Towards a Secure Framework for mHealth

A Case Study in Mobile Data Collection Systems

Samson Hussien Gejibo

Dissertation for the degree of philosophiae doctor (PhD)

at the University of Bergen

2015

Dissertation date: 5 November

To my wife, sons, and future generation of my family to come.

Scientific environment

The research presented in this dissertation has been conducted in the Programming Theory group of the Department of Informatics at the University of Bergen. The candidate collaborated with the open-source mobile data collection projects, openXdata and Open Data Kit (ODK).

University of Bergen

Department of Informatics

UNIVERSITAS BERGENSIS

OPENXDATA

OPEN DATA KIT

Acknowledgements

First and foremost I would like to thank God, the Almighty for having made everything possible by giving me strength and courage to do this work. I would like to express my deepest gratitude to my supervisor, Khalid A. Mughal, for all the support from the Ph.D. program enrollment until the completion, his excellent guidance, caring, patience, and for introducing me to my outstanding co-supervisor, Federico Mancini. I am grateful to Federico for his dedication, encouragement, and instant feedback. Khalid and Federico deserve much of the credit for this thesis, and I am indebted to them for the help and inspiration.

I would like to give a heartfelt, special thanks to Jørn Klungsøyr, for providing me the opportunity to pursue the Ph.D. program and introducing me to my supervisor, Khalid. He has been motivating, encouraging, and enlightening. He provided insight and direction right up to the end. For this, I cannot thank him enough. I am grateful to Daniele Grasso, Remi André B. Valvik, and André Kristensen, for being part of our security team and great contributions.

Gaetano Borriello, may his soul rest in peace. I express my deepest gratitude to him and his Open Data Kit core team members especially Mitch Sundt, Waylon Brunette, and Sam Sudar, for their support and discussions during my three months stay at the University of Washington, Seattle. I must say a special thank you to Martin Were, Ada Yeung, Nyoman Ribeka, and members of AMPATH-Kenya, for their support and guidance during my field visit in Eldoret Kenya.

I would like to thank my opponents Bruce MacLeod and Patricia Mechael for agreeing to serve on my dissertation committee, and Uwe Wolter, for coordinating the committee.

A very special thanks goes to my best friends Daniel Huluka and Befkadu Assefa for their genuine caring, encouragement, and the proofreading. I am also thankful to Fisseha Mekuria for the support and encouragement whenever I was in need.

I would like to thank Kjell Jørgen Hole for his generosity for supporting my three months stay at the University of Washington. I am very thankful to the Department of Informatics, the University for supporting my study. I must thank all group members of the Programming Theory for providing me extra funding for my visit to Kenya. I am also grateful to the administration staffs of the Department of Informatics.

Most importantly, none of this would have been possible without the love and patience of my beautiful wife Yodit and our two lovely children, Nathan and Lucas. I am very thankful to my wife for her love, constant support all these years.The days and nights away from them while writing this dissertation were truly difficult.

Finally, I would like to thank the COINS Research School of Computer and Information Security, for the funding support to participate in workshops. A special thanks also goes to the communities behind openXdata, Open Data Kit, and mUzima.

Abstract

The rapid growth in the mobile communications technology and wide cellular coverage created an opportunity to satisfy the demand for low-cost health care solutions. Mobile Health (a.k.a. mHealth) is a promising health service delivery concept that utilizes mobile communications technology to bridge the gap between remotely and sparsely populated communities and health care providers. So far, several mHealth applications have been developed and deployed in the field. Among those, a digital information gathering and dissemination system using mobile devices is the main focus of this work. This type of mHealth system is called Mobile Data Collection System (MDCS). Although MDCS succeeds over traditional paper form based data collection; it has also brought unique challenges such as data security in mobile communications technology. Despite MDCS are often used to collect sensitive health-related data, more work was needed to address security issues like confidentiality, integrity, availability and authentication to secure sensitive health related information in storage, data exchange and processing.

When we began this work, Java ME enabled feature phones, that dominated the scene for a decade, were the choice of most MDCS. At that time, in collaboration with our partner project, we proposed a secure custom protocol. The protocol has been implemented, tested, and integrated into our reference MDCS. We have confirmed the flexibility of our secure solution by retrofitting the existing openXdata system with user authentication, secure storage and communication solutions by modifying only a few lines of code in the client-server application.

However, in the past few years, the explosion of new mobile platforms and cloud-based services became game changer in our work. The move from feature phones to smartphones brought to the table the need to reevaluate, redesign, and port our earlier secure solution to smartphones based MDCS by considering the unique features and challenges of both smart phone clients and cloud-based server-side deployments.

In this dissertation, we analyze the challenges in securing mobile data collection systems deployed in remote areas, in resources-constrained environment, and in low project budget settings. We present a flexible and secure framework that offers user authentication both online and off-line, secure mobile storage, secure communication, and secure cloud storage. Besides, the framework provides data integrity, user account and data recovery, and multi-user management and is designed to be easily integrated in existing MDCS with minimal effort. Although fundamental security issues are conceptually identical in both old feature phone and current smartphone based solutions, our framework and the proposed solutions address the unique aspects of both mobile platforms. We also discuss the solution we designed for older Java ME based devices, and how they are still relevant. For this work, we collaborated with the open-source MDCS, openXdata and Open Data Kit (ODK).

Contents

List of Figures

List of Tables

Code Listings

1 Introduction

In the last decade, mobile technology have presented an alternative way of information dissemination and gathering. This potential has attracted individuals and organizations including researchers, governments, non-governmental organization (NGOs), donors, and standard and regulatory bodies. Mobile Health (mHealth) one of the innovations that emerged from these group collaboration efforts. mHealth try to leverage mobile communications technology to tackle health care challenges in developing as well as developed countries. However, mHealth also brought a range of unique challenges. One of the challenges is information security, which in the case of mobile technology is taken to a whole new level compared to traditional computers based electronic medical record systems.

This chapter introduces the mHealth system and its ecosystem and security challenges and briefly discuss the candidate's motivation, and the scope of the research. In the next section, we discuss Electronic Health (a.k.a eHealth) and mHealth.

1.1 eHealth and mHealth

According to [34, 39, 133], eHealth is defined as the use of information and communication technology (ICT) to improve health and health-care systems. Moreover, ICT is defined as a term used to encompass all technologies including telecommunications (both wired and wireless), Internet, computers, software, and storage media which enable users to create, process, store, access, transmit, and manipulate information[1]. The term eHealth has emerged together with eBanking (electronic banking), eCommerce (electronic commerce), and eFinance (electronic finance). These all together reshaped the service delivery landscape around the globe. eHealth systems are designed and developed using traditional server and desktop based applications, deployed on computers, and accessed through Internet infrastructure and other similar technologies. Several standard and regulatory bodies supported the eHealth to grow by building trusted rela-

[1]What is ICT: `https://ci.uky.edu/lis/ict/whatisict`

tionship between service providers and consumers through standards and regulations. Some of such standards are: the Health Insurance Portability and Accountability Act (HIPAA) (first outlined in 1996), Federal Trade Commission Act [133], FDA guidance for computerized Systems used in Clinical Trials[2], Omnibus data protection in European Union and Australia, and others [18, 133]. Since the advancement of mobile communications technology, services that use Internet infrastructure with the support of computers are being customized to the mobile environment. Services such as mBanking (mobile banking), mCommerce (mobile commerce), mFinance (mobile finance), and mHealth (mobile health) have emerged to advance the traditional services using mobile technology. Financial systems such as mBanking and mCommerce have already gained an enormous success. The health care sector is also trying to tackle outstanding challenges using mobile technology.

Thus, the broader definition given to eHealth makes mHealth as an emerging part of the eHealth infrastructure. The American National Institutes of Health defined mHealth as the delivery of health-care services via mobile communication devices [3], and while the idea of using mobile devices to provide health care related services is a very convenient prospect in industrialized countries, it is having much greater impact on the health care system of low-income countries. In fact, health care in these nations can be scarce or difficult to access due to restraints such as limited resources, finances and health-care workforce, and because of parts of the population living in remote locations. High mobile phone penetration makes mHealth a viable option for providing better health care[4] [5] through mHealth systems [81, 134].

1.2 mHealth in developing countries

mHealth intends to provide basic health services to unserved and underserved communities in low-income countries. Even though mHealth began in the context of low-income countries, industrialized countries have also shown interest [37] to use mHealth as a tool reduce expenses and offer personalized health care through remote monitoring and diagnostic tools. A typical example is the use of biosensors for the continuous remote monitoring of chronic diseases [103, 106]. The reason why these sensor technology is not yet widely deployed also in low-income countries is the rare availability of low-cost sensors. What is instead available are exceptionally low cost diagnostic tools like Rapid diagnostic test (RDT) toolkit for HIV and malaria, which costs less than a quarter of a dollar per a single use [6], and although they have nothing to do with mobile technology, some mHealth tools can still be used to efficiently collect and transmit the results of such tests.

However, mobile phone manufacturers have understood the potential impact of sen-

[2]FDA, Guidance for Industry, Computerized Systems used in Clinical Trials, `http://www.fda.gov/OHRMS/DOCKETS/98fr/04d-0440-gdl0002.pdf`

[3]mHealth Definition: `http://caroltorgan.com/mhealth-summit/`

[4]mHealth Services: `http://www.grameenfoundation.org/blog/mobile-technology-provides-lifesaving-tool-race-ebola-vaccines`

[5]mHealth Service: `http://www.uib.no/en/cih/72461/using-mobile-phones-track-immunizations`

[6]Rapid Diagnostic Tool, [Last Accessed: January 2015], `http://www.alibaba.com/showroom/malaria-rapid-diagnostic-test-kit.html`

sors and are embedding different types of sensors in the smart-phone platforms. Some sensors already appearing in mobile platforms are accelerometers, digital compass, proximity sensor, gyroscope, GPS, microphone, and camera [68, 81]. Based on the growing research and development in this area, there is strong indication that biosensors will also be an integral part of ordinary cell phone and smartphone platforms [68] [7] or new wearable gadgets [8] in the near future. Sensors embedded in mobile phones can in addition benefit from better computing power, speed and memory as compared to standalone sensors. Still, there are some obstacles to this integration. For instance, the level of expertise required to develop a mobile sensing application, security and privacy policies, and possibly low bandwidth. Besides, if data from the sensor is also to be processed on the device in order to meet the quality standards that are expected by medical or other instrumental records [9] [10], then even a smartphone might not have high enough computing power for this kind of computation. So, assuming that the technical challenges will be solved and biosensors are going to be part of commercial mobile devices, that of developing a mHealth application based on them will have to be a participatory design process where all stakeholders (for instance: policy makers, users, service providers, product vendors) are involved.

As of this writing, not much research has been published on how to solve some of the listed technological challenges, but some activities exist. Regarding security in resource constrained environment with low budgets, the candidate work including this thesis and a previous publication in collaboration with the Council for Scientific and Industrial Research in South Africa (CSIR) [47] and the SecourHealth framework [76] are some of the few existing results. When it comes to the problem of sensor technology and data processing, there are promising research activities at the CSIR, while the Open Data Kit (ODK) community aims to minimize the technical expertise needed to develop mobile sensing application through a mobile sensor framework [15]. Still, a great deal of work is needed to achieve a real and usable integrated mobile platform for biosensors.

1.3 Mobile Data Collection (MDC)

As a result of the situation described in the previous section, mHealth systems in low-income countries have not yet considered the use of sensors technology as an automated way of gathering medical data from individuals in the field. Instead, the systems leverage human operators that input collected data into mobile devices manually. Hence, in this dissertation, we consider mHealth systems where data is entered by human collectors (aka data collectors), and we are not considering the situation where a sensor is feeding data to a remote server or device, or that a monitoring program is installed on the mobile device and its correct operation must be guaranteed at all times as in considered by other research [124]. The collected data is also not thought to be readily accessible to health-care providers, or in general to be offered as a service. Data is

[7]Sensors for Health Care, [Last Accessed: April 2015], http://www.sensorsmag.com/specialty-markets/medical-devices/sensors-help-advance-health-care-856

[8]iWatch, [Last Accesed: April 2015], https://www.apple.com/watch/

[9]http://www.hhs.gov/ocr/privacy/

[10]http://www.fda.gov/

rather stored in a central repository which is not necessarily connected to any medical system that health operators can access. In general we are not concerned with medical record systems with patient journal that need to be update, retrieved and protected from unauthorized disclosure either, although similar problems may be relevant for some of our scenarios.

The particular area of mHealth that deals specifically with data collection as we described it here, is identified in a report by the United Nations Foundation and Vodafone Foundation [134] and it is called **Mobile Remote Data Collection** or MDC from here on. Nevertheless, almost all specialized mHealth applications incorporate data collection processes directly or indirectly. Therefore, the work presented here is applicable also in any context where secure framework is required to protect data at rest and in transit, although initially developed specifically for Mobile Data Collection Systems. In the next section, we discuss the computer security which eHealth security standards and regulations is built on and present unique challenges in mHealth work space that requires independent security analysis and solution.

1.4 Security in health related systems

The basic security principles one shall ensure in information systems are confidentiality, integrity, and availability of data (CIA) in any possible circumstances. FIPS PUB 199 [40] defines these security objectives as follows:

Confidentiality: "Preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information. . . "

Integrity: "Guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity. . . "

Availability: "Ensuring timely and reliable access to and use of information. . . "

When considering health related data and systems we shall also add information privacy and security considerations to the above principles what is presented in in [57, 133]:

“*Information Privacy: an individual’s right to control the acquisition, use, and disclosure of identifiable health data*”

> *Information Security: physical, technological, or administrative safeguards or tools used to protect identifiable health data from unwarranted access or disclosure, including security of wireless networks, security of devices, applications security, back-end systems security, and secure user practices.*

In order to achieve such objectives, security safeguards should be in place so that all aspects of a system are covered. The HIPAA [57] standard for protection of electronic health information (ePHI) identifies three types of safeguards:

- **Administrative Safeguards** is defined as the *"administrative actions and policies, and procedures to manage the selection, development, implementation, and maintenance of security measures to protect electronic protected health information and to manage the conduct of the covered entity’s workforce in relation to the protection of that information."*

- **Physical Safeguards** is defined as the *"physical measures, policies, and procedures to protect a covered entity's electronic information systems and related buildings and equipment, from natural and environmental hazards, and unauthorized intrusion."*
- **Technical Safeguards** is defined as the *"the technology and the policy and procedures for its use that protect electronic protected health information and control access to it."*

Technology in particular is also mentioned as the first line of defense for protecting data that are labeled private and sensitive in [133], and this is also what we focus on in this thesis.
An in depth analysis of information privacy when using mobile technology in the health-care system is presented in [8, 133]. We recommend these documents to readers interested in subject related to information privacy. The work presented in the dissertation is only for Information security.

1.4.1 How mHealth Security differs from eHealth Security?

As the focus of this thesis is mHealth security, a central question becomes: In what sense is research on the security of mobile devices different from common security research? Is it possible to transfer known security solutions from ordinary desktop computers to mobile devices? Could it possibly be the same, only with the additional word "mobile" in the title?

In particular "Is it possible to transfer known eHealth security standards and solutions to mHealth systems? If not, what makes mHealth security needs different from that of eHealth systems?".

Computing devices differ in their design, architecture, form factor, hardware specification and main usage purpose. A personal computer and today's smartphones have one thing in common, both can be used for computing, but the two are completely different as per the above mentioned parameters. When one think of designing or developing a solution, it's important to consider such differences among the target platform even if the purpose is the same. Browsers, games, tools like calculator and planner, media players and the likes work in a varieties of computing devices and the seamless functioning of such solutions depends on their design consideration of the nature of the target platform. Security solutions are not an exception. Becher, M. et al. [11] identifies the critical distinction between mobile devices and other computing devices in the design and implementation of security solutions. Although one can identify quite many specialities of mobile devices, the attack vectors are one way or another related to:

1. **The communication network model**: the communication between mobile devices and mobile network operators is very strong and trust based. The mobile network operator has strong influence on the device which is different from the communication model between computers and network providers, in which the network operator has almost no influence on the computers in the network. Even though the trust model is helpful for usage and development of applications, a weakness or vulnerability in the communication can open a door for serious security breaches.

2. **Strong attachment between the owner and the device**: mobile devices are very personal and they are strongly connected to the owner. Which means they have more power to perform some actions on behalf of their owners, which is usually unlikely in computers. A ping from a computer and a call attempt from a mobile device are both connection attempts but are not equally interpreted.

3. **Gains from attacks**: with attacks on different computing devices attackers aim at gaining access to the device's resource or sensitive data stored on it. In mobile devices in addition to this two, it's also possible to make financial gains by controlling the device for instance sending rogue SMS or calling to some numbers in which the target made to pay for a service that he/she hasn't initiated. As mobile devices are usually on the hands of individuals, knowing the location of the device by itself may be interesting enough for an attacker and a privacy breach for the target.

4. **Usage environment**: mobile devices are used in public places and the attack vectors can also extend to getting sensitive information through eavesdropping. Even if eavesdropping is used to get sensitive information from computer users, the public usage of mobile devices makes them more susceptible to that.

Mobile devices specialties do not only determine the attack vectors, but also the type of security solutions to block such attacks. Becher, M. et al. [11] also identifies device resources as the most obvious differences from computers when designing and implementing security solutions. The main limiting factors are CPU, memory, and battery. Sometimes these type of resource limitations are mitigated with transferring computation load from the mobile device to the mobile network. But that also requires usage of wireless network which uses more battery power, one of the limited resources in mobile devices. This leads to trade-off between which limited resources one can sacrifice on mobile devices.

