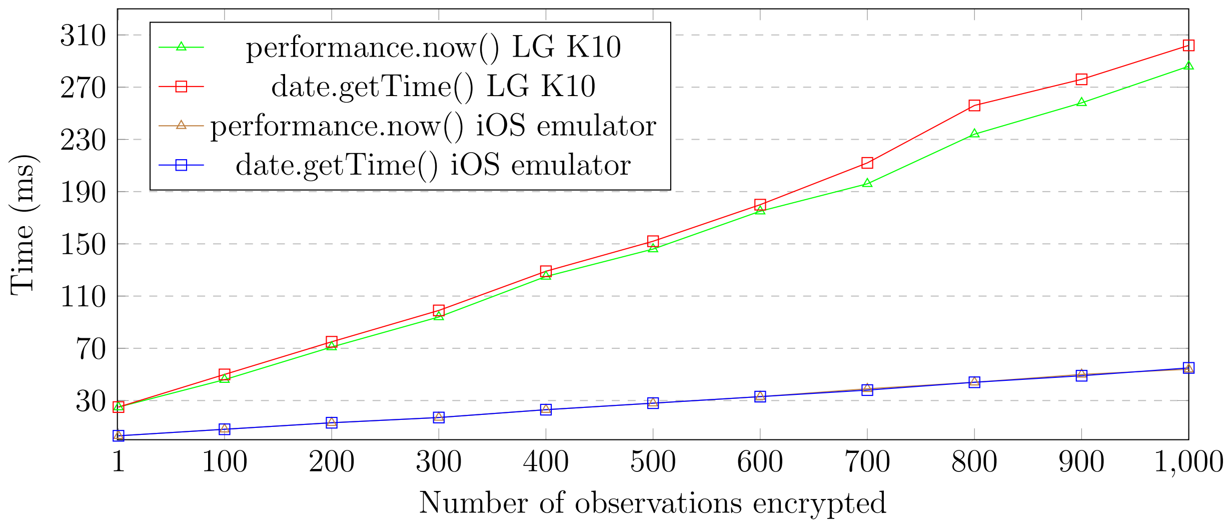


Figure 5.1: Median encryption run time, debugger attached

This performance.now() vs date.getTime test was done on an emulator for iOS because we did not have both a Mac and an iPhone available at the same time. A Mac is required in order to use the debugger.

Native encryption is significantly faster than JavaScript encryption. Native encryption run time grows at a rate of around 40 milliseconds for every 288 observations on LGK10, while JavaScript encryption run time on the same device grows faster by a factor of over 30.