In summary, mobile devices present some unique security challenges that have not been addressed in traditional computer security. Mobile devices are small, light and can be easily compromised. The mobility nature of the mobile devices does not allow us to provide physical security protection at the same level as the traditional computers at the data center or workstations. Mobile devices are becoming part of organizations ICT infrastructure and they are more and more connected to critical systems to access services and resources. Therefore, security researchers, platform providers, phone manufactures, corporate security teams, and standard bodies have recognized the challenges and have been searching for reasonable security solutions to mitigate the risks associated with the use of mobile devices for creating, storing, accessing, transmitting, and processing sensitive information. One more important difference to be aware about, is also the settings in which mHealth systems are deployed, since they can greatly influence the type of potential threats and define specific design constraints that may limit the possible countermeasures. This issue is further discussed in the next Section.

1.5 Motivation

In industrialized countries, data security and privacy are the most critical properties people may be concerned about when they disclose private information to health-care service providers. As a result, there is greater consistency in the level of securing data due to the data protection laws and effective enforcement and control by responsible government bodies [133]. The case in low-income countries is different. For example, the use of ubiquitous mobile devices to support HIV care projects in sub-Saharan African countries are in production use without addressing basic security requirements[107]. Among several mobile health systems, those developed to function properly in resource-constrained environment are those where data security is either completely forgotten or implemented only partially. In this section, we discuss the motivational factors of this research through real mobile health system use cases.

Health delivery centers in low-income countries are not easily reachable, particularly from remotely and sparsely populated communities. Even if they are reachable, the very few health-care professionals can not keep up with the high demand of health services. Moreover, studies show that due to the rare interaction of patients with the health-care centers, people with diseases like HIV and Tuberculosis are diagnosed later than the stage that was possible to treat them better or to cure. [107]. As an alternative solution, low-income countries initiated a program with community health workers (CHWs) that would deliver health-care services at door-to-door basis. The CHWs are chosen from a community and provided enough training to conduct basic health-care services. An information system is crucial for monitoring CHWs activities, providing a remote channel for professional support, data collection, surveillance and documenting provided care quickly, reliably and securely. The expansion of wireless network infrastructure coverage, notably cellular networks in most parts of low-income countries, created an opportunity to set up a networked channel between CHWs and health-care providers remotely and utilize the channel for exchanging electronic health data, providing remote professional assistance, remote monitoring, data collection, and diagnostic and treatment support.

In order to support HIV/AIDS prevention, treatment and care program in sub-Saharan African countries, the WHO provided a guideline [138] for a "Home based HIV Counseling and Testing" (hereafter HCT) program powered by CHWs. This guideline has been implemented by health-care organizations in sub-Saharan African countries. The aim of the HCT program is to provide HIV cares and treatment services at the home of individuals rather than waiting for them to present themselves at the health-care facility [107, 138]. Each CHWs are assigned a responsibility to visit a number of people door-to-door in the community. Individuals are counseled, examined, and tested by CHWs. The examination and testing processes are registered through a structured paper or electronic forms. The form includes personally identifiable information (aka PII) [36] such as full name, location, home address, and HIV test results [138]. This sensitive information requires comprehensive data protection mechanisms and the importance of data protection is mentioned briefly without details in the WHO guideline report as follows:

> *"Programmes must plan for how HCT service providers (CHWs) will safely move around with data, and how and where data will be stored securely in the field. Electronic records and handheld electronic data-entry devices must be password protected on a secure system and downloaded data stored according to the legal requirements of the data protection act of the country concerned."*

If we take a closer look at the above statements, even if a password is an important element of a complex secure solution on handheld data-entry devices, most of existing systems took the recommendation and implemented user authentication as the only means of data protection while data is stored clear in memory. This might be because it is assumed that the stored data is accessed only through the application layer. But, in actual scenario, the stored data are easily accessible through other data access means. This shows either the term "password based data protection" is misunderstood or implemented as security by obscurity.

It is also recommended to follow each country's legal data protection requirements when data is downloaded and stored on the devices which is offline use case. Offline capability is one of the main functional requirements of data collection tools that exist today due to poor remote connectivity in low-income countries. According to [133], most of low-income countries do not have data protection laws in the exception of constitution of countries or regulation related to individual data protection and security. Most of the countries participated in the HCT program are far from having the legal requirements for data protection. In 2006, a group of experts, organizations and government representatives met in Switzerland and proposed a comprehensive security guideline for electronic systems that collect, analyze, and disseminate HIV related patients information [61]. In the guideline, the group stressed the need of customization of the guideline according to the specific country needs and culture. However, we are not aware of any initiative for customizing the guidelines considering their needs from countries that are involved in HCT programs. It seems that they preferred to implement as a similar fashion as the guidelines is written without customization.

AMPATH is an HIV care program in Western Kenya which implements the HCT guidelines and it is one the largest in the Sub-Saharan Africa [11]. AMPATH has introduced a number of electronic based solutions to manage patient health records since 2004 using open source projects including openMRS [12], an Electronic Medical Record System (EMR), Open Data Kit [13], and a proprietary solution such as Pendragon forms software (a PDA based solution) [107]. As of December 2014, these electronic tools helped to manage, monitor, and follow-up the care delivery to one million individuals and patients in the catchment area of two millions[14]. However none of the tools used really provides adequate security, even if they comply with the HCT guidelines recommendations. The guidelines themselves, after all, do not really give a particularly good or exhaustive list of security requirements, as we have already noted earlier. Below we give a quick overview of the problems.

Open Data Kit based Home-based HIV/AIDS counseling and testing program (hereafter ODK-HCT) is one of the tools used at the AMPATH. ODK-HCT is an Android platform based data-entry tool used to assist and facilitate the data collection process in remote parts of Western Kenya [107]. It is customized from a generic, XForms standard based Open Data Kit Collect (ODK-Collect) [15] to fulfill the needs of the HCT program. CHWs use the household form to collect patient related data using the ODK-HCT app. The collected data are stored on the mobile device without any protection.

[11]http://www.ampathkenya.org/
[12]http://openmrs.org/
[13]https://opendatakit.org/
[14]https://opendatakit.org/2014/05/ampath-reaches-one-millionth-person/
[15]https://opendatakit.org/use/collect/

Later, the CHWs manually export the collected data from the device to the OpenMRS instance at the nearest health-care centers using direct USB connection. In ODK-HCT project [107], the measures taken to secure the system are stated as follows:

> *"To maintain the highest levels of security possible during this project, several measures were undertaken. Despite the capability for data transmission over wireless networks, we elected only to allow direct connections to our data repositories. Collected data were stored on the mobile devices in field, and after transmission to our servers these files were then deleted from the device's memory. Counselors were made personally responsible for their devices and took great care to protect them from theft and loss. As of the time of writing, no devices have been lost or stolen."*

When it comes to data security, the claim made in the above statements (we have seen a similar trends in other MDC system who claims "the highest level of security"), we may give a quick comment by splitting it into three parts: first, while data collected and stored, second, concerns on wireless connectivity, and last, data deletion process after data export. First, there are no data protection mechanisms in place while data is collected and stored. If the device is stolen or lost, collected data are accessible with no effort. In their scenario, they simply hope not to lose devices. Moreover, device sharing among CHWs is one of the functional requirements in ODK-HCT due to tight project budget, but the current solution does not offer data protection from one CHW to the others. Second, there are quite few OpenMRS instances installed in the field hospitals and the measure used to overcome the security concerns on wireless security requires CHWs to travel long distance in order to export the data. This means the collected data may have stay longer on the device and become vulnerable to loss and tampering. Beyond all, the system does not get all the advantages of wireless connectivity to push and pull data in different geographical locations instantly and automate the data submission process. During our field visit at the AMPATH, the candidate has noticed the availability of 3G cellular connectivity in most of part of the catchment area but it is not in use yet. Third, it is stated that the data on the device gets deleted after it is transferred to the OpenMRS instance. However, once the data is written into memory in clear, it is hard to assume that data gets deleted when the delete command is executed since the memory is designed to store data in copies and make it easier to recover [16] [17]. Therefore, even the user deletes the data, it might be easy to restore the deleted data using some tools. Overall, the ODK-HCT solution lacks the data protection mechanisms and requires a complete security solution.

Open Data Kit Clinic (ODK Clinic) is another mobile health solution introduced at the AMPATH. ODK Clinic is customized from ODK Collect to support the clinical decision support system (CDSS) at AMPATH [4]. CDSS aims to improve a health delivery by linking health observations with health knowledge to create an impact on health choices by clinicians.[5] Furthermore, ODK Clinic is designed to replace the paper based clinical summary sheets with electronic sheets. A clinical summary is a summary of patient information printed on a single page. The summary comprises all relevant patient data including patient demographic, medication and treatment history,

[16] `http://www.bbc.com/news/technology-28790583`

[17] `https://www.getsafeonline.org/smartphones-tablets/safe-disposal/`

and lab test results. ODK clinic connects to the central EMR and download and store the summary sheets on the device. The clinician can make offline search on the device and if the patient is not found on the device, the search query can be sent to the central server when there is connectivity. The app is protected with four digit PIN code, and the security level is stated as follows:

"*All data is stored on the device's internal memory which is not user-accessible unless the device is rooted, an unlikely attack vector in the environments where ODK Clinic will be used. Future versions of the application also promise to encrypt patient data.*" [4]

In many of MDC systems reviewed, either the MDC community knows little about how to secure the data or security is not their priority. Most of the assumptions made are not quite right when we come to data protection. The application has been in production use without even addressing the most common security problems and left the issues to be fixed in the future, hence making it even harder to provide a good secure solution. On the other hand, although the assumption on the data protection feature of Android for data stored under the application folder is true, it may take seconds to break the pin code protection the application relies on and get the sensitive data. Furthermore, it is only mentioned that the app is protected with pin, but not how the device protects data when shared among multiple clinicians who all have the same pin code.

The other challenging issue is the conflict between the liability and the security of data.

"Clinicians were concerned with the financial and legal implications of loosing an expensive phone with patient data."[4]

This is a reality in resource constrained settings and really hard to address the concerns altogether. Data can be protected in a reasonable manner, but it is not easy to get a lost device back.

We have also compared the secure solutions adopted by some Java ME [18] based MDCS systems to protect the data both in transit between client and server, and when stored on the mobile device. The results showed that as long as the platform in use offers some standard means of protection, these were adopted, but otherwise not much was done to improve the situation. This turned out to be a major problem especially for the confidentiality of the data stored on the phone, since J2ME does not offer any libraries for encryption. We found that most clients store their data in clear, but with different formats. OpenXdata [19] stores the serialized Java objects, but data elements are still recognizable, while CommCareHQ [20] and EpiSurveyor [21] store all data as XML in clear text. EpiSurveyor, in addition, stores passwords in clear text if one chooses to have the login form pre-filled. Windows Data Gathering (formerly called Nokia Data Gathering [22] client provides password protected data encryption. However a quick look at the implementation revealed that the encryption key is a direct MD5 hash of the password, and it is stored in clear on the phone. Hence, one can retrieve both the key

[18]Java ME: `http://www.oracle.com/technetwork/java/embedded/javame/index.html`

[19]openXdata: `http://www.openxdata.org/`

[20]Commcarehq: `http://www.commcarehq.org/home/`

[21]Magpi: `http://home.magpi.com/`

[22]Windows Data Gathering: `http://www.windowsphone.com/en-us/store/app/microsoft-data-gathering/a2bd7f51-6c7f-48d4-9fa7-12b35c550848`

and the encrypted data, rendering encryption useless.
There might be several reasons why these systems have been used in the health-care domain while not being as secured as we expected to be. However, through our experience, we deduce the following main factors.

Generic Nature of MDCS: MDCS have been used in several parts of the world to tackle challenges in health-care, agriculture, election, humanitarian aid, disaster recovery, education sector, and many more. For instance, ODK has been used for as a health-care tool [4, 107], for environmental monitoring and food security [23], in election [24], for flood disaster recovery [129], for monitoring and evaluation in agricultural work in Zambia [25] are some of the use cases of ODK around the globe. In spite of the differences in the use cases, ODK core principle is to maintain a single code base for each toolkit and leave the customization responsibility to the users community, so that the core ODK is kept as generic as possible. Since security requirements in the different ODK usage sectors vary, it turns out security is not one of the core components of ODK Toolkit. Additional features are added as an add-on to the core application, and security is an add-on category.

Lack of Awareness: In most cases, security is used as marketing tool, but most systems have not addressed basic security issues in satisfactory manner. Some of them do not inform of their system limitations and capabilities in a clear and understandable way and some of them made a wrong assumption which leads users community to have false confidence in using the systems. This is due to lack of awareness.

Lack of National Data Protection Laws and Regulatory Body: If on one side designing such a product is more expensive, the lack of compliance of a MDCS to the regulations of a certain country can be a deal-breaker for its adoption.

Resources-Constrained Environment: Funding is one of a key problem in open source communities which makes it more likely to prioretize functional requirements of applications over non functional requirements. It is not easy to get all on board to address the security challenges.

Low Demand: Some MDC providers might have inclined to put no resources for addressing the security issues because they feel the low-income countries are in need of services rather than data protection or privacy. They feel no demands from the users community which mostly happens to be in low-income countries.

These are some of the challenges that motivated the candidate to conduct this research and contribute a secure solution, create data security awareness among mobile data collection tools users and developers community.

[23]CGIAR's Climate Change, Agriculture, and Food Security with ODK, [Last Accessed: April, 2015] `https://opendatakit.org/2014/04/cgiars-climate-change-agriculture-and-food-security-with-odk/`

[24]Carter Center on Leading Edge of Technology Use in Election Observation, [Last Accessed: May, 2015] `http://blog.cartercenter.org/2012/08/06/carter-center-on-leading-edge-of-technology-use-in-election-observation/`

[25]ODK in Agriculture, [Last Accessed: March, 2015] `https://opendatakit.org/about/deployments/`

1.6 Collaborations

Since the beginning, our security research has strong ties to partners project. We started with the openXdata project, and later we partnered with the Open Data Kit community and mUzima project [26]. The mUzima project is a client - server solution from the University of Indiana, designed to serve as a tool for accessing patient data for example from the OpenMRS server [27]. At the time of this writing, the mUzima project is at the development stage, and our contribution to the project will be published in the future. Next, we briefly introduce the openXdata and Open Data Kit projects.

1.6.1 The openXdata Community

Our research effort first emerged late 2010 from a formal request of the openXdata community to conduct a security analysis on their open source mobile data collection system called openXdata [28], an open source, freely available, a server-client Java based MDC solution. We then formed the MDCS security research group and focused our attention on mobile data collection, mostly in the domain of health-care. We created a collaboration between the Department of Informatics, and the Centre for International Health, at the University of Bergen to work together in a cross disciplinary study aiming to solve the security challenges in MDCS. At the time, there was an immediate need for a standard secure solution for OMEVAC project (Open Mobile Electronic Vaccine Trials)[64] with an objective to develop an electronic system for management of vaccine trials in low-resources settings and mVAC project with an objective to develop a mobile device based solution to record immunizations at the individual level[29]. openXdata has been since one of our collaborators. Their open-source MDCS follows the previously described concept of downloading forms from a server and filling them out on mobile devices, and is being used in a number of projects in low-income countries such as Pakistan [63, 65] and Uganda [63, 102]. When we started, Java Mobile Edition (Java ME) was the dominant mobile platform as it ran on the target feature phones and most of MDC tools were developed using Java ME, including openXdata. In a few years, the dynamic nature of the mobile platforms development pushed the JavaME platform out of the market in favour of the Android platform which gained rapidly the majority of the market share. Android devices are getting cheaper and the new standard for data collection purpose. We then began a new collaboration with the Open Data Kit (ODK) community and tried to port the secure solution developed for Java ME based MDCS to Android.

1.6.2 The Open Data Kit (ODK)

Open Data Kit (ODK) is a free, open-source suite of standard tools that helps organizations, field researchers, authors, public health and medical practitioners to conduct data collections using Android mobile devices, data aggregations, analysis, and presentations. ODK's core developers are researchers at the University of Washington's

[26]mUzima: `www.muzima.org`

[27]OpenMRS, [Last Accessed: May, 2015], `openmrs.org`

[28]openXdata, [last accessed: April 2015] `http://www.openxdata.org/`

[29]mVAC: `http://www.uib.no/en/rg/childhealth/65382/mvac`

department of Computer Science and Engineering and active members of *Change*, a multi-disciplinary group exploring how technology can improve the lives of under-served populations around the world[30]. ODK began in 2008, as a Google.org sponsored project. In 2009 it was first deployed in Uganda and Brazil. From 2010 on many other contributors participated in the development of the ODK tools becoming soon a growing community of developers and users. The project is currently funded by a Google Focused Research Award and by donations[31].

The ODK Community consists in two user groups: the ODK-Developers and the ODK-Users. As an open source community, all the code is publicly available and members are welcome to contribute with their questions, bug reports, new features, and feature requests. Beyond this community, there is a larger set of projects that are offspring of ODK by extending or adding some particular features or customizing the product for certain settings. Even if many of them are commercial software, often they contribute to the development when the newly developed features are interesting for the core ODK team or they contribute patches to the core ODK code base. The diffusion of ODK in different parts of the world has opened up business possibilities for many small companies developing customizations and assistance on deployment. This phenomenon acted as a business driver encouraging users and developers to participate in the community and setting up an ecosystem of mutual convenience. Even more, the ODK team has a dedicated page with a title "Help for Hire" for promoting the small companies by displaying company description, location, and web page. As of this writing, there are close to 60 small companies registered around the globe and providing the customization and assistance services [52].

The candidate visited the core ODK developers team at the department of Electrical Engineering and Computer Science at the University of Washington. The visit helped the candidate to share first-hand experiences with ODK developers on subjects including data protection, data security and privacy awareness, work load and compromises, simplicity, ODK as solution survival business model, maintenance, challenges in new product release, and so on. The candidate field visit main objective was to have better understanding on how the ODK applications work and analyze its data protection level including authentication, data at rest and during data transmission.

1.7 Scope of the Research

The main goal of this research work is to propose a security solution tailored for mobile data collection systems in mHealth domain. The security solution is designed to be agnostic to any MDC systems and the solution through its protocol can be ported to any platform. However, in this study, the candidate worked closely with partner projects who mainly provide Java ME and Android based MDC systems. Hence, when it comes to actual implementation, testing, and evaluation, these are the two platforms that have been used. Besides, since most MDCS projects are low-budget and focus on low-end devices, iOS devices would not have been a representative platform. Still, some basic security issues are conceptually identical no matter the underlying system, and most of the proposed solutions can be applied in several contexts and mobile platforms.

[30]Change at University of Washington, [Last Accessed January 2015], `http://change.washington.edu`
[31]Google Focused Research: `https://opendatakit.org/about/`

Solutions we designed for older Java ME based devices, for instance, are still relevant and easily portable to more modern platforms like Android.

More specifically, in this work we have considered thick (native) client applications. Browser based MDCS or MDCS that are being developed as a hybrid application, which is a combination of native and HTML/Javascript/CSS, have not been considered. As of this writing, there are some research activities in this direction which will be presented in the future work, but we do not have concrete results yet. Furthermore, we considered mHealth systems where human collectors enter data, and we are not considering the situation where a sensor is feeding data to a remote server or device. See also the next chapter for a more detailed description of the MDCS we consider.

1.8 Dissertation Overview

This dissertation is organized into 9 chapters that are designed to be read in sequential or randomly order. A modular security framework for mHealth is presented in chapter 4. The rest of the chapters after chapter 3, present an in depth analysis and solution for each module within the security framework. Therefore, the reader is recommended to read each chapter to get insightful understanding of the entire security architecture.

Chapter 1 briefly discuss the motivation, the contributions, and the scope of the research.

Chapter 2 presents mobile data collection systems, architecture, mobile device platforms used in mobile data collection systems, discusses general security requirements for MDC systems such as user authentication, secure storage, and secure communication, and review platforms security model

Chapter 3 introduces SecureMDC, a modular security framework for mobile data collection systems and discusses integration approaches to an existing Android based mobile data collection systems

Chapter 4 presents an Authenticator Module of the security framework which is introduced in Chapter 3. This module provides local and remote user authentication, serves as a secure keys store, and account management.

Chapter 5 presents a Secure Storage Module of the security framework. This module provides data confidentiality, key management, recovery procedures and integrity on mobile devices.

Chapter 6 presents a Secure Communication Module of the security framework and secure cloud storage. This chapter discusses a secure solution for data transmission and data protection for cloud based mobile data collection systems.

Chapter 7 explores different approaches for secure application provisioning.

Chapter 8 discusses the candidate's overall approaches, remaining challenges, and future works.

Chapter 9 concludes with related works, conclusions, and contributions.

2 Mobile Data Collection Systems And Its Security

This chapter aims to familiarize the reader with mobile data collection systems, their working principles, architecture, and data flow. In the next sections, we describe more in detail what mobile data collection is, how mobile data collection systems work and what problems we try to solve with our work.

2.1 Mobile Data Collection

The family of systems studied for this research are centred on a common task: data collection. This task can be conducted in many possible ways and it can be easily confused with something different. For this reason, we need to define the concept of data collection and choose a model to describe it in a consistent way during the analysis. We have already informally done that in the previous chapter, but we formalize it better here.
Starting from the basics, the first question to answer is what can be meant by data collection. A general definition is given in Medical Subject Headings (MeSH)[1]:

> *Def: Data Collection is a systematic gathering of data for a particular purpose from various sources, including questionnaires, interviews, observation, existing records, and electronic devices. The process is usually preliminary to statistical analysis of the data.*

This definition is a good synthesis of the main features of data collection. First of all, it's defined as a gathering process, intended to serve other specific analysis tasks. Data is collected "...for a particular purpose..." indicating that the variables of interest must be specified in advance, in order to answer stated research questions, test hypotheses

[1]Medical Subject Headings (MeSH), [Last Accessed: March 2015], `http://www.ncbi.nlm.nih.gov/mesh/68003625`

and evaluate outcomes. For this purpose we do not only have to specify which data has to be collected, but also how.

According to [52, 115], a formal data collection process is necessary as it ensures that data gathered is both defined and accurate and that subsequent decisions based on arguments embodied in the findings are valid. Another feature of great importance is that information is gathered "...from various sources...". The variety of sources leads to different approaches involving the use of many possible tools, but also affecting the process and the results. This is the reason why the concept of Data Collection is so broad that often it is not used directly, or it is used to mean a more specific scenario. It is possible to classify a particular Data Collection scenario in terms of:

- Collected Data
- Collection Tools
- Collection Process

Collected Data can be pieces of simple information codified in the form of strings, numeric values and similar, as well as more complex ones such as pictures, audio or video recordings, geographical locations. Measurements coming from sensors can also be considered. According to the type of data to be collected and many other factors, the collection tools are chosen. The collection process describes how these tools are being used. In this work, we refer to a set of data collection scenarios in which the tools are mobile devices and the process involves form-based surveys. For this reason, here is given a more specific definition of data collection using mobile devices, also known as Mobile Data Collection (MDC) [52].

MDC is defined as the targeted gathering of structured information using mobile devices such as normal cellphones, smartphones, PDAs, or tablets [21]. To avoid confusion it is important to understand the difference between MDC and crowd-sourced data aggregation. In this last case, data aggregators collect unstructured data from services such as social networks, email, and SMS, and then mine data for information. By contrast, mobile data collection is performed using designed surveys to collect specific information from a target audience. The collectors can be either staff trained from an organization, or the target population being studied can be surveyed directly on their personal mobile devices [52].

2.1.1 Mobile Data Collection Systems

A Mobile Data Collection System (MDCS) is built upon a set of components to provide functionalities including form design, collect, manage data. While the architecture may vary greatly, there are some components that are common to all MDCS. For instance the form designer, MDC server, modules, and client applications or form runner as shown in figure 2.1. These components are categorized into form design, data collection, and data management processes. We present each process briefly as follows.

2.1.1.1 Form Design

Form Designer is a software tool used to create an electronic form from scratch or used to convert a traditional paper-based form into an electronic form (aka Form Defini-

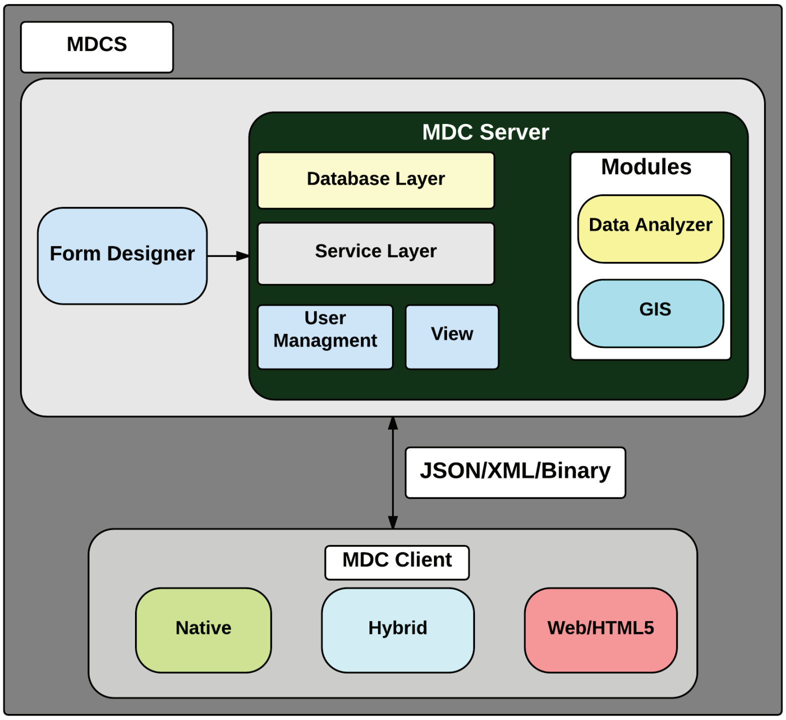


Figure 2.1: Mobile Data Collection System (MDCS)

tion). The form designer assists users to formulate relevant questions used to gather information needed to answer vital research questions. With some exceptions, most of designers provide graphical user interface to create simple or complex forms. They support question types including text, numbers, date, time, single or multiple select, geo-tagging, and multimedia. Upon this writing, a number of form designers are available on the market as open source and commercial projects including openXdata form designer [2], WEPI and Voozanoo 3 from Epi-concept [3], ODK Build and XLSform from OpenDataKit (ODK) [4], Vellum from Dimagi [5], and many others. Furthermore, the user can define skip-logic and validation criteria for a specific form through the form designer.

[2]openXdata, [Last Accessed: March 2015], http://www.openxdata.org/
[3]WEPI and Voozanoo 3, [Last Accessed: March 2015], http://www.epiconcept.fr/en
[4]XLS and ODK Build, [Last Accessed: March 2015], http://opendatakit.org/
[5]Vellum, [Last Accessed: March 2015], http://www.dimagi.com/

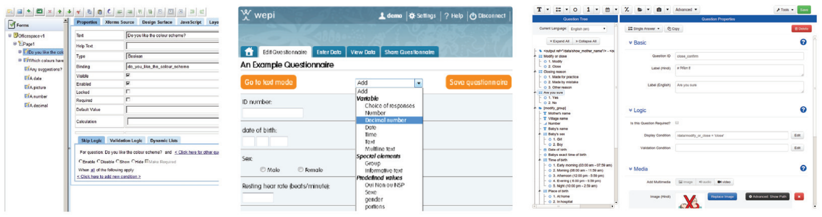


Figure 2.2: From left to right: openXdata, Wepi, and Vellum form designers screenshot

2.1.1.2 Data Collection

In the context of MDCS, data collection is a process of data gathering using a client application running on a mobile device or manually coded SMS forms. The concept of the mobile device applies to a broad range of tools, including regular low-end phones, Java-enabled phones, smartphones, tablets, and notebooks. From now on we focus only on these set of devices and clients that are implemented as mobile applications as native apps. Although the ubiquitous nature of SMS in low-and-high end phones, SMS coded forms based data collection does not handle complex forms and lacks skip logic and validation techniques. In this work, despite widely deployed SMS based solution (RapidSMS[6], MOTECH[7], and FronlineSMS[8]), we have not considered SMS based data collection. However, the proposed secure solution applies to browser based apps or SMS with some customization.

Before we devise into details of our specific client applications, it is important to have a clear understanding of how client applications are developed on the fragmented mobile platforms environment. Upon this writing, there are three categories: *Native apps*, *web/HTML5 apps*, and *Hybrid apps* and they are defined as in [9]:

> *Native apps are specific to a given mobile platform (BalckBerry or Symbian or iOS or Android) using the development tools and languages that the respective platform supports (e.g., Objective-C with iOS, and Java or C with Android, or J2ME with Symbian or BlackBerry). Native apps look and perform the best.*
>
> *Web/HTML5 apps use standard web technologies - typically HTML5, JavaScript and CSS. This write-once-run-anywhere approach to mobile development creates cross-platform mobile applications that work on multiple devices. While developers can create sophisticated apps with HTML5 and JavaScript alone, some vital limitations remain at the time of this writing, specifically session management, secure offline storage, and access to native device functionality (camera, calendar, geolocation, etc.)*

[6]RapidSMS, [Last Accessed: April 2015], https://www.rapidsms.org

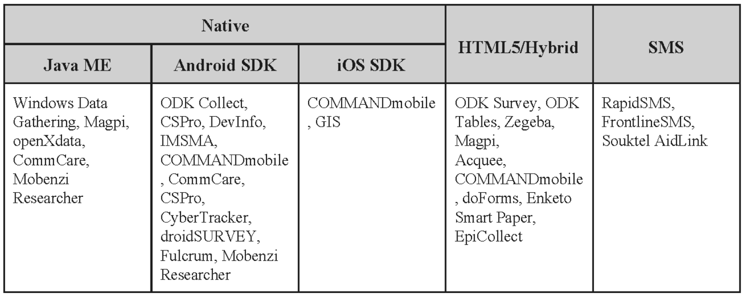
[7]MOTECH, [Last Accessed: January 2015], http://ghsmotech.org

[8]FrontlineSMS, [Last Accessed: January 2015], http://www.frontlinesms.com

[9]SalesForce, [Last Accessed: January 2015], https://developer.salesforce.com/page/Native,_HTML5,_or_Hybrid:_Understanding_Your_Mobile_Application_Development_Options

"Hybrid apps make it possible to embed HTML5 apps inside a thin native container, combining the best (and worst) elements of native and HTML5 apps."

There are a number of mobile platforms such as Android, iOS, Firefox, Blackberry, Tizen, Ubuntu Touch, Windows Phone, and Sailfish. These platforms provide their own software development kit (SDK). When we started our research, feature phones supporting Java ME like Symbian based Nokia handsets were the cheapest and the dominant platform globally [10]. Within a few years, however, smartphones became increasingly cheaper and more popular until today's situation where Android is the leading mobile platform in the market [11]. MDCS providers adjusted themselves according to the market shift and began developing an Android version of their client, or cross-platform apps that could run on most smartphones as shown in Table 2.1.



Native			HTML5/Hybrid	SMS
Java ME	Android SDK	iOS SDK		
Windows Data Gathering, Magpi, openXdata, CommCare, Mobenzi Researcher	ODK Collect, CSPro, DevInfo, IMSMA, COMMANDmobile, CommCare, CSPro, CyberTracker, droidSURVEY, Fulcrum, Mobenzi Researcher	COMMANDmobile, GIS	ODK Survey, ODK Tables, Zegeba, Magpi, Acquee, COMMANDmobile, doForms, Enketo Smart Paper, EpiCollect	RapidSMS, FrontlineSMS, Souktel AidLink

Table 2.1: Samples of Existing Mobile Data Collection Systems

2.1.1.3 Data Management

MDCS uses a centrally located server for managing, distributing form definitions, and aggregating collected data. The number of MDCS server-side services varies from one system to another, but most of them provide some or all of the following functionalities:

- Receive and aggregate submitted data from any server compliant client applications.
- Provide data repository
- Data export with supported formats
- Basic visualization features on the collected data through maps, graphs, and bar charts
- User management

[10] Vision Mobile Report on Mobile Platforms: The Clash of Ecosystems, [Last Accessed: May 2015], `http://www.visionmobile.com/blog/2011/11/new-report-mobile-platforms-the-clash-of-ecosystems/`

[11] Gartner smartphones market share, [Last Accessed: June 2015], `http://www.gartner.com/newsroom/id/2996817`

- Integrated data analysis tools such as R[12].

As of this writing, there are a number of server applications hosted centrally for managing data collection such as Open Data Kit Aggregate[13], openXdata server [97], OpenMRS [95], OpenClinica [91], DHIS2 [30], formhub [86], CommCareHQ [22], and Magpi [72]. Once a form definition is created, an organization or a research institute interested in conducting the research has to decide where to host the server for managing the forms and the collected data. There are four alternatives:

1. **MDC Server on Privately owned network or Private Cloud**: the server application can be deployed on a privately owned network or a private cloud that may exist on or off premises. The server application deployment on private OpenStack cloud[14] is a typical example. In both cases, the organization may own, manage, and operate the application and the hosting environment. The deployment may demand a high level of expertise and substantial resources such as dedicated IP, hosting location, network connectivity, hardware, power, cooling system, and obviously backup and possible fail-over solution. However, it may certainly provide a complete control over the collected data and form definitions.

2. **MDC Server as Software as a Service (SaaS)**: an organization may use data collection tools modeled as Software as a Service (SaaS). This solution requires low resources compared to the previous one, and it is the easiest and can be a cost-effective way to setup the hosting server within a short time. The only thing that is required by the project is to set up an account on the server, download the client on the mobile devices, and pay per number of users and data traffic, as in dedicated services like formHub, Commcarehq, Magpi, HarvestYourData [56]. All the data stored on these type of servers will be however potentially exposed to third party entities who might not be authorized to access it. For instance, the formhub is hosted on Amazon Web Services (AWS)[15] public cloud and the Magpi and HarvestYourData are on the Rackspace public cloud[16]. Our quick security review on these systems indicated that the systems are not transparent with issues such as data ownership, data protection on public cloud, and life cycle of the collected data. Instead of presenting their approaches to these concerns and challenges, they provided a link to the cloud provider and the system users do not have a clue about how their data is accessed or managed remotely.