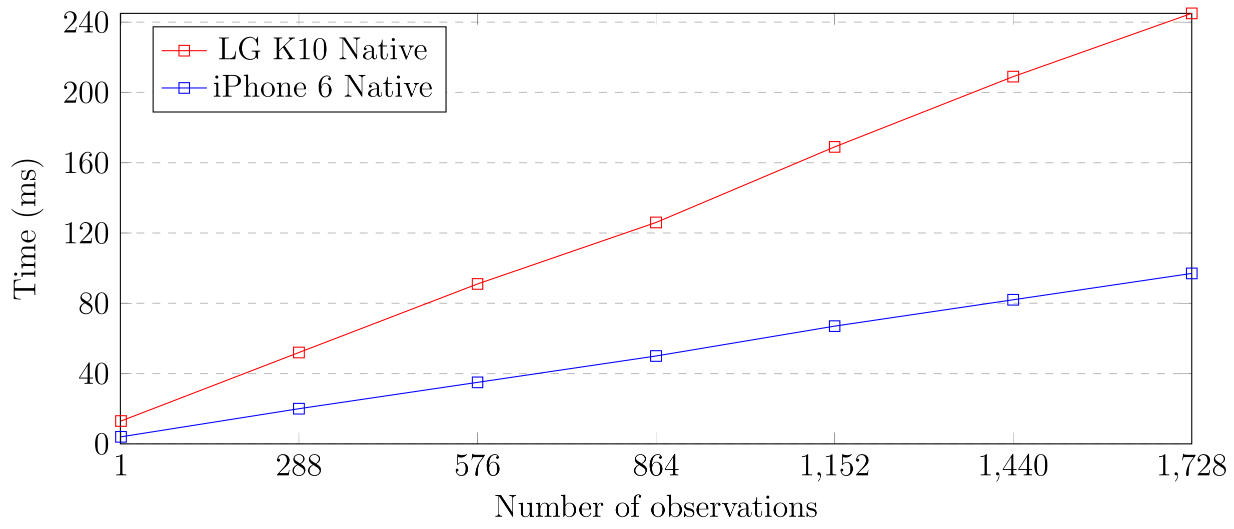


Figure 5.2: Median run time, native encryption

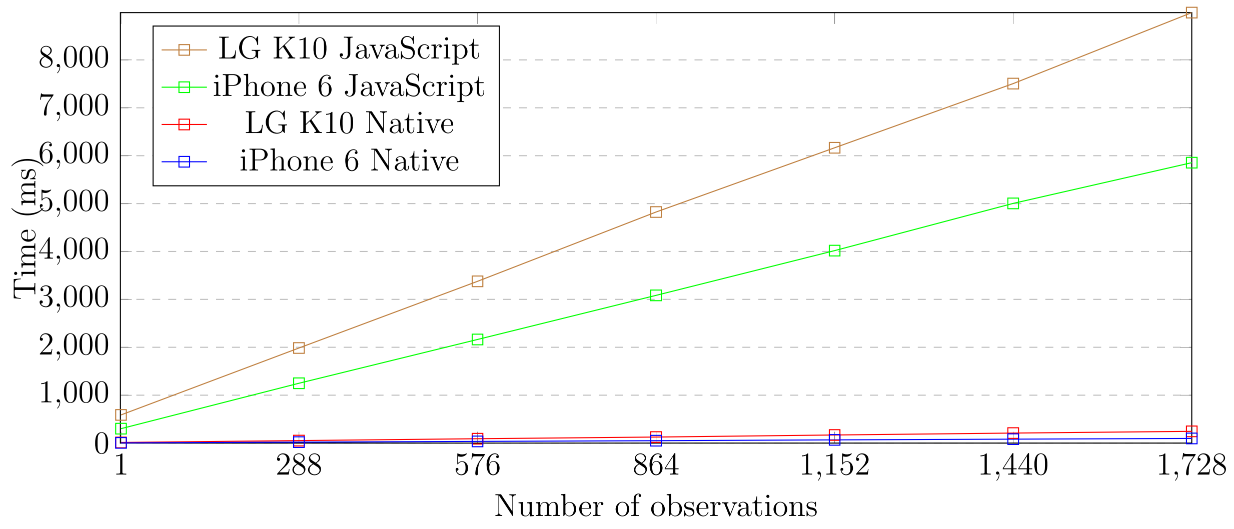


Figure 5.3: Median run time, native and JavaScript encryption

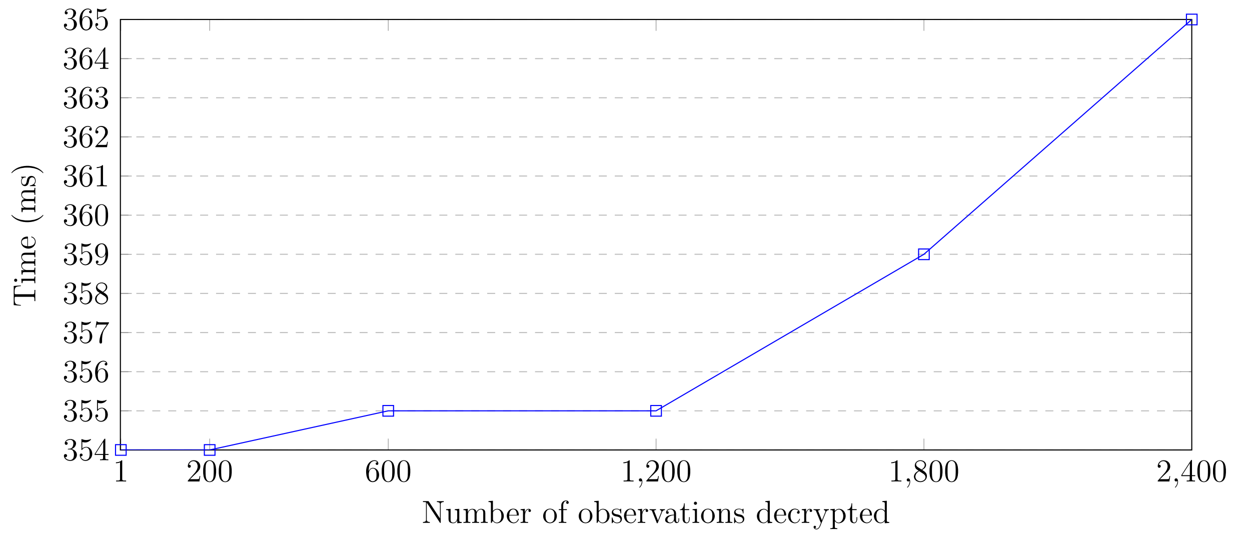


Figure 5.4: Median decryption run time

Decryption run time starts at 354ms for a single observation and does not increase until 200 observations. This is likely due to padding, which makes encrypted messages with 200 or fewer observations the same size. We ran the test with a large number of observations in order to illustrate that decryption run time grows much slower than other operations such as encryption and database queries. The difference between 200 and 2400 observations is only 11ms.

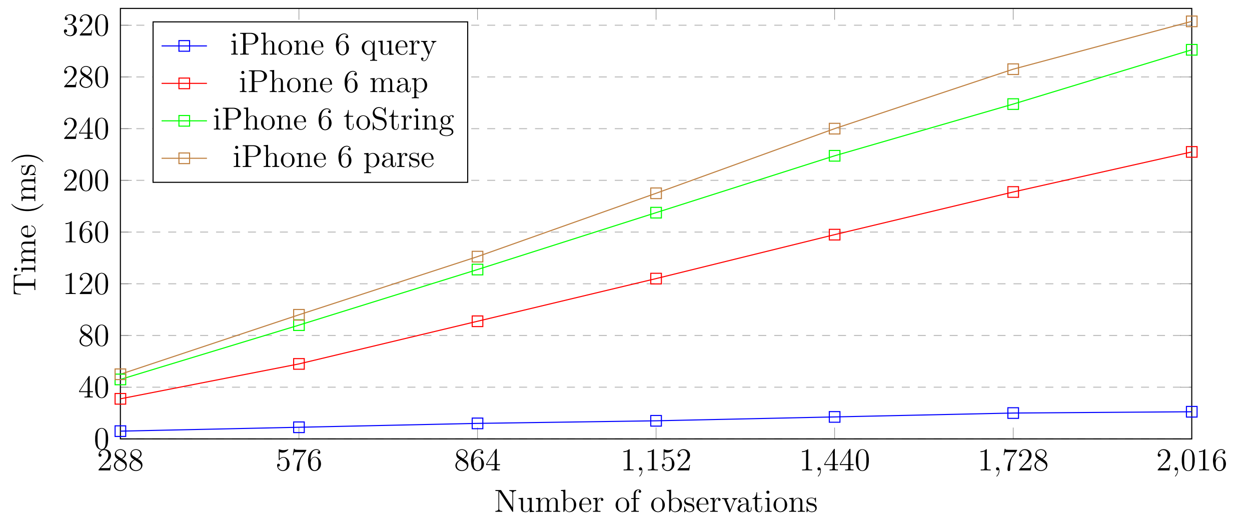


Figure 5.5: Median run time, retrieve observation internals (iPhone 6)

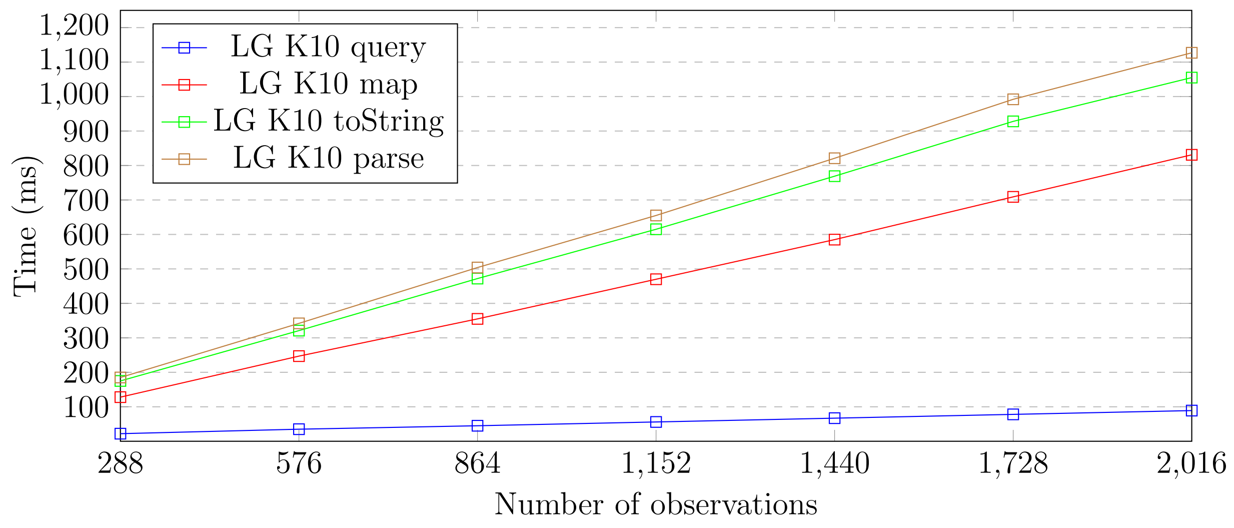


Figure 5.6: Median run time, retrieve observation internals (LG K10)

The "Retrieve observation internals" graph shows the cumulative run time of the operations that are used when observations are retrieved from the database. The first three steps happen in native code. First, a query is made on the database. The results are then extracted from a set of maps, before they are converted to JSON strings. The JSON string is then sent to the cross-platform "world", where it is parsed into a JSON object. The run time of a step includes the run time of previous steps.

A Couchbase Lite query returns a ResultSet which contains a collection of Results. When we make a query so that we get all the fields of a result without specifying them explicitly, each Result is contained in a map. For each Result we have to extract the actual result object from this map. This is the map step. As one can see on the graph, the majority of the run time comes from this step. If we had known which fields we wanted from each object stored in the database prior to making the query, this step would not be needed, reducing run time by around 50%. Although CouchBase Lite can be used to store arbitrary data, it is not very efficient.

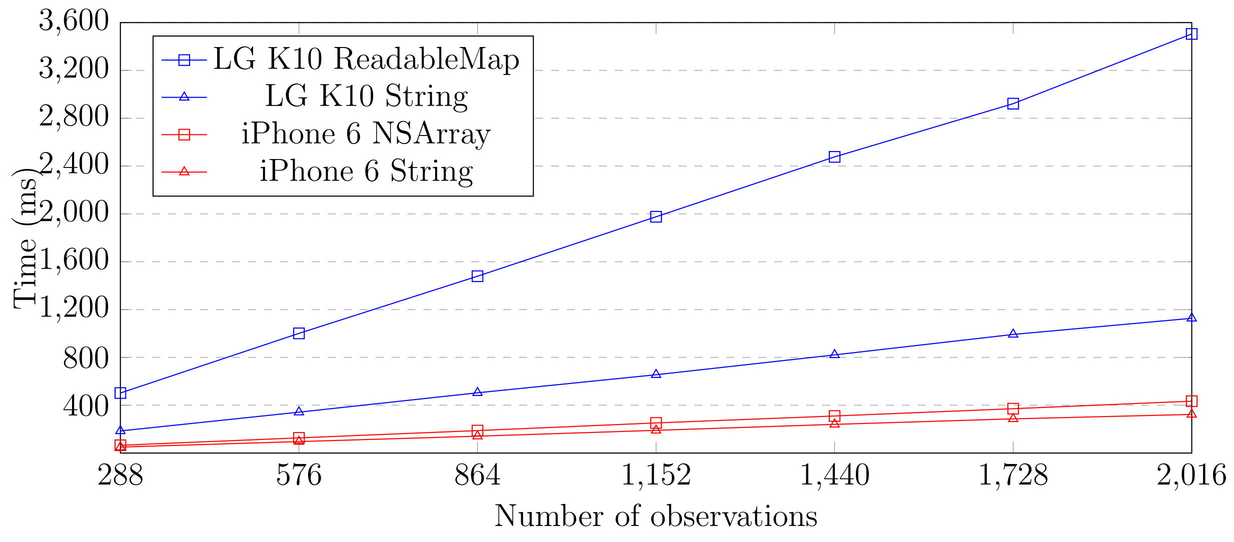


Figure 5.7: Median run time, retrieve observation for different argument types

Couchbase Lite queries return ResultSet objects, which cannot be sent directly to React-Native. Table 5.1 shows Java and Swift types that can be returned from native modules, and their corresponding JavaScript types.

JavaScript	Java	Swift
Bool	Boolean	BOOL, NSNumber
Number	Integer, Double, Float	NSInteger, Float, Double, CGFloat,NSNumber
String	String	NSString
Function	Callback	RCTResponseSenderBlock
Object	ReadableMap	NSDictionary
Array	ReadableArray	NSArray

Table 5.1: React native type conversion

In order to get a ResultSet that represents a collection of objects from Java to JavaScript, one must first convert it to an array of maps and then either convert it to a JSON String, or a ReadableArray. Without a performance test, it is not obvious which method is best, but as one can see on the graph, the String approach is the better option as it is more than twice as fast.

In iOS, an array of maps can be sent to React-Native directly. You do not have to convert it to a ReadableArray. This may be the reason why the difference between the two methods is smaller for iOS. This illustrates how typing mismatches between cross-platform and native languages can impact performance.

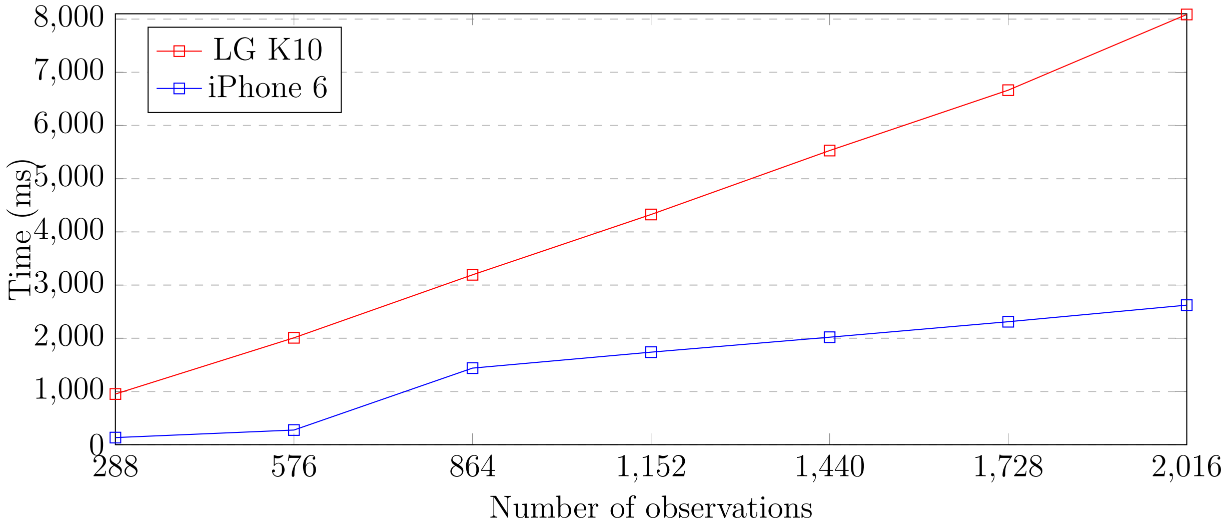


Figure 5.8: Median run time, insert observations

Insertion of observations into the database is the most time-consuming process. Interestingly, the insertion of 576 observations on iPhone 6 does not have the run time one would expect based on the graph. We did not find an explanation for this, but it seems like there is a certain point between 576 and 864 observations where the run time has a sudden jump.

Process	Parameters	Min	Max	Avg	Median
Encryption	Native 1 day of observations	49	944	56	52
Encryption	JavaScript 1 day of observations	1948	3187	2056	1985
Retrieve from database	1 day of observations	179	249	188	185
Signature verification	one signature	10	940	11	11
Signature verification	10 signatures	22	47	24	23
Get data from Keystore		26	85	29	28
Set data in Keystore		29	89	35	33

Table 5.2: LG K10 run time (ms)

Process	Parameters	Min	Max	Avg	Median
Encryption	Native 1 day of observations	18	136	21	20
Encryption	JavaScript 1 day of observations	1218	1487	1261	1249
Retrieve from database	1 day of observations	47	180	51	50
Signature verification	one signature	2	131	4	3
Signature verification	10 signatures	10	35	13	12
Get data from Keychain		4	11	5	5
Set data in Keychain		10	178	17	11

Table 5.3: iPhone 6 run time (ms)

Because the tests were only performed on two devices, which had different processing power, we cannot conclude that iOS performs better than Android in general. However, we make a generalization and let the LG K10 represent low range devices and have the iPhone 6 represent mid-range devices. The LG K10 has a single-core score of 588 [35], while the iPhone is sitting at 2271 [34] on the Geekbench 4 benchmarking software. The highest scores on mobile devices are around 5000, and the lowest are at about 300 [33][32].

The growth of the run times we have measured are linear, so for a set of time limits representing acceptable run times in terms of usability, we can estimate how rarely a patient can import and send data while tolerating the run time of the tasks. Nah F. suggests that web users are willing to wait 2 seconds for information retrieval [86]. Of course, the app is not a web application, and we are sending information, not retrieving it. However, Nah states that her findings are consistent with most literature pertaining to non-internet related computer response times. At response times below 1 second, users' flow of thought is uninterrupted, and they feel that they are navigating freely [89]. Once notices a delay, but will not feel that the wait is unduly.

Based on this, we have two numbers to work with. For tasks that have run times that are dependent on the amount of health data processed, we will find the maximum amount of data that the app can handle without breaking the 2-second limit. Other tasks also have a 2-second limit, but a response time of less than 1 second is ideal. A task begins when the user interacts with the app by pressing a button. The user will then wait for a response which indicates that the task has ended and that he can interact with the app again. The user will interact with the app three times in total.