3. **MDC Server on Infrastructure as a Service (IaaS)**: As in the first case, a virtual server is offered by third-party as Infrastructure as service (IaaS) and the project administrator setups the server and load MDCSs server application. Amazon EC2 is a typical example of IaaS. Theoretically, this would allow the project to have full control over the server applications and collected data, but IaaS providers do not take any responsibilities related to security breaches and data loss on the server that potentially would still expose data stored on it to a third party.

[12]R, statistical computing tool, [Last Accessed: April 2015], `http://www.r-project.org/`
[13]Open Data Kit Aggregate, [Last Accessed: April 2015], `https://opendatakit.org/use/aggregate/`
[14]OpenStack, [Last Accessed: May 2015], `https://www.openstack.org/`
[15]Amazon Web Services, [Last Accessed: May 2015], `http://aws.amazon.com/`
[16]Rackspace, [Last Accessed: May 2015], `http://www.rackspace.com/`

4. **MDC Server Application on Platform as a Service (PaaS)**: Platform as a Service (PaaS) provides a range of platforms, and the project owner needs to deploy the server application onto the platform. Open Data Kit Aggregate server is a typical PaaS application developed to be deployed in the Google App Engine. Apart from limited application level control, PaaS providers do not allow MDC server application owner to manage or control the underlying cloud resources such as network, operating systems (OSs), servers, and storage [71]. In general, IaaS is more flexible and provides more resources and data control over PaaS.

The paradigm of Cloud Computing has been naturally embraced by the MDC service providers, attracted by the idea of outsourcing all the server part, making it fast setup, scalable, cost-effective, and less maintenance. These are some of the enabling factors for the development of these type of systems, especially projects in low resource settings. In general, as we discussed above, MDCS can be deployed on a cloud as SaaS, PaaS or IaaS. MDC SaaS providers are growing in number. They provide a complete MDC solution out of the box. The only expected task from the user is to register and pay per usage of the service. On the other hand, the PaaS and IaaS MDC service deployment approach provides the flexibility to the user to customize and deploy their MDC application and take care of the management and maintenance. However, from a security point of view, all these scenarios share the disadvantage of involving a third party in the data collection process. In the SaaS service model, the SaaS MDC provider itself, as well as any other provider on which it is built upon such as the cloud infrastructure provider. In the Paas and IaaS service models, we still involve a cloud infrastructure provider as third party, but at least we have a better control on who is handling our data in the IaaS case. With a downloadable data collection back-end, it is sometimes possible also to install it on a private cloud, or on a custom server held locally. Having a larger number of possibilities requires a more careful choice, and the trade-off is between having more control on the collected data or improving their availability. The whole data collection process is focused on data, making availability the most important quality to pursue.

For the above specified reasons, the cloud seems to be the wisest choice, but at the same time we need to protect the stored data, mitigating the risks deriving from the lack of control on their location. When the collected data are sensitive, and especially for health related information, the situation is even more critical, because the system has to comply with regulations that may differ in the country where the data are actually stored. The problem of secure cloud storage is one of the main existing challenges for mobile data collection. In the secure cloud storage chapter, we will discuss the challenges and proposed solution when MDCS deployed in a cloud.

2.1.2 MDCS Data Flow

In this work, we focus on form-based mobile data collection, where information is gathered using structured forms. Unlike paper forms, electronic forms may include any attachment made available by the collection device, as GPS coordinates, photo, video, and input data from internal or external sensors. A form is composed of fields, each one with a name, a data type and some metadata. The form definition encapsulates the project-specific tasks while the underlying system is designed to be unaware of the data collected, being only focused on the higher level tasks [52].

It is important to express the difference between a Form Definition (Form Def or a Form) and a Form Data (also called Form Instance or Submission). While Form Definition describes the format and properties of the data we want to collect, a Form Data contains a unit of the collected data. We can also think about it as the relationship between a Class and its Instances. The process of form-based mobile data collection is summarized as follows:

1. **Design**: the form definition is designed and uploaded to the server.

2. **Download**: collectors download the form definition.

3. **Collection**: collecting instance data by filling the downloaded forms and storing on the device.

4. **Submission**: the collected data are uploaded to the server.

5. **Analysis**: data on the server are processed and used by decision makers, on-line or off-line.

These steps must be seen in the context of the data flow described in 2.1, emphasizing the distinction between the client part and the server part, as well as the two-way communication and the direction of data collection.

The collection process itself, schematically illustrated in figure 2.3. The collection process starts by setting up the server with the list of predefined collectors, form managers, local administrators and data viewers, which all receive a user-name and password to get access to the server. The project organization remains almost unchanged over time. The form managers design the form definitions to be used for each project, and the collectors can download these definitions on their mobile client to start collecting data. Data collectors must first of all install the application on their mobile devices, and then register to the server for the first time, where they are presented with the list of possible form definitions they can download. A predetermined workflow might decide in which order the forms should be downloaded. The form definitions are then downloaded, and the data collection starts. Once the forms are filled, they are regularly uploaded to the server and deleted from the phone. At this point, the form managers and data viewers can synchronize with this central repository to download the most recent data to their personal machines for further analysis.

Other differences among MDCS regard the mode of use of the client and the technology used. For example whether multiple collectors share a mobile device or if a collector can modify form submissions, and which platform the client application has been developed, i.e. Java ME, Android, iOS or HTML5 browsers.

The model we consider in the rest of this work is meant to be general enough to reason at a product-independent level, but at the same time not too abstract, focusing on the form-based, one-way mobile data collection as it seems to be the most used in practice.

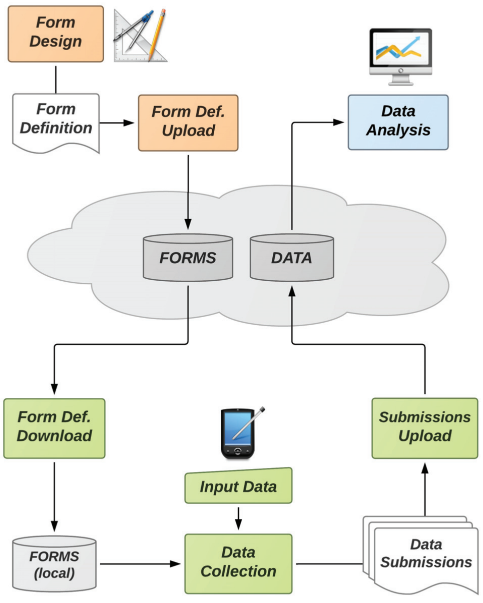


Figure 2.3: Mobile Data Collection System Data Flow Diagram

2.1.3 Advantages of Mobile Data Collection Systems

Paper based data collection has many obvious pros and cons. Paper cons mostly related to data quality, cost, time, error-prone, storage need, usability, data management, and transport. Some of paper-based data collection pros are: people are familiar with paper, require less training, no power source related problem, and less probability of losing data. Working on a desktop or laptop solves many of the problems of the paper, but introduces a dependence on power sources and makes the data collection in a specific location such as health centers. While the collected data are portable, the data collection tools are not.

Internet, wireless network, and mobile technology brings great opportunity on this front, providing a communication channel that can be used to connect data collection sources to a server for storage and analysis. Unfortunately, this introduces another dependency, the availability of network connectivity. These constraints have a huge impact when data is to be gathered in areas where electric power or network coverage are not guaranteed or not available at all. These settings are common in rural areas and in developing countries, where the coverage of cabled network connection has expanded at a slower rate compared to the wireless networks, and most of the people, even when not having a personal computer, have a mobile phone. It is not a surprise then, that in these cases mobile data collection has often been chosen as a replacement for paper-based forms without passing through a PC-based phase at all [52]. Another constraint in these contexts is the restricted economic budget that many projects dispose of, discouraging the use of expensive custom hardware and driving towards the use of cheaper general-purpose devices. Technology achievements allowed manufacturers to sell increasingly performing devices at lower prices, enabling the adoption of them as rich clients for data collection [52].

With the advent of smartphones, the capabilities of consumer devices have gone beyond the original needs, inspiring a new era of MDCS. Cameras, sensors, and on-board GPS systems provide new sources of data, along with more standard interfaces enabling connectivity with external tools.

2.2 Relevant MDCS Systems

As of this writing, there are several mobile data collection systems (MDCS), both open source and commercial. However, as it stated in the chapter 1, we began this project in partnering with projects such as Open Data Kit (ODK), openXdata (OXD), and mUzima. The proposed solution is designed in the context of these systems requirement. Thus, in this section, we briefly introduce the systems.

2.2.1 Open Data Kit (ODK)

Open Data Kit (ODK) is an open-source, modular toolkit that enables organizations to build application-specific information services for use in resource-constrained environments [16, 55]. The toolkit is a collection of Android based applications and cloud-based server application. Upon this writing, ODK is evolving from ODK 1.0 to ODK 2.0. ODK 2.0 is not only a new release but also a complete redesign of ODK 1.0 to keep the state of the art and addresses some challenges in ODK 1.0. An uni-directional device-to-server data flow in ODK 1.0 has been re-architecture to provide bi-directional data flow in ODK 2.0. Consequently, it is possible to retrieve submitted data and make changes across multiple devices using ODK 2.0 tools. Despite the ODK 1.0 is deployed in several countries, the ODK core team does not support or actively work on ODK 1.0. ODK 1.0 is mainly maintained and new features are contributed by the community who are using ODK 1.0. The ODK team has outlined the new set of design principles that are considered under ODK 2.0 development. The principles include easy application source code customization with limited programming experiences, simple form designer and human readable output using JSON, making changes on submitted data through bidirectional data flow, and so on. As a result of this, the ODK 2.0 toolkit consists of the following mobile and cloud-based applications:

ODK Tables: is a mobile app developed that provides a way to synchronize, store, view and manipulate data on a mobile device. This application is designed to support bidirectional data flow to make changes on an existing data.

ODK Survey: is the new version of ODK Collect (a native mobile from ODK 1.0) which provides the basic mobile data collection (offline, online) using form definition. As it is claimed in [16], its easiness in customization makes ODK Survey different from the previous release ODK Collect. For this particular purpose, run-time language (e.g. Javascript) is used over compiled time language (e.g. Java). The main operations are:

Download Form Definition: the client connects to the aggregate server and shows the list of available forms, the user chooses one or more, and they are downloaded in the form of JSON or XML files. An alternative to this is performing the process off-line by manually loading the forms on the external memory of the phone.

Fill-in form: based on the form definition, the phone creates a wizard that allows to compile fields one by one, adding multimedia attachments if needed, also created on-the-fly. At the end of the process data can be saved temporarily or marked as finalized, meaning that they are ready to be sent to the server and do not require modifications. Locally, data are stored in SQLite database and files such images stored in the file systems. Data can be submitted over-the-air or transfer them manually using the SD-Card.

Uploading Collected Data: the uploading process is mostly designed for typical data collection scenarios in which network coverage can be unstable. Data are sent one, by one and the operation can be interrupted and resumed later.

ODK Sensors: is not an actual sensor application, but it is a framework that simplify the interface between a variety of external sensors and consumer Android devices [15]. The framework is designed to facilitate sensors based application by simplifying through abstraction the underlying details of the sensor-specific code so that developers focus on the functionality aspect of the application.

ODK Scan: is a mobile application that uses computer vision techniques to convert paper forms into electronic forms [29]. At the beginning, the paper forms are designed using machine-readable data types such as bubbles and checkboxes, and the ODK Scan application automatically recognize the data types.

ODK Submit due to intermittent connectivity, often data submission is initiated by the user and organizations are interested to control data transfer costs. ODK Submit aims to develop a service that enables organizations to specify parameters such as data priority, data importance, deadlines, and the cost of the transport mediums. ODK Submit then factors in the device's connectivity history, and intelligently uses the connectivity available (e.g., SMS, GPRS/3G, Wi-Fi) to create a priority routing system that improves data timeliness in the intermittent and expensive connectivity of the developing world [16]. As it is discussed in [16], connectivity history can be used to decide which connectivities to use at a particular time.

ODK Aggregate: is server side application, designed for aggregating data coming from different sources and multiple devices. ODK aggregate provides services such as data presentation and data analysis and export. ODK Aggregate can be hosted on the cloud or dedicated privately owned server. In particular, Aggregate is designed to be hosted in the Google App Engine Platform as a Service infrastructure. With Google App Engine, Aggregate support two types of user authentication services: via Google OAuth or home-grown RFC2617 [42] based HTTP Digest Authentication authentication. In addition, anonymous access is supported, allowing anyone to download a form and publish data, useful when we want to adopt a crowd-sourcing approach where the collectors are not known to the organization or for demonstration purpose. In summary, ODK Aggregate provides the following services:

User Management: this is needed to manage access control and is the primarily responsibility of the site administrators. From a web client, they can register new users and assign roles to them. The roles are:

Data Collector: able to download forms and submit data.

Data Viewer: able to log onto the ODK Aggregate website, filter and view submissions, and export them in many formats.

Form Manager: has all the previous capabilities plus the ability to upload a form definition, delete a form and its data, and upload submissions manually through the ODK Aggregate website.

Site Administrator: all the previous ones plus the ability to manage users.

The overall ODK 2.0 system architecture [16] is shown in figure 2.4.

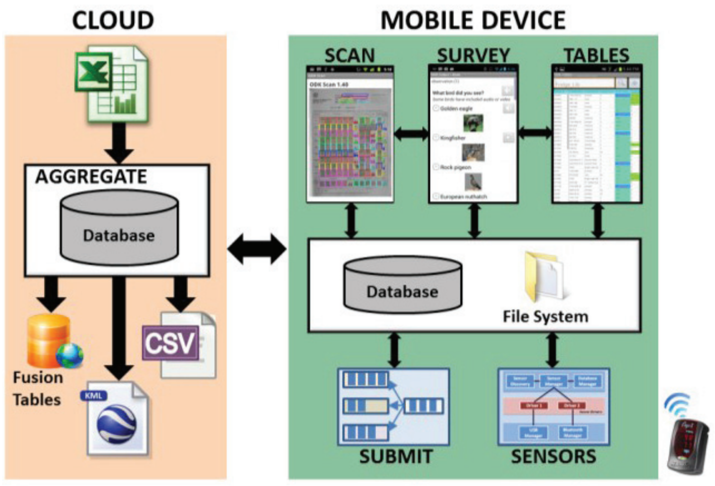


Figure 2.4: Cloud based server component (left) and client-side mobile applications (right). ODK Aggregate server provides services including forms distribution, data aggregation, synchronization, and export. The client-side apps share a common database/file system. ODK Scan, Survey, and Tables (above the data storage) are tools designed for data collection, processing, and presenting. Submit and Sensors are tools for optimizing the services and abstract the underlying complexity of use of sensors respectively.

2.2.2 openXdata (OXD)

openXdata (OXD) system is a community supported open-source mobile data collection system that is primarily designed for data collection in resource-constrained environments[17]. OXD and ODK share similar basic data collection design principles both on client and server and differs in low-level implementation, choices of mobile platforms, data exchange formats, form definition structure, and authentication protocol. One of the other main difference between OXD and ODK is OXD client is developed as a single fully-fledged app (which all services consolidated into single app) and ODK is developed as multi-applications, each app is designed to provide specific functionality. Similarly, OXD consists of a client and server components that provide tools to:

- Design forms,
- Manage the users involved in the study,
- Collect Data using Java ME-based mobile application. At the time of this writing, there is an ongoing system and tools development within OXD community to transform the software to reflect the current state of art, for instance, creating a new form designer and runner engines based on HTML and JavaScript. The form runner is a client-side application engine that handles form presentation, data gathering, validation, skip logic, and other features. The form runner can run on the normal web or mobile browser or can be packaged as a hybrid application and distributed as a thick application through for instance Google Play Store.
- Server-side application for data Aggregates, storage, analysis, presentation, and export collected data in many formats.

2.2.3 mUzima

mUzima is a work-in-progress project from University of Indiana consisting mostly in the client-side application, which aims to develop an adaptable application for mobile phone and tablet platforms with primary implementation in the health space that allows to work both online and offline[18]. The current implementation is customized to work well with the widely used open source medical record system, OpenMRS[19] which then takes the role of server-side application. When the application developed will be completed, the application is planned to be deployed at the AMPATH [20], one of the largest HIV care program in Sub-Saharan African countries. The AMPATH program serves a population of 3.5 million people in western Kenya and the program delivers primary health care services, cares and treatment services for more than 158,000 HIV-infected patients, and prevent HIV/AIDS through door-to-door counseling and testing[21]. Therefore, mUzima tools must address adequately any security concerns before use at the AMPATH and other OpenMRS related projects around the world.

[17]openXdata, [Last Accessed: June 2015], `openxdata.org`

[18]mUzima, [Last Accessed: June 2015], `muzima.org`

[19]OpenMRS, [Last Accessed: May 2015], `openmrs.org`

[20]APMATH-Kenya, [Last Accessed: April 2015], `http://www.ampathkenya.org/`

[21]AMPATH Partner, Indiana University, [Last Accessed: December 2014], `http://medicine.iupui.edu/INTM/kenya`

2.3 Requirements Analysis for MDC Systems

In the previous sections, we presented what mobile data collection is and its importance in the domain of mHealth. In the next few sections, we briefly discuss the mobile data collection systems from a security perspective, identify general functional and non-functional requirements. There are some additional specific requirements to a particular service such as user authentication. Hence, these requirements are discussed later in Chapter 4, 5, and 6.

The domain of mHealth is not the only example where we need to protect the data. As the MDCS cover different kind of tasks, different needs suggest the adoption of a secure solution. One of these needs is the physical security of the collectors, that may be located in a remote hostile area, collecting secret data. For example collected data can be reports about human rights abuses by a government or a company [52].

In the next subsections, we define the problems related to security in mobile data collection, exploring how these are perceived and addressed currently and which are the opportunities, in order to locate the role of this research in the context of other works. We also find here the description of the case study which was used as a starting point for developing research intentions. At the end, the precise goals of this work are defined.

2.3.1 Functional and Security Requirements

An important aspect of concerns is the functional requirements we can expect from these kind of applications on the client side, as they also put some constraints on the security solutions that can be designed. After thorough discussion with ODK and openX-data community, we identified the following functional requirements:

Sharing: The same mobile phone might be shared among multiple collectors while the same collector might use more than one phone. The client must therefore support multiple users.

Off-line capabilities: As often collectors might not have sufficient connectivity to connect to the server, the client application must also work off-line and for example store collected data on the mobile device until connectivity is available or data can be backup through a different connectivity such as USB and Bluetooth.

Rendering and storing media files: : As a part of a data collection process, different data types might have to be collected. The client must then be able to capture, store and display video, audio, text, geo-coordinates and so on.

Tight budgets: Any solution that such system might have to adopt should not exceed the cost of what a typical project can afford.

Among all MDCSs available today, there is none with a complete comprehensive secure solution for the data handled either by the mobile client or sent and stored on the server, wherever it might be processed. We will discuss what solutions are in place and whether they are adequate or not, but first we identify the critical security aspects in MDCSs. Most of them are standard security requirements, but nevertheless, are often neglected.

A good place to start is the OWASP Top Ten Mobile Risks [22], where the ten most common security risks for mobile devices and corresponding server side infrastructure are classified. In particular we focus on the three of them we have worked on, which are in the top 5: Insecure Data Storage, Insufficient Transport Layer Protection and Poor Authorization and Authentication. Besides these general security concerns, there are others that are more specific to MDCSs: for instance Data recovery and Inter-process communication.
Here is a brief description of the different security aspects we will consider:

Insecure Data Storage: Insecure data storage is probably the most concerning aspect of mobile devices, especially when a large amount of health related data are collected. The reason is that, unlike desktop computers, mobile devices are much more likely to be lost or stolen, or to be readily available in a short period of time if left unattended. It is enough that most application data is stored unprotected on the memory card, which can easily be taken out of the phone and read without problems on another device. This is why it is a necessity to always encrypt sensitive data before storing them locally on a mobile device. With encryption, however, come also a lot of other unavoidable problems. How to generate strong cryptographic keys? How to store them in a secure way? Which algorithm to use? Does the device offer adequate support? Similar problems also arise on the server side, at least when the server is not under direct control, like for example in the cloud case.

Insufficient Transport Layer Protection: Protecting the data in transit from the mobile devices to the server, and vice-versa, is another natural security requirement. When using wireless communication, eavesdropping the data traffic is not a difficult task, and if no encryption is in place, sensitive information can be easily stolen. However, only because the communication to the first wireless hotspot or router is encrypted, it does not mean that the entire communication up to the server is, and end-to-end encryption should be adopted. The most widespread solution to this problem is usually HTTPS, but we will argue later that better solutions might exist for MDCSs.

Poor Authorisation and Authentication: An authentication is a major problem both on the client and on the server side. Especially considering that the same phones might be shared among collectors, there should be mechanisms that guarantee that collectors only have access to their data, and this can be done only if some form of authentication and access control is in place on the client application. A similar problem is present when a user wants to transfer data from or to the server. Then it is essential to have mutual authentication, where the user knows that the server is the real one, and the server knows the identity of the user to enforce the right permissions. There exist various authentication protocols that allow for mutual authentication, but often the main problem is the generated traffic and the definition of a root of trust. For example, if we use a certificate to authenticate the server, we have to trust the certificate. This can be done by either buying one from a well known Certificate Authority or by provisioning it to the user through

[22] OWASP Top Ten Mobile Security Risks, [Last Accessed: January 2015], `https://www.owasp.org/index.php/Projects/OWASP_Mobile_Security_Project_-_Top_Ten_Mobile_Risks`

a secure side-channel. The secure side-channel becomes then the new problem: how do we know the side-channel is trusted? We will discuss this issue later in application provisioning chapter 7. Similar problems might exist when generating and distributing user accounts.

Data recovery: When data is going to be encrypted, it is important to have some recovery solution in case the collectors or even the project administrators lose their encryption keys or passwords. At the same time we do not want to have a recovery mechanism that can be easily exploited by an attacker or unauthorized entities.

Process to Process Communication and Separation : Typically, when a collector needs to fill a form, it is not only text that is entered, but also GPS-coordinates, video clips, pictures and so on. On modern smartphones platform such as as Android, this can be done by any application by simply accessing the existing hardware on the phone (like the camera) through public APIs offered by the OS. The problem is that often the data captured in this way is not under the direct control of the application, which only gets a copy or a link to it, and it is therefore very difficult to secure it or make sure it is deleted from the phone memory.

2.4 Mobile Platforms Security Reviews

Java ME and Android based MDCS are the focus of this work. In this section, we briefly describe the security model for Java ME and Android platforms.

2.4.1 Java Micro Edition (Java ME) and its Security Model

Java Micro-Edition Connected Limited Device Configuration (Java ME CLDC) was identified as the platform of choice to run mobile applications on mobile devices which are usually resource-constrained (i.e. cell phones, set-top boxes, etc.). Some of the major reasons that made Java as the defacto standard were its security, portability and network support features. Java plays a central role with respect to security of mobile devices because of its layered application on mobile devices.

Java Platform Security Architecture which defines the security features of Java Standard and Enterprise Edition provides a comprehensive security API to effectively addresses most of security concerns in the application but this security model is not fully deployed in Java ME CLDC platform due to the innate nature of mobile devices that constrains the resource that can be allocated for security design implementations. Even some important features from main line Java has been either disabled or implemented in lightweight manner which is less secure.

Debbabi, M. et al [28] makes an interesting comparison between Java Standard Edition (Java SE) and Java ME with respect to their security model namely - permissions, protection domains, security manager and security policy. For instance in Java SE permissions are applied on access to resources such as File, Property, Runtime, etc ... but in Java ME permission are applied only on access to connectivity protocols and push functionalities such as SMS, Bluetooth and MMS. When it comes to the protection domains, in Java SE a protection domain is created by instantiating the *java.security.ProtectionDomain* class and it's an object of this class that specifies a set of permissions. Whereas in Java ME it is only defined in the security policy which is not open to users for manipulation and there is no way by which the user can add new protection domains. Even on the security policy Java SE allows static and dynamic modification but Java ME allows only static modification. However, on various devices the policy configuration file is not open to users which makes even static modification difficult. The last parameter the Debbabi, M. et al paper [28] considers for comparison between Java SE and Java ME is the Security manager. Same like the protection domain, in Java SE it's a class named *java.lang.SecurityManager* that represents the implementation and in Java ME, it's up to the original equipment manufacturer(OEMs) to provide an implementation. In addition, in Java SE AccessController can be used to analyze the context at any moment and decide whether or not to grant a requested permission which makes it flexible enough to address issues related to access control mechanisms and decisions. For more details on Java ME and comparisons, we recommend readers to read the Debbabi, M. et al paper [28].

The Java ME platform provides a digital signature verification feature against with installed certificate authority (CA) root certificates. Typically, Java ME applications consist of two important file: a Java Archive (JAR) file and a Java Application Descriptor (JAD). The JAR file contains the main application, and the JAD file describes JAR file using a set of attributes. The application can be installed by sending the JAR file directly to the mobile phone, through over-the-air (OTA), Bluetooth, WiFi or cable, or it can be installed through the JAD file. In the last case, the JAD file will have to contain some mandatory attributes that must match with those in the manifest file contained in the JAR, plus optional and custom attributes. Among the required attributes is the URL of the JAR, so that the phone can download it automatically by generating a request and sending it to the specified URL in the JAD. The application owner can sign the JAR file with a code signing certificate, and add the signature as an attribute in the JAD file. This allows to verify that the Java Archive (JAR) file downloaded is indeed the same indicated in the JAD file, and that the entity distributing it, is a trusted one. Moreover, code signing protects the application by allowing only signed software to update installed application, so that it cannot be tampered with after installation. Thus, signing the JAR file containing the client application, is a necessary condition to guarantee any kind of security. Therefore, code signing is an important element of the secure software delivery model but it is not a complete solution.

2.4.2 Android Security Overview

Java ME addressed critical security needs on feature phones but there were some remaining challenges such as:

1. Lack of secure key storage
2. Lack of built-in crypto library
3. Restriction on system resources access
4. Usability issues – user login /password strength

The advent and proliferation of smartphones brought an opportunity to develop advanced, flexible, and secure applications. According to Gartner report[23], among several mobile platforms in the market such as iOS, Windows phone, Blackberry, Ubuntu Mobile[24],and Firefox OS[25], Android leads the market by a wide margin. Android is an open source project developed by the Open Handset Alliance and Google and available since 2008. Android is an application execution environment for mobile devices that includes an operating system, application framework, and core applications [117, 126]. For device drivers, memory management, process management and networking Android uses a software stack built on the Linux kernel.
With the move from feature phones to smartphones new functionalities and needs were inevitable such as the requirement for new security architecture; cloud storage became accessible to mobile devices; multiple apps interaction made possible instead of single app. Due to these advancements on mobile devices, ultimately a client-server secure solution compliant with HIPAA requirements on electronic records to support the following scenarios were needed:

- Multiple users per phone
- Multiple phones per user
- Offline data collection
- Lightweight, efficient, and cost-free secure data transmission
- Seamless Integration with zero or few lines of code.

A secure solution compliant to all such requirements might be hard to achieve but android is claimed to address most of the security challenges in Java ME and any security issues the might arise due to the added new functionalities. The major security mechanisms introduced in android are:

- Application Permissions
- Component Encapsulation
- Digital Signing of Applications

Application Permissions: it enforces restrictions on specific operations [117] that an application can perform. With more than 100 built in permissions applications can control a range of operations on smart phones. Applications need to explicitly request for a permission but as permissions have associated protection levels the system makes some checks on the application's signature or users' approval depending the level of protection.
Component Encapsulation: applications can encapsulate their components within their content to prevent access by other application unless otherwise the application enables the "exported" property of the component [117]. Even in those accessible components application permissions take effect to control the access level.

[23]Gartner Mobile Platform Marketshare, [Last Accessed: January 2015], `http://www.gartner.com/newsroom/id/2944819`
[24]Ubuntu Mobile, [Last Accessed: January 2015], `http://www.ubuntu.com/phone`
[25]Firefox OS, [Last Accessed: January 2015], `https://www.mozilla.org/en-US/firefox/os/2.0/`

Digital Signing of Applications: In Android platform there is a mandatory requirement to digital sign apps (.apk files) prior to installation on mobile devices. The embedded certificate can be self-signed and there is no need for a Certificate Authority. App signing on Android is an effective solution to :

- Ensure the authenticity of the author on the first install
- Ensure the authenticity of the author on updates
- Establish trust relationship among apps signed with the same key (share permissions, UID, process). Each process in Android has its own sandboxed address space, typically running under a unique user ID which is assigned by the kernel and used to enforce restriction on accessing resources, services, and communicating with other applications.

Android Inter-process communication (Android IPC) is a framework that allows data exchange across processes. The IPC is used for passing messages, file descriptors, synchronization signals, remote procedure call(RPC), and so on.

2.5 Summary

mHealth focuses on mobile devices as the primary tool used to deliver health services and disseminate information to different stakeholders. Among many mHealth services, remote mobile data collection (MDCS) allows the collection and transmission of data from remote geographical locations to data storage repositories through wireless or cellular network. MDCSs are a combination of a client application running on mobile devices, wireless infrastructures, and remotely accessible server databases. Most of the existing systems share common principles and guidelines to collect data remotely. As also reported in [14], MDCSs have been mostly aimed at projects with tight budgets, deployed in developing countries with sparsely populated areas, where low data rate and intermittent connectivity exist. This resulted in the development of light mobile clients targeting the so called "feature phones", i.e., cheap phones with only basic functionality, but able to run simple Java applications. It is not surprising that most MDC Systems until a couple of years ago would provide only Java ME based clients which could run on phones such as Nokia 2330c with very low specifications, i.e., a 4.7MHz processor, 4MB RAM, and 128KB persistent storage. At this point, security was not considered in any of MDCSs. In fact, the lack of support for cryptographic libraries in these feature phones, their limited computing capability and memory resources, and least privileges access for third-party application to the phone, made it extremely difficult to adopt the most basic and standard security solutions. Even using the HTTPS [109] protocol for secure communication could be a big problem because of the lack of a standard list of pre-installed Certificate Authorities (CAs) on the phones, the cost of these certificates and the poor implementation of the protocol itself on many devices [122].

However, in the past few years we observed a dramatic change in the mobile operating systems (Mobile OS) market share. Java ME, that dominated the scene for decades, has left the competitive era to the newborn Android [26] and iOS [27]. We also saw that novel mobile OSes such as Ubuntu Mobile [28] and Firefox OS [29] are trying to penetrate the market by targeting the emerging market. This OS explosion has brought new opportunities and challenges to the data collection systems community. The boost in device computation performance, longer battery life, larger screen, and natural way of communicating with the device are some of the benefits. On the other hand, cost of the device and maintaining multiple code base targeting different OS have been the major challenges. Besides, the skills needed to customize the codebase written in native languages require high-level computing competence that is hard to find in low-and-middle income countries. New security challenges are, of course, also another problem.

The server side of MDCSs has also undergone some evolution. Although the technologies used in the implementation might not have changed so drastically as the mobile counterpart, the way of deploying and offering services on the web has changed. It should suffice to mention cloud-based services.

In this chapter, we mainly focused on the background of the dissertation and the security challenges faced by MDCSs specifically when low resources settings and low-budget applied. In the next few chapters, we will discuss how we approach the security challenges, design a secure solution using Android and Java ME based MDCS and cloud-based servers are employed. However, some fundamental security issues are conceptually identical no matter the underlying system, and most of the proposed solutions can be applied in several contexts and Mobile platforms. We also discuss the solutions we designed for older Java ME based devices, and how they are still relevant.

[26] Android, [Last Accessed: May 2015], `http://www.android.com`
[27] Apple iOS, [Last Accessed: February 2015], `http://www.apple.com/ios/`
[28] Ubuntu Mobile, [Last Accessed: May 2015], `http://www.ubuntu.com/phone`
[29] Firefox OS, [Last Accessed: January 2015], `http://www.mozilla.org/en-US/firefox/os/`

3 The SecureMDC Framework Overview

This chapter presents a security framework for mobile data collection systems (MDCS) in the healthcare domain, which is the main contribution of this dissertation. The security framework is called SecureMDC and aims to secure existing MDCS by considering the functional and security requirements discussed in chapter 2 and the next few chapters. Primarily, the framework provides services such as user management, secure storage management, secure data transmission, account and data recovery, and application provisioning. The framework is designed to make the integration process with the existing MDCS as easy and transparent as possible. In the next section, we present a list of design choices and criteria that are considered in the SecureMDC framework. The design criteria are gathered from our MDCS review experiences, dialog with the MDCS communities, and API design best practices[1].

3.1 SecureMDC Design Criteria

Before starting any work, it is imperative that some goal or ground rules be laid down regarding what we are trying to accomplish. To be more precise, we identified the following criteria for the SecureMDC framework to take into consideration.

3.1.1 Generic, Ease of Use, and Transparent Design

When having to incorporate security into an existing system, it is important that the integration process be as seamless as possible from an implementation and usability point of view, and maintenance costs do not increase significantly. If adding security meant to rewrite large pieces of the code base, or radically change interfaces and data flow, not many developers and users would be willing to put up with it. Hence, a security solution for MDCS should be flexible enough to be integrated into these systems with

[1]Joshua Bloch. How to design a good api and why it matters, [Last Accessed: May 2015], `http://lcsd05.cs.tamu.edu/slides/keynote.pdf`

as little effort as possible. This can translate into a trade-off between ad-hoc solutions that might be especially efficient for specific clients, and more generic approaches that guarantee wider compatibility.

This is the most important criteria the SecureMDC is designed to address. At the time of this writing, there are several MDCS in use as discussed in chapter 2, and many of them lack proper security. If we are to be successful in getting any of these to adopt the SecureMDC framework into their applications, integration into an existing system needs to be as hassle free as possible. Trying to address this issue, we have focused on making the framework design as transparent as possible. This means that a programmer using the framework to the least possible degree needs to know anything about the implementation of the underlying framework. Ideally any programmer with Android or Java ME experience should be able to use the framework without having security background.

3.1.2 Follow standards

Interoperability between MDCS in a resource-constrained environment is a major factor for the success of MDCS. Our security solution should not introduce a barrier that hinder different systems from interactions. Instead, it should follow existing standards and protocols to facilitate smooth integration between systems.

3.1.3 Functionality Decoupling and Flexibility

The full framework provides more than one functionality. Since each of these services cover different areas, they should be as decoupled as possible, preferably entirely independent of each other. This would make the framework very flexible since a programmer could choose to use only the functionality that is needed, and not being forced to add unwanted functionalities that have implications for performance and maintenance cost.