1. Store data
2. Initiate data sharing
3. Log in

The first task starts when the patient initiates the storing of data and ends when the data has been persisted in the database. The second task starts when the patient initiates data sharing and ends when the log-in page is presented. Finally, the third task starts when the patient has entered his credentials and pressed the log-in button. The app will then send data and wait for a response from Helsenorge, which will end the task.

We calculate the sum of the run time of all the main processes that are involved in S1, S2, and S3 separately in order to figure out the response time of the tasks. This will give a better understanding of how much different processes within each task affect the response time.

Insertion of 2 days' worth of observations takes 2 seconds on LG K10 (2017). On iPhone 6, insertion of 5 days of observations takes an equal amount of time.

We have not been able to get accurate results by attempting to measure the authentication response time in S2. In order to get a good understanding of the authentication run time, we would have to measure how long it takes to open the browser, enter credentials, and receive an access token. We are no able to measure the time it takes to open the browser accurately, as we cannot determine programmatically at which time the browser opens because it is done outside of the application context. A large portion of the time it takes to authenticate is spent by the patient as he manually enters credentials. The time taken by this step differs from user to user. In addition, the layout of the login page and method of authentication depends on the authentication provider. Like the first step, the last one is difficult to measure accurately. This is because it is initiated by the browser, so we cannot determine the start time. We did attempt to measure the response times with a stopwatch, which gave the following results.

Start event	End event	Response time (seconds)
Authentication initiated	Login page opened	< 1
Log in	Access token received	< 1

Table 5.4: Authentication response times

Each measurement was done 10 times, and the response time was less than one second for all of them.

Retrieval of observations from the database can be done in the background at application startup, or data can be cached when imported. We can therefore exclude this when calculating the response time the user experiences when sending data. In the worst case, the user starts sending data immediately after opening the application. This should be okay because the retrieval of observations from the database will happen while the user is authenticating. As long as retrieval from the database does not take longer than authentication, it should be good enough.

Table 5.5 lists run times of the operations in task 3 chronologically. The equations are calculated with simple linear regression using the least squares method. For a number of observations x, one can estimate the response time of a given operation. The first part of task 3, "Get access token", starts when the user presses the login button in the browser. It ends when the access token is received, and the user is taken back to the application. As discussed earlier, we do not have accurate measurements for this, but it takes less than one second. Due to this uncertainty, it is hard to say what the exact response time of task 3 is. Therefore, we use two equations for calculating the total response time in order to get a range of possible values for x. $0.16346X + 699.452 = 2000$ is used for the high end of the range, where "Get access token" is instantaneous and we can send the maximum amount of observations. $0.16346X + 1699.452 = 2000$ is the equation for the low end of the range, in which "Get access token" takes a full second.

Operation	Response time 6 days of data	Response time 27 days of data	Equation
Get access token	1000ms	0ms	< 1000
Get encryption key	22ms		22
Verify encryption key	23ms		23
Encrypt observations	246ms	1066ms	$0.13554X + 12.14286$
Send data[1]	48ms	167ms	$0.01959X + 14.64286$
Decrypt data	359ms	409ms	$0.00833X + 344.66667$
Verify access token	283ms		283
Response from server	See footnote [1]		
Total	1981ms	1970ms	$0.16346X + (1)699.452$

Table 5.5: LG K10 response time on sending data

The x value of each equation is put into the equation for observation retrieval from database in order to find out how long it will take.

Operation	Response time 6 days	Response time 27 days	Equation
Get database key from Keychain	28ms		28
Retrieve observations from database	978ms	4310ms	$0.55097X + 26.14286$
Total	1006ms	4338ms	$0.55097X + 54.14286$

Table 5.6: LG K10 response time on retrieving data

[1]In order to measure the response time of "Response from server" individually, one must measure the start time at the server, and the end time at the mobile device. The clocks of the two systems were not synchronized, so the "Response from server" response time is included in "Send data". This enables one to measure both start and end time at the mobile device.

On the LG K10 (2017), health data should be sent once every week to ensure acceptable response time. One can possibly send data less often, but not more rarely than once every month. As long as the user is done authenticating within 4.3 seconds of application startup, database retrieval will not affect the response time. It is not unrealistic to assume that this will be the case with ID-Porten's two-factor authentication in combination with the relatively slow speed of user input on mobile devices.

We could not do tests involving communication with the Helsenorge server on the iPhone 6. At the time the iPhone tests were done, we only had a laptop with Windows available for hosting the Helsenorge server. We had trouble getting Apple devices to connect to the localhost of Windows machines, so the iPhone could not connect to the Helsenorge server. The observations marked with a star (*) were measured on the Android device.

	Response time		
Operation	13 days of data	56 days of data	Equation
Get access token*	1000ms	0ms	< 1000
Get encryption key*	22ms		22
Verify encryption key	12ms		12
Encrypt observations	204ms	868ms	$0.05357X + 4.42857$
Send data*[1]	88ms	331ms	$0.01959X + 14.64286$
Decrypt data	376ms	479ms	$0.00833X + 344.66667$
Verify access token	283ms		283
Response from server*	See footnote [1]		
Total	1985ms	1995ms	$0.08149X + (1)680.738$

Table 5.7: iPhone 6 response time on sending data

Operation	Response time 13 days	56 days	Equation
Get database key from Keychain	28ms		28
Retrieve observations from database	613ms	2608ms	$0.16106X + 10.78593$
Total	641ms	2636ms	$0.16106X + 38.78593$

Table 5.8: iPhone 6 response time on retrieving data

On the iPhone 6, health data should be sent once every two weeks to ensure acceptable response time. One can possibly send data less often, but not more rarely than once every two months. As long as the user is done authenticating within 2.6 seconds of application startup, database retrieval will not affect the response time.

5.5 Scenario tests

We only had an iPhone available for a limited time. Because of this, all scenario tests for iOS were done on a simulator.

Test	Expected behavior	Threats	Attackers
Persistence			
Database decryption with wrong password	Decryption fails	T1,T2	A4,A6
Database decryption with no password	Decryption fails	T1,T2	A4,A6
App is reinstalled	Keychain/Keystore is empty	T1,T2	A4,A6
App is installed	Unique database is key generated	T1,T2	A4,A6
Authentication			
Authentication over HTTPS with self-signed certificate	Authentication fails	T2	A3
Token verification over HTTPS with self-signed certificate	Verification fails	T2	A3
Authentication with wrong credentials	Authentication fails	T2	A2
Verify expired access token	Verification fails	T2	A2
Verify access token from other provider	Verification fails	T2	A2
Sending data			
Connect to web site with invalid SSL certificate	Connection fails	T1,T2	A3
Verify OpenPGP key with wrong signer key	Verification fails	T1,T2	A3
Encrypt message with OpenPGP	Message is encrypted	T1,T2	A3
Encrypt same message twice with OpenPGP	Different ciphertexts	T1,T2	A3
Decrypt OpenPGP message with wrong key	Decryption fails	T1,T2	A3
Decrypt OpenPGP message with wrong password	Decryption fails	T1,T2	A3

Table 5.9: Scenario tests

Persistence

The function used to create the random database password was used in isolation multiple times, and produced different outputs each time. This test could have been improved by using statistical analysis to determine the randomness of the function. The application was also installed several times. Each time a different key was produced. On re-install, a new encrypted database is created with a new key. Previously generated keys were used in an attempt to open new databases, but this failed.

The default way of opening a database With Couchbase Lite does not require a password. Attempts to open a password protected database this way failed. Opening an unencrypted database with a password also failed. Multiple passwords were used in an attempt to open an encrypted database, but only the correct password was successful.

A database key was stored in Keychain/Keystore and the app was reinstalled. Usually, a new database key is automatically generated if the Keychain/Keystore does not contain one, but this feature was disabled. When attempting to retrieve a database key, none could be found.

Authentication

The OIDC-Provider is hosted both in the cloud with Microsoft's Azure, and locally on the development machine. When hosted in Azure, an SSL certificate signed by a trusted root certificate is used automatically. A self-signed certificate is used on the local server.

With AppAuth on Android, it is possible to specify whether or not requests over HTTP or HTTPS with self-signed certificates should be allowed with the dangerouslyAllowInsecureHttpRequests parameter. On iOS, insecure HTTP requests are disabled by default, and the parameter does nothing.

The dangerouslyAllowInsecureHttpRequests parameter was set to false, and authentication with the local server was initiated. In the browser window, a warning appeared, saying that the connection was insecure. After ignoring the warning and logging in, tokens were not received from the local server. Instead, the error message "Error: Failed exchange token" appeared. There were no issues when authenticating with the Azure server, which had a valid certificate. When authenticating with the local server and insecure HTTP requests were enabled, the warning in the browser still appeared. However, the tokens were successfully retrieved from the authentication server after login.

The Helsenorge server should not trust an authentication server with a self-signed SSL certificate when verifying tokens because this will allow anyone to state that any token is valid. When attempting to verify a token with the local server certificate, the Helsenorge server threw an error, and the response could not be read. Token verification with the Azure server was successful.

The OIDC Provider has two users, Alice and Bob. We attempted to authenticate with different combinations of Alice's and Bob's credentials as well as with random strings. Authentication was only successful when both the user name and password of either Alice or Bob was correct.

An attacker could try to create his own OIDC-Provider and have the patient authenticate with that. When an authentication server creates a token, it signs it with a a shared secret or a private asymmetric key. The only difference between the attacker's OIDC-Provider and the FullFlow OIDC-Provider would be the secret or the private key, assuming they are not compromised

When testing this scenario, two OIDC-Providers with different RSA keys were used. Authentication was first done with the "attacker's" OIDC Provider. The received token was sent along with patient data to the Helsenorge server, which attempted to use the true FullFlow OIDC-Provider for validation. This failed. To account for false negatives, the FullFlow OIDC-Provider was restarted with the attacker key and verified a token signed by the attacker, as expected. This means that a token will only be verified as valid by the same OIDC-Provider that created it. When verifying expired access tokens, the response from the OIDC-Provider was "invalid"

Sending data

So far, only self-signed SSL certificates have been discussed. There are other types of certificates that should not be trusted. A certificate can be expired, revoked, or issued to other domains. All of these have been tested with scenario S3 by making calls to known insecure websites [42]. These sites will either respond with an HTML page, indicating a bad SSL implementation, or there will be an error message saying that one could not connect.

One test made HTTPS calls to a web site with a revoked ssl certificate. The calls were successful on both Android and iOS. This means that an A3 attacker can do a man-in-the-middle attack given that he has access to a revoked certificate from the Helsenorge server. OpenPGP will deal with this when sending health data, but the issue should still be fixed. It seems that both Android and iOS do not use Certificate Revokation Lists (CRL) or the Online Certificate Status Protocol (OCSP) properly in order to detect revoked certificates [44][43][45]. In order to test this, calls were also made to the insecure web site from the chrome browser, completely separate from the app. The browser also accepted the certificate, which indicates that the root of the issue lies outside of the application. It seems that both Android and iOS do not use blacklists properly in order to detect revoked certificates.

It is not certain that the OIDC-library used for authentication will accept a revoked certificate. It may do a check internally, but this has not been tested because the library only communicates with OIDC-Providers. The insecure website, which is meant for testing SSL implementations, does not implement the OIDC protocol. It simply presents an HTML page which says that it is insecure.

To summarize, a revoked certificate can threaten S3, and possibly S2 as well. One should configure an OIDC-Provider with a revoked certificate in order to test whether or not S2 is affected. A solution should be found so that the app will not accept revoked certificates in either S3 or S2. This has not been done due to time limitations

Two new OpenPGP key pairs were created for a hypothetical attacker. One key pair is used for encryption, the other for signing and verifying the public encryption key. When only the public encryption key of the Helsenorge server was substituted by that of the attacker, key verification failed on the app. In order to make sure that the attacker keys were created correctly, and will validate when they should, the Helsenorge verification keys were replaced with the attacker verification key, which made the app verify the attacker encryption key as expected. The Helsenorge encryption key was signed by a Helsenorge signer key trusted by the app, while the attacker encryption key was signed by an attacker signer key not trusted by the app.

Some of the tests are redundant, e.g. Helsenorge verification keys with Attacker encryption keys and the opposite, Attacker verification keys with Helsenorge encryption keys are testing the same thing. Verification should only succeed if the encryption key is signed by one of the keys in the collection that is fed into the verification function. The redundant tests are still included for completeness.

Table 5.10: Key verification test results on Android

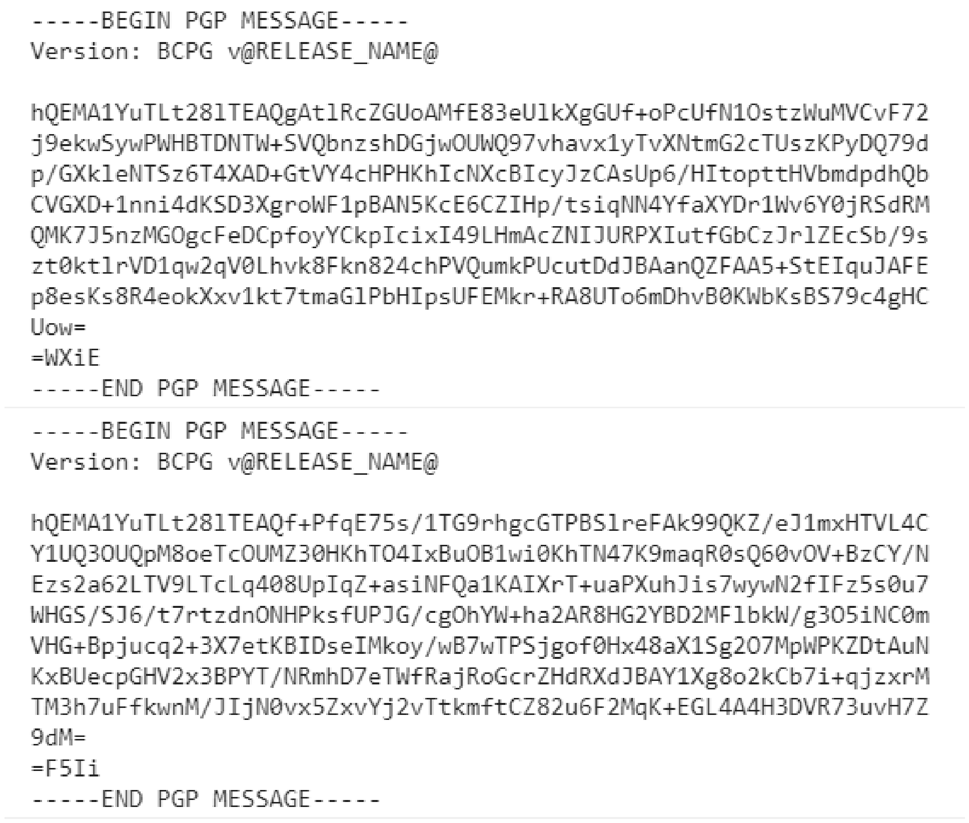
Verification keys	Key to be verified	Verification Expected	Verification Actual
Attacker	Attacker	Succeeded	Succeeded
Helsenorge	Helsenorge	Succeeded	Succeeded
Helsenorge	Attacker	Failed	Failed
Attacker	Helsenorge	Failed	Failed

Key verification on iOS has three parameters: A collection of verification keys, a key to be verified, and a detached signature.