3.1.4 Lightweight and Low Cost

Most of the challenges faced when trying to secure MDCS come from the fact that many projects using these systems run on very low budgets. Hence, the security framework should find an acceptable compromise between cryptographic strength and available computational power and battery usage, reduced overhead and maintenance cost.

3.1.5 Battery consumption

A very important aspect of remote data collection, is that mobile devices should have long battery life, since, in some remote regions, electricity might be a scarce resource, and it might not always be possible to charge them whenever needed. Therefore, if the security framework consumes too many resources and shortens the battery life considerably, it might render mobile data collection quite simply an unfeasible alternative. In this work computation intensive operations are considered carefully for their impact on performance, and therefore on battery life.

3.1.6 Usability

Data collectors are the one who interact with the people and collect data. The literacy level of these data collectors and understanding of how the system works or how data is gathered using mobile devices varies considerably. Collections can be conducted by data collectors with low educational background or graduates such as clinician and community health workers. Therefore, usability and simplicity are the most important requirements from a users point of view. Consequently, the security framework must find a balanced solution between security and usability [76].

3.2 Secure Solution Integration Approaches

Currently, there are some mobile data collection tools that are developed as a single fully-fledged mobile app. This means that multiple functionalities and tasks are consolidated into a single application and shipped to users. As an alternative approach, the ODK team develop multiple apps separately, and each app specialized in a certain functionality. In the later approach, applications are expected to interact each other to share their functionalities. For instance, ODK toolkit consists of mobile apps such as Survey, Tables, Scan, Submit, and Sensor as described in chapter 2, section 2.2.1. When the Tables app, which specialized in data presentation, wants to capture data using the form definition, it uses Survey app. The ODK team argues that this approach can facilitate adding new functionalities to the ODK toolkit without touching other applications [16]. However, both approaches may have pros and cons if we compare them with issues such as maintaining single code base versus many, performance and memory footprint, provisioning multiple apps against the single app, and security. However, among all, a working secure solution that fits both uses case is the primary concern of this work. Thus, to incorporate both use case scenarios, we identified the following two design approaches.

1. **Distributed Secure Architecture**: This architecture aims to meet the MDCS requirements and design criteria by developing a secure solution for individual MDCS client. Thus, each client app should provide secure authentication and storage management by its own. The design approach has two drawbacks. First, each app has to maintain the key store that is needed for user authentication and secure storage. Second, the secure solution might be tightly integrated into the existing app that lowers the transparency of the integration. Third, each client should incorporate the library project in its APK, and this maximizes the memory footprint for the app. Besides, code duplication may occur particularly when different applications interact each other for services sharing, for instance in ODK client apps. Communication and Sharing data between apps might be insecure as well.

2. **Centralized Secure Architecture**: On the contrary, the centralized, secure architecture aims to provide security services as a standalone secure app that acts in the middle of multiple apps. The solution is completely transparent and does not require refactoring on the existing client apps for integration. Apps share the same data storage schemes.

The secure architecture choice differs from MDCS to MDCS, but the main objective of this work is to meet MDCS requirements and design criteria. We found the centralized, secure architecture more flexible, less maintenance, and allows data sharing among multiple apps. As a result, we chose the centralized approach for security framework discussed shortly.

The following four concrete integration approaches can be applied in either distributed or centralized, secure architecture. The secure API can be integrated in four different ways with different degree of control and responsibility for the programmer. If the programmer is knowledgeable when it comes to security, (s)he can control how users are authenticated, where and how data are stored and managed, and how the secure communication channel is established. If the programmer is not familiar with security design then this task can be taken care of by a SecureMDC framework discussed later in this chapter. The approaches will go from programmer control to the framework control.

3.2.0.1 Full Programmer Control

With this approach, the MDC application accesses the security API through the *Crypto* interface with default implementation as shown in the listing 3.1. The *Crypto* interface abstracts the underlying implementation complexity and provides services such as encryption, decryption, message digest, data signing, key derivation function and so on. The *Crypto* library is designed and implemented to accept caller input, process it, and return the result back to the caller immediately. In other words, the *Crypto* library does not store or manage any keys or data related to MDCS Client. Also, the programmer is responsible to initialize the *Crypto* library directly; this means that the MDCS client will manage user credentials, tokens, data protection keys, and any additional parameters used. From a programmer perspective, this gives the programmer a full control over how user credentials and data is stored and managed during a transaction. From a security API provider perspective, there is less work and bookkeeping needed as key management, where and how the data stored, or how the client communicates with the server, is managed by the application. Thus, the challenging tasks are left to the MDCS client or to the programmer to manage. Furthermore, the programmer has to take care of keeping any keys used safe both from a potential attacker and from being lost.

Figure 3.1 shows how the MDCS client added the *crypto* API. As previously discussed, the *crypto* API provides security features with a default implementation that the programmer can leverage to build security components including user authentication, secure data storage, and secure data transmission. It is also programmer's responsibility of managing user credentials and keys.

We summarized the pros and cons of this approach as follows:

Pros

- The programmer has full control of the code and when and what is done.
- The *crypto* API enforces no restrictions, there are no limitations to what other items might be put in the stream.

```
public interface Crypto {
//"public" and "private" keys are for asymmetric and "secret" key is for
  ↪  symmetric algorithms.
enum KeyType {"public", "private", "secret"};
//recognized key values: "encrypt", "decrypt", "sign", "verify", "derivekey",
  ↪  "wrapKey", and "unwrapKey".
enum KeyUsage {"encrypt", "decrypt", "sign", "verify", "deriveKey", "wrapKey",
  ↪  "unwrapKey"};
// Recognized key format values are:
enum KeyFormat {"raw", "spki", "pkcs8", "jwk"};
// used to get the security provider used for the implementation.
public String getProvider();
//encrypt data using the specified AlgorithmIdentifier with the supplied
  ↪  CryptoKey.
public byte[] encrypt(AlgorithmIdentifier algorithm, CryptoKey key, BufferSource
  ↪  data);
//decrypt data using the specified AlgorithmIdentifier with the supplied
  ↪  CryptoKey.
public byte[] decrypt(AlgorithmIdentifier algorithm, CryptoKey key, BufferSource
  ↪  data);
//sign data using the specified AlgorithmIdentifier with the supplied CryptoKey.
public byte[] sign(AlgorithmIdentifier algorithm, CryptoKey key, BufferSource
  ↪  data);
//verify a signature using the specified AlgorithmIdentifier with the supplied
  ↪  CryptoKey.
public boolean verify(AlgorithmIdentifier algorithm, CryptoKey key, BufferSource
  ↪  signature, BufferSource data);
//Make digest of a data using the specified AlgorithmIdentifier.
public byte[] digest(AlgorithmIdentifier algorithm, BufferSource data);
//generate cryptographically key using the specified AlgorithmIdentifier with a
  ↪  particular key usage.
public byte[] generateKey(AlgorithmIdentifier algorithm, KeyUsage keyUsages );
//drive cryptographically key using the specified AlgorithmIdentifier,  with a
  ↪  particular key usage.
public byte[] deriveKey(AlgorithmIdentifier algorithm, KeyParameters parameters,
  ↪  , KeyUsage keyUsages);
}
```

Listing 3.1: Overview of The Crypto interface.

- The programmer could design his or her own systems for authenticating user, downloading, storing, and uploading users data to the server.
- It is easy to manage and maintain the *crypto* API
- The *crypto* API is generic and agnostic to MDCS and can be used in multiple MDCS applications.

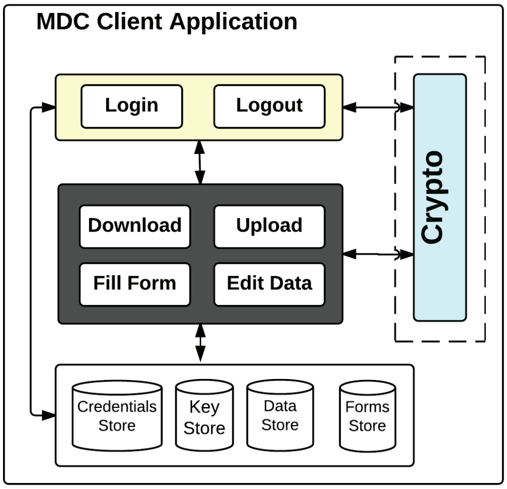


Figure 3.1: Full Programmer or MDCS Client Control

Cons

- It is up to the programmer to create the security components such as user authentication, secure storage, and secure communication.
- Programmer is expected to understand security concerns and risks. It is easy to make mistakes.
- Requires a high learning curve.
- Application to application communication may get complicated and insecure during interprocess communication (IPC)[2].

Even if this approach provides flexibility to the programmer, the dialog we had with programmers' in the MDCS communities indicates that the programmers' are more focused towards the functionality aspect of the application. So they prefer an out-of-box secure solution or a solution that requires fewer resources, zero or minimum security knowledge, and seamless integration approaches. Hence, it was required to look for other alternative approaches.

3.2.0.2 Partial Programmer Control

With this approach, the secure API takes the responsibility of handling user-specific credentials and keys. The API would more or less aggregate the calls on *crypto* library shown in the listing 3.1 to create a key and account management components and expose it to the MDCS client through interfaces.

[2]Android IPC, [Last Accessed: June 2015], http://developer.android.com/guide/components/aidl.html

Figure 3.2 shows how the MDCS client incorporates the secure API with the crypto library. The secure API provides the key and account manager that securely store and manage user credentials and keys on the mobile device. The MDCS client interacts with the key and account manager through exposed interfaces. However, the secure API still expect the programmer to implement the login user interface that retrieves user credential and pass it to the security API account manager. Internally, the key manager uses user's password to protect user's keys and credentials. When the user attempts to login, the key and account manager uses user credential to authenticate and unlock credentials and keys store. If the login is a success, the secure API returns user key and token used to protect user's data and re-authentication respectively. The programmer is required to initialize necessary objects for data encryption, decryption, and secure communication. For data recovery purpose, the programmer should also have a contingency plan for backing up the keys used to secure storage if the user forget his or her password. Failing to do so could result in not being able to decrypt data.

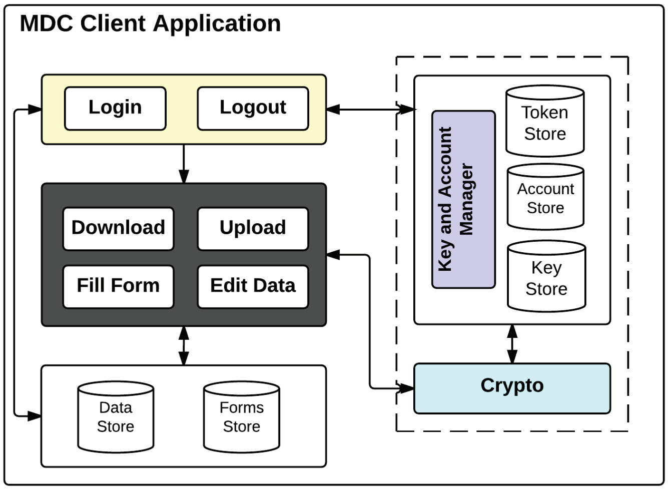


Figure 3.2: Partial Programmer or MDCS Client Control

This approach handles keys and credentials storage securely and leaves the programmer with a decent amount of control. Once the keys store is unlocked, and the storage and communication objects are initialized, the MDCS client can use these objects to encrypt and store forms and data in a secure manner. The programmer can still decide where and how to store the encrypted data.
The pros and cons of this approach are summarized as follows:

Pros

- User credential and keys are securely stored.
- Except the key and credential storage, the programmer has full control on how to encrypt and store data or to establish a secure tunnel.

- The *crypto* API still available to the programmer and enforces no restrictions.
- The programmer has control of what goes on while the data is being fetched; this means that giving feedback to the end user would be simpler.
- May work on multiple apps scenarios with reasonable complexity.

Cons

- Programmer is expected to understand security concerns and risks.
- Requires a high learning curve.
- Integration into existing systems could prove difficult or require some changes.

3.2.0.3 Transparent Design

A third approach to the integration challenge is to make a compromise between the two previous solutions. With this approach, we wrap the secure API around the public API of storage and communications management of a given platform such as Java ME and Android. The programmer does not need any special knowledge of the underlying security complexity. With few lines of code changes, the programmer can use the secure API through standard public API to store, retrieve data in the storage, and establish a communication channel in a secure manner. The secure API manages encryption, decryption, key management, authentication, and other security services transparently.

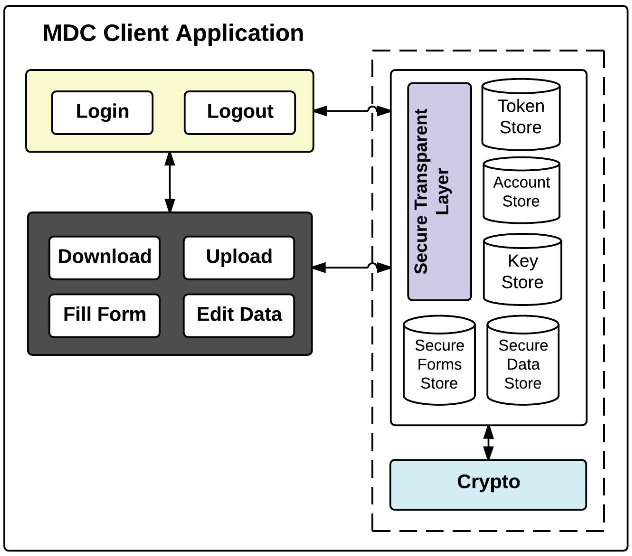


Figure 3.3: Transparent Design

Here are some of pros and cons of transparent design approach:

Pros

- Very easy to integrate into existing systems.
- Programmer needs no knowledge about underlying security concerns.
- The secure API puts no restrictions on the programmer compared to normal Java ME or Android.

Cons

- There is only one binary storage in Java ME and the Java ME platform provides quite a few public APIs for managing data storage and transmission. Thus, we managed creating transparent secure storage and communication by wrapping these public APIs securely. However, things have changed with the introduction of Android and other smartphone platforms. There are many ways of storing data such as SQLite database, flat files, object storage, and binary. The platform provides public API for manipulating data in this storage. Similarly, a range of native and third party API are available to communicate with the server including HTTP API from Apache, Java, or Android itself. As a result, the MDCS use different API to accomplish similar tasks. Therefore, if we start wrapping the secure API around these public API, it may require a significant amount of resources for development, and maintenance. We may conclude that this might not work a platform like Android where it is possible to use multiple public APIs.

The above three integration approaches apply to both Java ME and Android platforms. However, the fourth approach discussed below is applicable only to the Android platform. The strict application sandbox enforcement in Java ME and lack of API for accessing resources or communicating with other apps in Java ME makes the next proposed approach infeasible. In contrast, Android leverages Interprocess Communication (IPC) extensively and offers public API to third party apps to interact each other. In the next section, we introduce a modular, secure solution that handles providing security services to MDCS systems through IPC framework.

3.2.0.4 Full Secure Framework Control (SecureMDC Framework)

In this method of integration, the SecureMDC framework handles providing all important security services to MDCS. Once the MDCS client app launches, the screen control would be given to the SecureMDC framework. Depending on the configuration, the framework may prompt users to authenticate to the server on the first run, ask users to register, or just log in. Once the registration or authentication is successful, control is given back to the MDCS client to perform activities such as form downloading, form filling, data editing, and submission to MDCS server. All these activities are performed with the help of the SecureMDC framework.

This method makes the programmer completely free from any security concerns. The framework takes care of server authentication, user registration, local authentication, account recovery, data protection at rest and during transmission, and initializing the different components with the appropriate keys. The security framework can be packaged as standalone application and uses standard IPC calls to exchange data with the MDCS client or can be packaged as a library project and added to the main MDCS client. We present this design approach thoroughly in the next few consecutive chapters. Before delving into the details, here are some of pros and cons of the approach:

Pros

- Easy to integrate into existing systems.
- Programmer needs minimum knowledge about underlying security concerns.
- Handles both the single fully-fledged app and multiple apps scenarios.
- Single and simple data storage scheme for multiple apps.
- Single sign-on is possible.

Cons

- Data cleanup process must be carefully designed, otherwise, data leakage or loss may occur.
- MDCS may be required to pass some form description information to the SecureMDC framework when performance optimization is needed. We discuss this subject thoroughly in chapter 5.

An overview of interprocess communication between these apps is shown in figure 3.4. In the next section, we discuss the interaction between MDCS client and the SecureMDC framework in depth and provide our analysis on risk and challenges involved with this approach.

3.3 The SecureMDC Framework Overview

The SecureMDC framework is a set of modular security features that are designed to fulfill MDCS requirements and design criteria. The framework abstracts the underlying secure implementation complexity and provides simple interfaces for interaction. The framework is designed to be secure by default, but it is also flexible, customizable, and able to adjust itself to different MDCS security settings. We first developed the framework for Java ME based MDCS and later we ported to Android based MDCS. Also, we considered the current state of arts including cloud-based MDCS, application provisioning, and standard secure communication mechanisms. Figure 3.5 shows standard data protection mechanisms that are already developed and integrated into our partner MDCS projects. Primarily, the framework attempts to provide a comprehensive secure solution for user authentication, secure mobile and cloud storage, secure communication, application provisioning, and account and data recovery methods. The remaining items in the figure 3.5 including GCM agent (Google Cloud Messaging), session management, and access control mechanisms are research in progress, and the findings will be presented in future work. In this section, we describe the framework at a high level, giving an overview of its architecture, design principles, the interaction among its internal components, and some of the ongoing development.

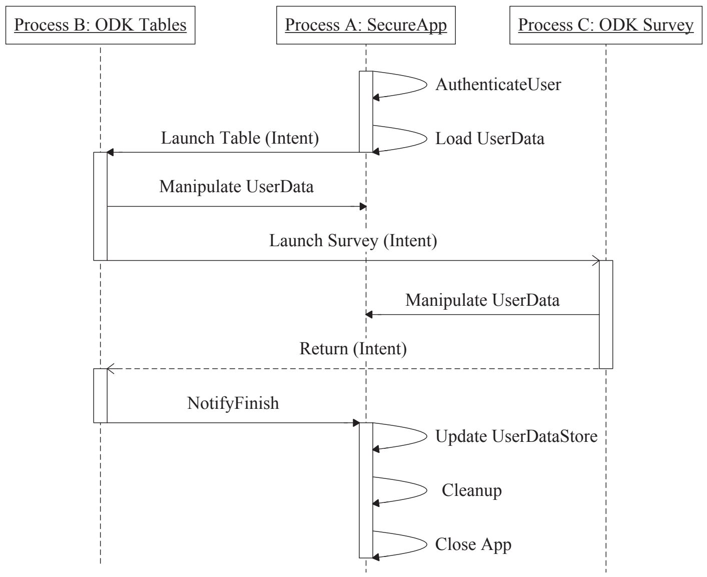


Figure 3.4: SecureMDC and MDCS client Communication Flow Diagram

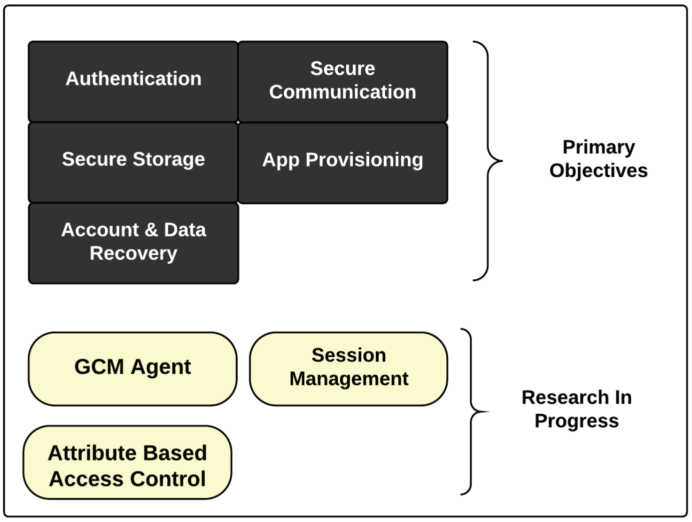


Figure 3.5: secureMDC Working Items

3.3.1 SecureMDC Framework Architecture

When we design the framework, the modular and centralized design approach is followed to fulfill the design criteria listed in the section 3.1. The SecureMDC framework comprises three main modules, and each module provide specific security service. Figure 3.6 shows the modules and each module functionality is explained as follows:

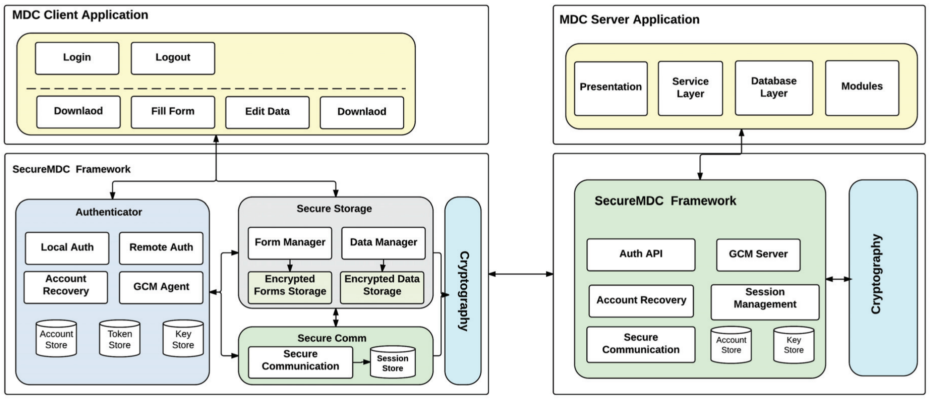


Figure 3.6: SecureMDC General Architecture

1. **Authenticator**: is a security module that handles user authentication on a mobile device and a remotely located server and account recovery. It exposes the authentication services through simple interfaces with a default concrete implementation. The MDCS client delegates the user authentication to this particular module. Thus, any attempts of accessing the MDCS client leads to the invocation of the Authenticator module. The Authenticator is flexible and can be configured to provide additional features such as device authentication and single sign-on. The main reason for having the module is the requirement that a phone may be shared among multiple collectors who should not have access to each others collected data.

2. **Secure Storage**: is a security module responsible for MDCS application resources protection and management on the mobile device. The secure storage is accessible via simple interfaces with a default concrete implementation that handles encryption, decryption, cleaning residual data after user logged out, and a recovery plan when the application crashes, or the battery dies.

3. **Secure Communication**: is a security module responsible for establishing a secure tunnel between client and server. The Hypertext Transfer Protocol Secure (HTTPS) is a widely used protocol for securing HTTP messages [108]. Many of reviewed MDCS provides HTTPS features. However, in some of our previous work [44, 73] we argued why HTTPS cannot always be considered as the best solution for secure communication in low-budget data collection projects. The

reasons were mainly three: bad support/implementation in older phones, criticisms towards the Certificate Authority Trust Model (see for instance [112]) and the cost of the certificates. Then, we looked for other protocols that are not dependent on certificates and realised that a password-based approach would be best. The Secure Remote Password Protocol (SRP) is one of the most reliable mutual authentication and key exchange algorithms. The SRP is a Password Authenticated Key Exchange (PAKE) protocol based on a pre-shared password for mutual authentication and key exchange, The SRP is standardised in RFC 2945[139]. It allows mutual authentication and secure key exchange, while being resistant to on-line brute-force and Man-in-the-Middle attacks (MITM). Besides, it can be used in combination with TLS [25] to create a secure transport layer without the need of certificates or new protocols. We will thoroughly discuss and present a comprehensive secure communication solution in chapter 6.

The SecureMDC framework is designed to be integrated with an existing MDCS client and server and securely encapsulate the messages between them. The encapsulation or wrapping process help us to make a transparent integration, and no prior knowledge is required to interact with the security framework. In the next few chapters, each of the modules are examined separately and proposed adequate security solution and described thoroughly. Before that, we briefly present at high-level how the modules work and interaction with the MDC application, and some ideas for future development.

3.3.2 Framework Internal Interaction

Application to application communication was not possible when we dealt with the Java ME platform. The Java ME sandbox model is very restrictive on either system level API or allowing an app to app communication. Android provides a way to escape from application sandbox and communicate with another application running on a different process. Android also provides a comprehensive permissions model that protects applications from malicious code that tries to misuse applications internal communication channel. Signature based permission is one of the key components of Android permission model. Each application is signed by a developer, and can be updated only by the same developer, i.e., by other applications signed with the same key. Similarly, applications signed with the same key will be assigned the same Linux Kernel User Id, and will be able to communicate with each other through the IPC we mentioned earlier and share permissions and resources. If the reader is interested to know more about Android's application signing and verification, David B. et al. [10] presented a thorough analysis on the Android signing and verification architecture.

However, this means that any MDC applications that are interested in leveraging the security framework are required to sign the security framework and the MDC application with the same signing key, which might be a challenging task. Fortunately, Android provides an easy way to incorporate project like the security framework into the main application as a *library project*. This way the SecureMDC framework can be packaged as a library project and signed using the MDC app key. The application and framework will still run as two different processes and communicate through the IPC even though it is called a "library". The downside is the tight integration that makes it difficult to guarantee the integrity of the framework and its maintenance. If the framework is updated, a new version of the application would have to be packaged and installed. Therefore, we believe that it would be best if the security framework and the MDC application could be signed each by their signing key, and run as separate applications. This is also the working assumption we will make for the rest of the dissertation. A discussion around this topic is given in Chapter 8 under future work.

3.3.3 Way forward

As we have already mentioned, the framework is constantly evolving to adapt itself to the new technologies that continuously emerge on the market, and to provide better security and usability. Lately, we began researching on one of push notification technology called Google Cloud Messaging (GCM) for Android for managing the client remotely. GCM is a service that allows the server and client to exchange small but important messages through Google infrastructure in a secure and automated manner. This service hold an important role in managing MDC clients remotely and providing services such as user revocation, data synchronization, and wiping data protection keys remotely. This is still research in progress, and we briefly present the technology and the use cases in chapter 5 under section 5.4.4.

Session Management is the other topics in our to-do-list. The SecureMDC framework has a cookie based session management in place but at upon this writing, there is an active research to replace the traditional cookie-based session management with a Java Script Notation (JSON) based web token (JWT) that is digitally signed using another standard called JSON web Signature (JWS). A draft is submitted to the Internet Engineering Task Force (IETF) [3] and we have been closely looking the solution. Currently, we are actively working to adopt the solution, and we briefly introduce the technology and importance to our work in the later section.

Role Based Access Control (RBAC) with its pros and cons is the widely used access control mechanisms for most MDC servers. Attribute-based Access Control (ABAC) is the fine-grained access control mechanisms that provides resources access based on a combination of several attributes. Unlike RBAC, ABAC flexibility and manageability, we begin a research to investigate the use of ABAC in MDC systems. The progress we made so far will be presented in the future.

[3]JSON Web Token (JWT) : `http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html`

4 SecureMDC Authenticator: A User Authentication Module for MDCS

This Chapter is based on the candidate's previously published work at the International Conference on Availability, Reliability and Security (ARES 2012). The full paper is available at [73].

4.1 Introduction

User authentication on mobile devices is not a trivial task. Any comprehensive authentication mechanism for MDCS should incorporate local and remote authentication. The word *"local"* refers to authenticating users on a mobile device without requiring server connectivity (i.e. offline mode), whereas the word *"remote"* refers to the usual client-server authentication with connectivity (i.e. online). Many of reviewed MDCS use weak remote authentication methods. Local authentication is either not available or implemented in an insecure manner. On the other hand, there is a growing need to have an easily integrable and usable authentication solution for MDCS. In this chapter, we thoroughly discuss one of the SecureMDC framework modules called *Authenticator*, as shown in figure 4.8. The Authenticator provides a comprehensive user authentication service both on offline and online.

The rest of the chapter is organized as follows. In section 4.2, we present functional and security requirements specific to user authentication on resource constrained MDC systems and perform risk analysis. Then, we discuss currently available solutions to address functional and security requirements in section 4.4. At the end, we present discussions and conclusions.

4.2 Requirements and Risk Analysis

After field visits and thorough discussion with our project partners including the ODK team, the openXdata team, and the mUzima team, we identified the following functional requirements which state what the system must do and non-functional requirements (Security Requirements) which constrain how the system must accomplish the functional requirements.

4.2.1 Functional Requirements

The functional requirements incorporates:

- **Multi-Users Per Device (Phone Sharing)**: The same mobile phone might be shared among multiple collectors while the same collector might use more than one phone. Thus, the client must support multiple users.

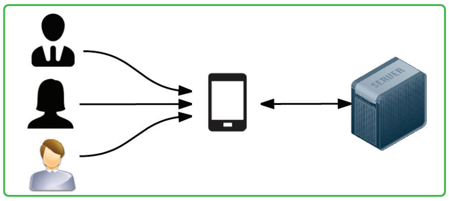


Figure 4.1: Multiple Users Per Device

- **Single User per Multiple Devices**: a collector uses more than one device at the same time.

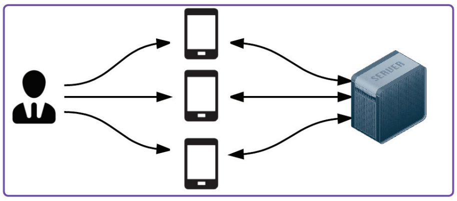


Figure 4.2: Single User per Multiple Devices

- **Offline capability**: As often collectors might not have sufficient connectivity to connect to the server, the client application must also work off-line and for example store collected data on the mobile device until connectivity is available or data can be backup through a different connectivity such as USB and Bluetooth.

- **Minimize Maintenance and Overhead Costs**: Any solution that such system might have to adopt should not exceed the cost of what a typical project can afford.

- **Ease of Integration**: Even with a system such as the secure framework that is made to cover all the requirements and limitations discussed so far, if it is hard to use or requires the MDCS to undergo large changes in order to be secured, few people would use it. As such, any system addressing security for MDCS needs to be easy to use and implement into existing systems for the developers, but also easy to use for the end users.

- **Usability**: many studies show that strong security enforcement alone does not provide the ultimate goal related to data protection. System usability is a crucial factor in particular to mobile data collection systems. In many deployments, the staff responsible for data collection may include people with a little education background and/or little experience with computers or use of mobile devices [121]. Therefore, the security framework should find a balanced, flexible and transparent solution that maximizes usability and minimizes security risks. [76]

4.2.2 Security Requirements

In addition to the functional requirements, the authentication service must satisfy the following set of security requirements.

- **Mutual Authentication**: When data are transferred electronically, sending and receiving parties will need to be authenticated each other.

- **Revocation**: the authentication service should incorporate procedures for withdrawing access rights when staff is no longer employed at the site.

- **Account Recovery**: the authentication service should provide account recovery mechanism if a user forget his/her password or username. The recovery mechanism does not compromise the system or users data.

- **Single Sign-On**: The growing numbers of mobile applications within a single ecosystem of mobile data gathering demands single sign-on service that provides end user to login once and use multiple services on the mobile device. Open Data Kit (ODK) can be a good example that needs single sign-on service for its multiple mobile apps.

4.3 Threat Model

The disclosure of personal health information can result in a severe damage in terms of reputation for the organization running the project, or even legal issues and fees. For the patients this can lead to a loss of privacy over some of the most personal data, the ones regarding their health conditions, but also in most cases their location and contact data. Apart from the collected data, also user credentials must be kept as secret as possible.

Leaked credentials not only can allow the impersonation of the user on the MDCS, but probably also on other systems, since people tend to re-use the same credentials [52].

On the mobile device, the most likely threats in a Remote Mobile Data Collection scenario are the following:

The data is lost or stolen : the collector may lose or damage the phone. As a result, data can be lost or accessible to an authorized user..

Malicious applications : a collector might download malicious apps on the mobile device that can interfere with the MDCS client and steal, read or hijack the collected data. The MDCS application might be replaced with a malicious version at download time [52].

Basic Authentication : If the collector credentials are not properly protected during the authentication process, they may be stolen with a Man-in-the-Middle Attack (MITM) or simply tapping the connection to the server, or intercepting the process performing the authentication on the device [52].

4.4 Available Solutions

Mobile platform vendors have recognized the risks related to the phone being lost or stolen, and they provide built-in security features to minimize the risks. Device authentication is one of the security services that the mobile platform provides. It is common practice to authenticate the user using single, two, or multiple authentication factors. These factors can be categorized as follows:

- **Something user knows**: such as password or pin code
- **Something user has**: such as a hardware token or a smartcard, or mobile devices
- **Something user are**: including Biometric characteristics such as fingerprint, which is started appearing in smartphone devices
- **Location and Time**: are gathered from the client and assist the server to decide based on user geographical location and time. Google and Facebook are some of the examples which use these attributes during user authentication.

The first two factors (something user knows and has) often provided by most mobile platforms, but the fourth attribute (Location/Time) is considered as best practice to allow or deny access of service/resource based on user geographical location and/or time. The third one, Biometric authentication, particularly fingerprint, is one of the key emerging technology on mobile devices. These and other authentication technologies are the first line defense on mobile device.Furthermore, mobile devices increasingly connected to the backend server, services, and resources. So credentials stored on the device may have the capability of compromising the server. Hence, a comprehensive authentication solution both on the device and the remote server is vital to the project. In the next section, we discuss local authentication and explore the built-in security features that Android platform offers for authenticating users on the mobile device. Later, we present weak and strong remote authentications methods.

4.4.1 Local Authentication

As we stated in the section 4.1, we refer local authentication as an offline authentication mechanism for a user before accessing the mobile app services and data. Some systems use server-side authentication to allow user have access to the mobile app on the device. This approach does not require to store some secret on the mobile device. However, there is always a redirect for user authentication to the server when the user tries to access the application, and this type of authentication requires connectivity. However, intermittent connectivity is one of the challenges in low income countries, and the server-side approach does not work in that context. Hence, local authentication is performed based on some secret stored on the device. The remaining challenge are to keep the secret secure and have a credential recovery mechanism in place. The first line of security is not to store the secret in clear, but rather salted and hashed, encrypted or similar. The other is to make it difficult to extract from the device such as hardware-based credential storage such as smartcard. Android and other similar smartphones platform provide a set of security features for protecting data on the device. However, many of these security features are not activated by default, in other word, the user must activate them individually. Next, some of the built-in local authentication features are presented.