Table 5.11: Key verification test results on iOS

Verification keys	Key to be verified	Signature	Verification Expected	Verification Actual
Helsenorge	Helsenorge	Helsenorge	Succeeded	Succeeded
Helsenorge	Attacker	Attacker	Failed	Failed
Helsenorge	Attacker	Helsenorge	Failed	Failed
Helsenorge	Helsenorge	Attacker	Failed	Failed
Attacker	Attacker	Attacker	Succeeded	Succeeded
Attacker	Helsenorge	Helsenorge	Failed	Failed
Attacker	Attacker	Helsenorge	Failed	Failed
Attacker	Helsenorge	Attacker	Failed	Failed

A message was encrypted with the Helsenorge encryption key. Decryption was attempted with the attacker decryption key, but failed. Decryption of the message was then attempted with the Helsenorge decryption key, but an incorrect password. This caused the decryption to fail. In order to account for false negatives, a message was encrypted and decrypted with the attacker encryption and decryption keys, which was successful.



```
-----BEGIN PGP MESSAGE-----
Version: BCPG v@RELEASE_NAME@

hQEMA1YuTLt28lTEAQgAtlRcZGUoAMfE83eUlkXgGUf+oPcUfN1OstzWuMVCvF72
j9ekwSywPWHBTDNTW+SVQbnzshDGjwOUWQ97vhavx1yTvXNtmG2cTUszKPyDQ79d
p/GXkleNTSz6T4XAD+GtVY4cHPHKhIcNXcBIcyJzCAsUp6/HItopttHVbmdpdhQb
CVGXD+1nni4dKSD3XgroWF1pBAN5KcE6CZIHp/tsiqNN4YfaXYDr1Wv6Y0jRSdRM
QMK7J5nzMGOgcFeDCpfoyYCkpIcixI49LHmAcZNIJURPXIutfGbCzJrlZEcSb/9s
zt0ktlrVD1qw2qV0Lhvk8Fkn824chPVQumkPUcutDdJBAanQZFAA5+StEIquJAFE
p8esKs8R4eokXxv1kt7tmaGlPbHIpsUFEMkr+RA8UTo6mDhvB0KWbKsBS79c4gHC
Uow=
=WXiE
-----END PGP MESSAGE-----
```

```
-----BEGIN PGP MESSAGE-----
Version: BCPG v@RELEASE_NAME@

hQEMA1YuTLt28lTEAQf+PfqE75s/1TG9rhgcGTPBSlreFAk99QKZ/eJ1mxHTVL4C
Y1UQ3OUQpM8oeTcOUMZ30HKhTO4IxBuOB1wi0KhTN47K9maqR0sQ60vOV+BzCY/N
Ezs2a62LTV9LTcLq408UpIqZ+asiNFQa1KAIXrT+uaPXuhJis7wywN2fIFz5s0u7
WHGS/SJ6/t7rtzdnONHPksfUPJG/cgOhYW+ha2AR8HG2YBD2MFlbkW/g3O5iNC0m
VHG+Bpjucq2+3X7etKBIDseIMkoy/wB7wTPSjgof0Hx48aX1Sg2O7MpWPKZDtAuN
KxBUecpGHV2x3BPYT/NRmhD7eTWfRajRoGcrZHdRXdJBAY1Xg8o2kCb7i+qjzxrM
TM3h7uFfkwnM/JIjN0vx5ZxvYj2vTtkmftCZ82u6F2MqK+EGL4A4H3DVR73uvH7Z
9dM=
=F5Ii
-----END PGP MESSAGE-----
```

Figure 5.9: Same message encrypted twice

Even though the OpenPGP protocol itself has not been broken, there may be bad implementations. A few tests have been done to make sure that basic properties of secure encryption algorithms are included. The plaintext "hehe" was encrypted twice and produced two almost completely different ciphertexts. Both share the string "hQEMA1YuTLt28lTEAQ" at the start, but this is just a header that contains public information such as algorithms used for encryption and decryption [24].

Chapter 6

Conclusion

We have shown how to use various technologies in order to build a secure cross-platform mobile application for sharing self-collected health data in a clinical context. React-Native lacks viable cross-platform libraries for OpenPGP and a secure database that works well with health data in the FHIR format. The framework's support for native modules was therefore an important factor. The main reason for choosing a cross-platform framework was the ability to have a single shared code base, which saves development time. Some of this benefit was lost because of the native components. Xamarin Forms can be considered if one wants to avoid writing any native code, but we have not proven that it will support the three scenarios. Because React-Native uses JavaScript, a dynamically typed language, it can handle FHIR data in the JSON format without any issues. Couchbase Lite is the only database that fulfills all the requirements out of the box on interpreted and cross-complied applications. However, its design makes it inefficient when retrieving semi-structured data.

The application preserves confidentiality (T1) and integrity (T2) by using Couchbase Lite for persisting encrypted patient data on the device (S1). Android devices must use version 6 or higher in order to secure the database password in Keystore. A combination of OpenPGP and HTTPS secures data sent over the internet (S3), and ID-Porten is used for authentication (S2) in order to identify patients. Penetration tests were executed, and one vulnerability was found. A revoked certificate can threaten S3 given that protection provided by OpenPGP is bypassed. It is possible that S2 is affected as well.

In order to integrate the app with the Norwegian infrastructure, we use FHIR as the data format for exchanging health data (S3). EHRs and other systems, such as those operated by Helsenorge and FullFlow will be interoperable as they will use the same format. For the sake of supporting the storage of data (S1) in the FHIR format, we use a schema-less NoSQL database that is suitable for storing semi-structured data. By authenticating with ID-Porten (S2) and having it manage the user accounts of both our app and Helsenorge, we can ensure that self-collected health data will be stored in the correct kjernejournal.

In order to ensure acceptable performance on both low and mid-end devices, the app should remind patients to send data at least once a week on low-end devices, and one every two weeks on mid-end devices. If a patient sends data more rarely than this, the application can be slow to respond, which will cause the user experience to diminish. If a patient imports blood glucose data in bulk rather than continuously, this must be done at least once every two days on a low-end device, or once every five days on a mid-end device to ensure responsiveness. For the sake of usability, we do not require the patient to enter a password in order to access the database when opening the app. We rely on The Keychain and Keystore for encrypting a generated, cryptographically-secure random database password when the app is installed.

To summarize, there are issues with some of the technologies, but they are still suitable for the development of a secure cross-platform mobile application for managing and sharing health data with healthcare systems. There is some uncertainty due to an issue with revoked SSL certificates. If this can be dealt with, the goals of all three scenarios are achievable. The scenario tests, with one exception, indicate that the application preserves the confidentiality and integrity of patient data. The technologies facilitate integration with the Norwegian healthcare infrastructure. Performance tests show how often health data should be imported and shared in order to ensure usability.

Chapter 7

Further Work

Security mechanisms and modules for data sharing are in place, but there is no business logic that imports health data or decides which data should be sent. For the app to be usable by patients, these things have to be implemented. The primary goal for further work should be to find suitable ways of importing data. On low-end devices, the application can only store two days' worth of blood glucose measurements at a time before the response time becomes too slow. Because of this, integration with continuous glucose monitors which automatically import small amounts of data at a time without user interaction would be ideal. The security of such a solution should be assessed. Users should also be able to manage their self-collected health data after it has been imported. In addition, it would be beneficial to have the option of importing health data from Helsenorge so that data sharing goes both ways.

The application should have a proper privacy policy that gives the patient information about what his health data is used for, and who has access to it. The app must also get explicit consent from users when needed.

It is also possible to reduce the complexity of key verification. Instead of having to make both a detached and embedded signature, one can attempt to implement detached signature verification on Android. Alternatively, one can try to find a C/C++ library that supports embedded signature verification, and use it for iOS. This would make it easier to maintain the Helsenorge server.

More thorough security testing should be done. One could for example get an SSL certificate signed by a CA in order to test with HTTPS on the Helsenorge server. Additional tests can also be done if issues related to using plain HTTP with IdentityServer 4 are resolved. The issue with revoked SSL certificates should be investigated. Even more comprehensive testing can be done if one has access to rooted or jailbroken devices.

More research should be done on Android's Keystore in order to verify that issues in old versions have been fixed. Lastly, one should look into how to implement artificial delay on responses from the Helsenorge server in order to defend against OpenPGP timing attacks.

Bibliography

[1] Mobile health apps available in 2017. https://research2guidance.com/325000-mobile-health-apps-available-in-2017/, 2017.

[2] Fhir extension. http://www.hl7.org/FHIR/extensibility.html, (accessed April 12, 2019).

[3] Fhir observation. http://hl7.org/fhir/STU3/observation.html, (accessed April 12, 2019).

[4] Item attribute keys and values. https://developer.apple.com/documentation/security/keychain_services/keychain_items/item_attribute_keys_and_values, (accessed April 27, 2019).

[5] ksecattraccessiblewhenunlockedthisdeviceonly. https://developer.apple.com/documentation/security/ksecattraccessiblewhenunlockedthisdeviceonly, (accessed April 27, 2019).

[6] Releasing react-native 0.56. https://facebook.github.io/react-native/blog/2018/07/04/releasing-react-native-056, (accessed April 27, 2019).

[7] Norwegian Centre for E-health Research. https://ehealthresearch.no/en/projects/fullflow, (accessed February 16, 2018).

[8] How to compile and use see. https://www.sqlite.org/see/doc/trunk/www/readme.wiki, (accessed February 19, 2019).

[9] Sqlitedatafile.cc. https://github.com/couchbase/couchbase-lite-core/blob/89516d2a483ffc091e0fa758404ad8e490f7e249/LiteCore/Storage/SQLiteDataFile.cc, (accessed February 19, 2019).

[10] Application sandbox. https://source.android.com/security/app-sandbox, (accessed February 2, 2019).

[11] Communication between native and react native. https://facebook.github.io/react-native/docs/communication-ios, (accessed February 2, 2019).

[12] File system basics. https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html, (accessed February 2, 2019).

[13] Bouncy castle. https://www.bouncycastle.org/, (accessed February 21, 2019).

[14] Objective pgp. https://github.com/krzyzanowskim/ObjectivePGP, (accessed February 21, 2019).

[15] Open keychain api. https://github.com/open-keychain/openpgp-api, (accessed February 21, 2019).

[16] React-native performance. https://facebook.github.io/react-native/docs/performance, (accessed February 21, 2019).

[17] Spongy castle. https://github.com/rtyley/spongycastle, (accessed February 21, 2019).

[18] Swift-pgp. https://github.com/kryptco/swift-pgp/, (accessed February 21, 2019).

[19] react-native-oidc. https://www.npmjs.com/package/@ist-group/react-native-oidc, (accessed February 22, 2019).

[20] react-native-oidc-client. https://www.npmjs.com/package/react-native-oidc-client, (accessed February 22, 2019).

[21] Android keystore system. https://developer.android.com/training/articles/keystore, (accessed February January 9, 2018).

[22] Ios security. https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf, (accessed February January 9, 2018).

[23] The sqlite bytecode engine. https://www.sqlite.org/opcode.html, (accessed March 11, 2019).

[24] Rfc-4880. https://tools.ietf.org/html/rfc4880#section-5.14, (accessed March 3, 2019).

[25] React-native-openpgp. https://github.com/orhan/react-native-openpgp, (accessed March 7, 2019).

[26] En av verdens første fhir-implementasjoner. https://www.dips.com/no/en-av-verdens-forste-fhir-implementasjoner, (accessed May 1, 2019).

[27] Kjernejournal for safer healthcare. https://helsenorge.no/kjernejournal/kjernejournal-for-safer-healthcare, (accessed May 1, 2019).

[28] Certified oidc implementations. https://openid.net/developers/certified/, (accessed May 19, 2019).

[29] Recommendation for using hl7 fhir for data sharing. https://ehelse.no/standarder-kodeverk-og-referansekatalog/standarder-og-referansekatalog/recommendation-for-using-hl7-fhir-for-data-sharin (accessed May 19, 2019).

[30] Openid connect core 1.0. https://openid.net/specs/openid-connect-core-1_0.html, (accessed May 23, 2019).

[31] Saml v2.0 enhanced client or proxy profile version 2.0. http://docs.oasis-open.org/security/saml/Post2.0/saml-ecp/v2.0/csprd01/saml-ecp-v2.0-csprd01.html, (accessed May 23, 2019).

[32] Android benchmarks. https://browser.geekbench.com/android-benchmarks, (accessed May 27, 2019).

[33] ios benchmarks. https://browser.geekbench.com/ios-benchmarks, (accessed May 27, 2019).

[34] iphone 6 benchmark. https://browser.geekbench.com/ios_devices/38, (accessed May 27, 2019).

[35] Lg k10 (2017) benchmark. https://browser.geekbench.com/android_devices/495, (accessed May 27, 2019).

[36] Android build numbers. https://source.android.com/setup/start/build-numbers, (accessed May 29, 2019).

[37] Android keystore. https://source.android.com/security/keystore, (accessed May 29, 2019).

[38] Android version usage statistics. https://developer.android.com/about/dashboards, (accessed May 29, 2019).

[39] React-native keychain. https://github.com/oblador/react-native-keychain, (accessed May 29, 2019).

[40] React-native secure-random. https://www.npmjs.com/package/react-native-securerandom, (accessed May 29, 2019).

[41] Rn secure-storage. https://github.com/talut/rn-secure-storage, (accessed May 29, 2019).

[42] Bad ssl. https://badssl.com/, (accessed May 3, 2019).

[43] Evaluation of certificates revocation (crl/ocsp). https://forums.developer.apple.com/thread/24298, (accessed May 3, 2019).

[44] More on android and revoked ssl certificates. https://commonsware.com/blog/2014/04/18/more-android-revoked-ssl-certificates.html, (accessed May 3, 2019).

[45] Revokedgrc. https://revoked.grc.com/, (accessed May 3, 2019).

[46] Most popular cross-platform mobile frameworks. https://stackshare.io/cross-platform-mobile-development, (accessed May 31, 2019).

[47] Normen. https://ehelse.no/normen/normen-for-informasjonssikkerhet-og-personvern-i (accessed May 31, 2019).

[48] Stack overflow survey 2019. https://insights.stackoverflow.com/survey/2019, (accessed May 31, 2019).

[49] Infodoc. https://www.infodoc.no/produkter/, (accessed May 5, 2019).

[50] Owasp top 10 mobile. https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10, (accessed May 7, 2019).

[51] openpgp. https://www.openpgp.org/, (accessed May 8, 2019).

[52] Pgp. https://en.wikipedia.org/wiki/Pretty_Good_Privacy, (accessed May 8, 2019).

[53] C. P. Rahul Raj and. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *2012 Annual IEEE India Conference (INDICON)*, pages 625–629, Dec 2012.

[54] Cosimo Anglano. Forensic analysis of whatsapp messenger on android smartphones. *Digital Investigation*, 11, 05 2014.

[55] A Begoyan. An overview of interoperability standards for electronic health records. 04 2019.

[56] J. Bellamy-McIntyre, C. Luterroth, and G. Weber. Openid and the enterprise: A model-based analysis of single sign-on authentication. In *2011 IEEE 15th International Enterprise Distributed Object Computing Conference*, pages 129–138, Aug 2011.

[57] D. Bender and K. Sartipi. Hl7 fhir: An agile and restful approach to healthcare information exchange. In *Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems*, pages 326–331, June 2013.

[58] Kevin Bocek. Is https enough to protect governments? *Network Security*, 2015(9):5 – 8, 2015.

[59] C. Braghin, S. Cimato, and A. Della Libera. Are mhealth apps secure? a case study. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 335–340, July 2018.

[60] P. Brooks and B. Hestnes. User measures of quality of experience: why being objective and quantitative is important. *IEEE Network*, 24(2):8–13, March 2010.

[61] M. Cagnazzo, M. Hertlein, T. Holz, and N. Pohlmann. Threat modeling for mobile health systems. In *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, pages 314–319, April 2018.

[62] Zhaoquan Cai, Hongyang Yan, Ping Li, Zheng-an Huang, and Chongzhi Gao. Towards secure and flexible ehr sharing in mobile health cloud under static assumptions. *Cluster Computing*, 20(3):2415–2422, Sep 2017.

[63] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices*, SPSM '14, pages 11–20, New York, NY, USA, 2014. ACM.

[64] Luis Corral, Alberto Sillitti, and Giancarlo Succi. Mobile multiplatform development: An experiment for performance analysis. *Procedia Computer Science*, 10:736 – 743, 2012. ANT 2012 and MobiWIS 2012.

[65] Abhijit Das, Chiara †, Chiara Galdi, Hu Han, R Raghavendra, Jean-Luc Dugelay, and Antitza Dantcheva. Recent advances in biometric technology for mobile devices. 09 2018.

[66] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[67] Jon Friedman and Daniel V. Hoffman. Protecting data on mobile devices: A taxonomy of security threats to mobile computing and review of applicable defenses. *Inf. Knowl. Syst. Manag.*, 7(1,2):159–180, April 2008.

[68] Samson Hussein Gejibo. *Towards a Secure Framework for mHealth. A Case Study in Mobile Data Collection Systems.* PhD thesis, University of Bergen, 2015.

[69] M. Ghazal, Y. Al Khalil, F. Haneefa, A. Mhanna, D. Awachi, and S. Ali. Towards secure mobile process tracking using wearable computing in mgovernment applications. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pages 288–294, Aug 2016.

[70] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 162–168, New York, NY, USA, 2017. ACM.

[71] M. Halpern, Y. Zhu, and V. J. Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–76, March 2016.

[72] Mark Harris, Karen Patten, and Elizabeth Regan. The need for byod mobile device security awareness and training. volume 5, pages 3441–3451, 01 2013.

[73] Jens Heider and Matthias Boll. Lost iphone? lost passwords! practical consideration of ios device encryption security. 2011.

[74] Jens Heider and Rachid El Khayari. Keychain weakness faq further information on ios password protection. 2012.

[75] Victoria Hordern. Data protection compliance in the age of digital health. *European Journal of Health Law*, 23:248–264, 06 2016.

[76] A. K. Jain and D. Shanbhag. Addressing security and privacy risks in mobile applications. *IT Professional*, 14(5):28–33, Sep. 2012.

[77] Bill Karwin. *SQL Antipatterns: Avoiding the Pitfalls of Database Programming.* Pragmatic Bookshelf, 1st edition, 2010.

[78] Abdul Nasir Khan, M.L. Mat Kiah, Samee U. Khan, and Sajjad A. Madani. Towards secure mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(5):1278 – 1299, 2013. Special section: Hybrid Cloud Computing.

[79] K. Knorr and D. Aspinall. Security testing for android mhealth apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–8, April 2015.

[80] Rajiv B. Kumar, Nira D. Goren, David E. Stark, Dennis Paul Wall, and Christopher Longhurst. Automated integration of continuous glucose monitor data in the electronic health record using consumer technology. In *JAMIA*, 2016.

[81] Wanpeng Li and Chris J. Mitchell. Analysing the security of google's implementation of openid connect. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 357–376, Berlin, Heidelberg, 2016. Springer-Verlag.

[82] Yi Liu, Yinghui Zhang, Jie Ling, and Zhusong Liu. Secure and fine-grained access control on e-healthcare records in mobile cloud computing. *Future Generation Computer Systems*, 78:1020 – 1026, 2018.

[83] Hans Löhr, Ahmad-Reza Sadeghi, and Marcel Winandy. Securing the e-health cloud. pages 220–229, 01 2010.

[84] Serge Mister and Robert Zuccherato. An attack on cfb mode encryption as used by openpgp. In *Proceedings of the 12th International Conference on Selected Areas in Cryptography*, SAC'05, pages 82–94, Berlin, Heidelberg, 2006. Springer-Verlag.

[85] A B M Moniruzzaman and Syed Hossain. Nosql database: New era of databases for big data analytics - classification, characteristics and comparison. *Int J Database Theor Appl*, 6, 06 2013.

[86] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004.

[87] N. Naik and P. Jenkins. An analysis of open standard identity protocols in cloud computing security paradigm. In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 428–431, Aug 2016.

[88] A Nayak, A Poriya, and Dikshay Poojary. Article: Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5:16–19, 01 2013.

[89] Jakob Nielsen. Usability engineering. In *The Computer Science and Engineering Handbook*, pages 1440–1460. 1997.

[90] Anders Steen Nilsen. Integrating FHIR in Infodoc's Plenario: How to run untrusted JavaScript safely. Master's thesis, 2019.

[91] Patrick J. O'Connor, JoAnn M. Sperl-Hillen, William A. Rush, Paul E. Johnson, Gerald Amundson, Stephen E. Asche, Heidi L. Ekstrom, and Todd Gilmer. Impact of electronic health record clinical decision support on diabetes care: a randomized trial. *Annals of family medicine*, 9 1:12–21, 2011.

[92] A. Papageorgiou, M. Strigkos, E. Politou, E. Alepis, A. Solanas, and C. Patsakis. Security and privacy analysis of mobile health applications: The alarming state of practice. *IEEE Access*, 6:9390–9403, 2018.

[93] J. Payne. Secure mobile application development. *IT Professional*, 15(03):6–9, may 2013.

[94] Mor Peleg, Yuval Shahar, and Silvana Quaglini. Making healthcare more accessible, better, faster, and cheaper: the mobiguide project. *Eur. J. ePractice: Issue Mobile eHealth*, 20:5–20, 01 2014.

[95] Jaroslav Pokorny. Nosql databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82, 2013.

[96] M. Reza Rahimi, Jian Ren, Chi Harold Liu, Athanasios V. Vasilakos, and Nalini Venkatasubramanian. Mobile cloud computing: A survey, state of art and future directions. *Mobile Networks and Applications*, 19(2):133–143, Apr 2014.

[97] Mohamed Sabt and Jacques Traoré. Breaking into the keystore: A practical forgery attack against android keystore. *IACR Cryptology ePrint Archive*, 2016:677, 2016.

[98] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing.

[99] Z. Tu and Y. Yuan. Understanding user's behaviors in coping with security threat of mobile devices loss and theft. In *2012 45th Hawaii International Conference on System Sciences*, pages 1393–1402, Jan 2012.