4.4.1.1 PIN Codes and Password

PIN codes are the popular method of securing mobile devices. PIN codes are relatively easy to use and remember on mobile device. As of this writing, Android requires a minimum of 4 and maximum of 16 numeric digits when setting a new PIN code for screen locking. Internally, when the user set a new PIN, Android generate a random salt, concatenate with the user PIN and compute both MD5 and SHA-1 hash of the concatenated value. It is clear that why Android computes both MD5 and SHA-1 PIN hash value. The hashed PIN code is stored in a file that has system level protection, which means that no third-party applications can have read/write access to the file.

On the other hand, Android offers password-based authentication. The password provides wide varieties of character selection including lowercase, uppercase, numbers, and special characters. As a result, the entropy can be maximized [66]. But, one of the challenges is most of password policy and rules such as password size and mixture of characters in it are optimized for physical desktop keyboards [116]. So applying the same set of rules on the mobile device with virtual touch screen keyboard or small size physical keyboard might not work. For instance, typing numbers or special characters on the smartphone requires navigation to a second or third keyboard page [66]. But, one of the challenges is most of password policy and rules such as password size and mixture of characters. Generally, finding a balance between security and usability is a challenging task. From an implementation point of view, in Android, the password is implemented in the same way as PIN code as discussed above.

A brute-force attack is one of the potential attacks on either PIN or password authentication. Android tries to protect the device from online brute-force attack by enforcing 30 seconds delay after each five subsequent failed authentication attempts. [33].

4.4.1.2 Pattern

The idea behind pattern-based authentication mechanism is people are naturally good at remembering visual pattern than random string or numeric. It is 3x3 matrix (9 points. As a rule, user expected to select at least four points and each point is used only once. Each point on the matrix is assigned an index value, where 0 is top left and 8 bottom right [33]. Hence, the user selection is stored as a byte sequence. Because of the limited number of selections available (minimum 4, maximum 9 points) which result in low entropy, pattern authentication is more insecure than PIN or other authentication methods. In addition, since the pattern byte sequence is computed and stored under Android */data/system/gesture.key* as a simple user input hash (SHA-1) value, if an attacker able retrieve the file, s(he) can able to see the pattern. Furthermore, pattern authentication is vulnerable to the so-called *"smudge attack"* [3, 119]. This attack as a result of the capacitive touch screen that is designed to work by using user finger, and drawing the unlock pattern several times leaves a trace on the touch screen which can easily see with high probability using appropriate tools [9, 33]. For these reasons, pattern based authentication is categorized as weak.

4.4.1.3 Facial Recognition

Facial recognition is a biometric authentication method that Android platform offer since version 4.0 (a.k.a Ice Cream Sandwich) for unlocking the device using human face [100]. The security level of this method is categorized as a *weak* authentication mechanism. The two-dimensional (2D) face recognition introduced in Android 4.0, can easily be compromised by holding a picture of the user with the device (user picture can be retrieved from social networks or online sources). Android improved the face recognition algorithm by adding *Liveness* check in Android 4.1 (a.k.a Jelly bean). The liveness check means the user has to blink their eyes during authentication [20]. However, this additional feature can be tricked by using photo editing or playing blinking video[1]. There are many hits on face recognition circumventing tutorial on youtube. Furthermore, Android provides a PIN code or a pattern authentication method as a fallback to face recognition during poor lighting and camera malfunction [33]. Hence, any possible attacks on PIN or pattern are also applied here. A three-dimensional (3D) facial recognition has been named as one of sophisticated and more secure authentication mechanism that succeed the security challenges in the 2D facial recognition [13, 20, 105]. As of this writing, this technology is not incorporated in the Android platform.

4.4.1.4 FingerPrint

Smartphones manufacturers have already understood the potential impact of sensors and they have been embedding different types of sensors in mobile devices. Some sensors already appearing in mobile platforms are accelerometers, digital compass, proximity sensor, gyroscope, GPS, microphone, camera, touchscreen [47, 69, 103, 106], and some smartphone vendors like Apple and Samsung have embedded fingerprint sensor

[1]Android 4.1 Jelly Bean Face Unlock tricked by blinking video, [Last Accessed: March 2015], `https://www.youtube.com/watch?v=UKIZBbvloO8`

in their flagship products [77]. Apple made progress by embedding fingerprint sensor on the latest iOS products [2], but the sensor is built only for system applications which means that the fingerprint sensor is not accessible to third party applications. Samsung also embedded fingerprint sensor on some of the smartphone models such as Galaxy S5[3]. Unlike Apple, Samsung has released a public SDK called "Pass"[4] that allows a third-party application to use the fingerprint sensor in the device. The shortcoming of this SDK is that it only works on Samsung smartphone devices in some selected models. The upside is that it lets us provide reinforced security, since we can identify whether the current user actually is the authentic owner of the device[5].

Smartphones Fingerprint API preview for Android M (upon this writing, Android M has not been released, only preview is available[6]) is found here with Sample App. As of this writing, Android Compatibility Definition for Android M hasn't been published. So, if fingerprint sensor, the key hardware component of the fingerprint framework, is left as a "SHOULD" requirement (most likely to be true), then OEMs decide either to include the sensor or not. However, since Android Pay is strongly tied to fingerprint framework, this may drive OEMs to include the fingerprint sensor.

4.4.2 Remote Authentication

MDCS uses password-based authentication to allow authorized user into the system, and there are many password related attacks [83]. Many of these MDCS implement weak authentication schemes based on challenge-response (HTTP Digest Authentication [43]), encoded passwords (HTTP Basic Authentication [43]), or form-based authentication. All the methods try to solve the same problem: the client proves to the server that it knows some secret password P, usually set and exchanged in advance. Hence, in these systems, the password is a key to the kingdom and must be protected while it is at rest and during a transaction. The protection mechanism may affect the protocol adoption for exchanging the credential between client server. Here, we briefly discuss the different type of measures taken to protect the password.

Password based authentication starts with assigning the user a credential such as a username and a password. Traditionally, the password can be generated randomly and saved in the database in clear or hashed. The problem with storing the password in clear is that an attacker who gets into the database can get the password. As a remedy to this issue, the password can simply be hashed using, for instance, SHA-1 algorithm and stored in the database. However, the rainbow table makes it easy reversing the hashed value to recover the plaintext password. This particular problem can solved by appending a unique string characters, called salt, to the password before computing the hash. The salt is not a secret, but it is stored in the database together with

[2]Apple Touch ID sensor, [Last Accessed: May 2015], `https://support.apple.com/en-us/HT201371`

[3]Samsung Galaxy S5 Specs, [Last Accessed: May 2015], `http://www.samsung.com/global/microsite/galaxys5/specs.html`

[4]Samsung Pass SDK for Fingerprint, [Last Accessed: May 2015], `http://developer.samsung.com/release-note/view.do?v=R000000009`

[5]Cult of Android - Samsung Will Allow Third-Party App, [Last Accessed: April 2015], `http://www.cultofandroid.com/53140/samsung-will-allow-third-party-app-developers-use-galaxy-s5s-fingerprint-scanner/`

[6]Android M preview, [Last Accessed: June 2015], `https://developer.android.com/preview/overview.html`

the hashed password. This approach requires significant computational resources to reverse the hashed password, but still vulnerable when a weak password is used. Once the password is protected in the database; the next challenge is securing the credential during client-server authentication. In the next section, we discuss some weak and strong remote authentication techniques.

4.4.2.1 Weak Authentication Methods

HTTP Basic Authentication Scheme (Basic Auth)

The HTTP Basic Authentication Scheme (Basic Auth) is widely used simple authentication method [43]. During user authentication, the client sends a (*base64*) encoded of username and password in plaintext to the server. The server decodes and verifies the request against a plaintext or hashed version of the same username and password. The pros and cons of this technique is discussed as follows:

Pros: Simple to use and works with most browsers. It works without installing client software

Cons

- Basic Auth does not use cookies, session id, or token for session management and every client request must contain username and password. This clearly unsafe practice to send credential in an insecure channel in every request. So, Basic Auth is vulnerable to man-in-the-middle (MITM) attack. This type of risk can be mitigated by using TLS/SSL secure link [58]. We also argue that having TLS/SSL link may bring another security and maintenance challenges that we discuss in chapter 6
- Basic Auth does not have the concept of "logging out". Once the user is authenticated the first time, the Basic Auth user agent on the client caches username and password internally to use it for a subsequent request. Therefore, since Basic Auth does not provide a way to log out or timeout, the user credential stays in the user agent storage (in the browser) for an unspecified period. This problem can not be solved in a standardized way, but it might be solved through some ad-hoc solutions which may or may not work in all clients (browsers).
- Basic Auth does not provide mutual authentication i.e. client and server verify each others' identities, which is one of our key requirement in the requirements list.
- With Basic Auth, the server is dependent on the client user agent with passes in the credentials. For instance, during credential encoding, special characters may be treated differently by user agents in different browsers and might lead to user authentication to fail.

In summary, even though Basic Auth is simple and easy to use, it is vulnerable to many potential attacks, and it failed to meet our security requirements. Therefore, we do not recommend or use Basic Auth.

OpenRosa Authentication API

An alternative secure authentication mechanism to Basic Auth, the OpenRosa Consortium[7] proposed an Authentication API for MDC systems based on HTTP Digest Authentication Scheme [43] with some modification. To authenticate the user, the Authentication API on the client computes a hash of the user credential $(username and password)$ before sending it over the network, which make it safer than Basic Auth. It uses MD5 hashing algorithm that is considered too weak hashing algorithm and *nounce* value for preventing replay attacks. The OpenRosa Authentication API specification details is found in [96]. The pros and cons of this authentication technique are presented as follows:

Pros:

- Unlike Basic Auth, the password is not sent in clear text rather a hash is computed following the specification procedure $MD5(username : realm : password)$, where realm is an authentication provider domain that limits the scope of risk when the passwords are compromised). Similar to Basic Auth, token or session id is not used. But, the computed hash value with nounce can be used for the subsequent request without storing the clear text password.
- Client nounce is used to mitigate chosen-plaintext attacks such as rainbow tables.
- The authentication technique can be used in places or countries that are well known security solutions like HTTPS are not allowed (blocked)

Cons:

- To verify the client request, the server needs to build the same hash. The server must have access to the username, password and realm in plain text to compute the hash. It means that the server must keep the user's password in clear text that takes away all the advantages of storing credentials in salted and hashed values. If we assume that an application is exposed to single realm which is a constant realm for all users, then the server can store the password in hashed form, the same way as we specified above as best practice. If the realm is compromised or changed, then the user must be authenticated again and re-generated the entire users credentials using new realm $(MD5(username : password : realm))$.
- Does not provide mutual authentication. Only the server verifies the client identity, not vice versa.
- The user payload is not protected or the technique does not provide a key exchange mechanism.
- It prevents the use of key derivative functions such as bcrypt, scrypt, or PBKDF2 that slow brute-force attacks.
- If an attacker gains an authenticated computed hash value, $MD5(username : password : realm)$, a brute-force attack is possible against the password. Further more, the attacker can perform Man-in-the-Middle attack and instruct the client to use the Basic Auth instead of the digest authentication.

[7]OpenRosa Consortium, [Last Accessed: April 2015] `https://bitbucket.org/javarosa/javarosa/wiki/OpenRosaAPI`

In general, OpenRosa Authentication API are considered as weak authentication method[8]. It does not fulfill most of our security requirements and it vulnerable to several attacks. As a result, we do not use or recommend the authentication API for MDC systems.

Form-Based Authentication

Form-Based Authentication is another popular but weak method of remote authentication. The name "Form" refers to the notion of a user being presented with an editable "form" to fill in and submit in order to log into some system or service[9]. The server validates the credential pragmatically and creates a session for successful authentication. Form-based authentication does not use digest or basic auth protocol rather the programmer decide the way the client communicate with the server. Like basic auth, form-based authentication still requires a secure tunnel for user credential.
As a summary, these authentication methods are designed to make simple client and server authentication but the specifications itself do not address the security concerns such as exchanging credential in a secure tunnel or how both client-and-server mutually authenticate each other. Furthermore, the methods do not provide extra features such as a secure key exchange protocol. In the next section, we present a comprehensive, strong authentication method called Secure Remote Password Protocol (SRP) [139]. The Autheticator uses the SRP protocol for remote user authentication and key exchange.

4.4.2.2 Strong Authentication Method

A strong password based authentication method provides mutual authentication, reasonably protects low-entropy password from offline attacks, combine authentication and key exchange schemas based on only the user password, resistant to a number of attacks, and compromised database is not enough to impersonate the user. Secure Remote Password-Based Protocol (SRP) one of the strong authentication based on user's password and presented as follows.

Secure Remote Password Based Protocol (SRP) Background

The Secure Remote Password protocol was first proposed by **Tom Wu** at the Stanford University in 1997 and presented at NDSS (Network and Distributed System Security Symposium) in 1998. [131]. The protocol is standardized in RFC 2945 [139]. Since the standardization, the SRP protocol has been studied, optimized, and released with new versions such as SRP-6 [132](2002) and SRP-6a(2005). The SRP-6a maintained to stay without any changes for the last decade.

We have chosen the SRP protocol because of its simplicity, resistant to many attacks, and its security features such as mutual authentication and key exchange mechanism. Next, we present the protocol description in detail.

[8]Competitive Analysis of SRP, [Last Accessed: May 2015], `http://srp.stanford.edu/analysis.html`

[9]Form-Based Authentication, [Last Accessed: April 2015],`https://en.wikipedia.org/wiki/Form-based_authentication`

The SRP Protocol Description

The following protocol description is based on the original work [131, 132], specifically the latest version, SRP6a.
The protocol begins with assuming that the user gets hold of a username (Identity) and password distribute through some kind of side-channel. On the server-side, SRP does not compute a hash of the password (salted), instead it computes a verifier, an equivalent of the hashed password, yet calculated in a different way than the traditional hashed password.

$$x = H(salt, I, P) \tag{4.1}$$
$$v = g^x \tag{4.2}$$

where:

$$\begin{aligned} x &= \text{private key derived from user salt, identity, and password} \\ salt &= \text{randomly generated salt} \\ I &= \text{user identity, i.e. username} \\ H() &= \text{Hash function} \\ g &= \text{primitive root modulo n (a.k.a generator)} \\ v &= \text{verifier} \end{aligned}$$

First, equation 4.1 generates the private key x (*password-equivalent*) using user specific salt, identity, and password as it is specified in the SRP protocol [132]. Then, the value of x is passed to equation 4.2 to generate the verifier v. The private key x can be generated in different ways and the verifier changes accordingly. The most important point here is that the actual implementation of x is designed to maximize the resources needed when someone tries to reverse and recovery the actual password. Finally, the server discard the password and the private key x and only stores the verifier v, salt, and I (username) in the user database.

Note: Before we delve into the details of the SRP authentication and key exchange handshake, we would like to inform the reader that all the arithmetic operations are done in module n, where n is a large safe prime number. This means that g^x in the equation 4.2 should be read as $g^x modn$. The values of n and g can be fixed or exchanged during the handshake.

The optimized version of the SRP-6a protocol handshake first initiated by the client by sending unique user identifier (username) and asks for server parameters as shown in figure 4.3. Generally, the handshake described in the figure 4.3 is easier to understand, efficient, and requires only two round trips. By changing the order of the exchanged messages in the optimized version, the number of round trips is reduced by one. As we stated previously, the values of n and g can be fixed or it is also possible to exchange these values during the handshake as shown in the figure 4.3. The overall handshake is summarized as follows.

1. The client initiate the handshake by sending a request with user identity I, i.e.username and ask the server some parameters

2. The server lookup salt and verifier that are associate to the user identity
3. The server chooses random ephemeral private key b and computes the respective ephemeral public keys B
4. the server return parameters:g, n, $salt$, B
5. The client chooses random ephemeral private key a and computes the respective ephemeral public keys A, computes the scrambling parameter u, calculate the premaster secret S and then the the evidence value $M1$.
6. the client send the computed evidence $M1$
7. the server verifies $M1$ and in order to the client to verify the server, the server computes $M2$ and return as a response to the client.
8. If the mutual authentication is success, finally the client and server generate the session key K.

For implementation special recommendations are stated in the selection of prime numbers n and g [132]. Furthermore, additional safeguards are stated in the SRP-6a protocol, in particular:

1. The client will abort if it receives B = 0 or u = 0
2. The server will abort if it detects A = 0
3. The client must show his proof M1 first. If the server detects that the proof is incorrect, it must abort without showing its own proof M2.

SRP Protocol Security Analysis

A number of studies analyzed SRP strengths and weaknesses [75], addressed security flaws in hash function used, proposing further changes on the protocol [19], and others debated over the role of the protocol compared to other similar protocols [35]. The SRP protocol is resistance to a number of attackes. Here, we present some of them but further analysis on different attacks are presented in details in [25, 67, 75, 130, 132, 140].

Passive Attacks

Attacks on the Session Key K: The main target of this type of attack is to obtain the session key K from client/server exchanged values such as A, B, M1, M2. The resistance of SRP is based on one-wayness of the discrete exponentiation function. From SRP-6 these attacks have been made even harder, since the evidence messages M1/M2 are calculated from the premaster secret S, instead of the relatively smaller key K [52].

Dictionary-Based Attacks: Trying to guess the password off-line using a dictionary does not give a real advantage, since the attacker should also try exhaustively all the possible random values of a and b. An on-line dictionary attack corresponds to posing as the user and trying to guess the password. This can be limited by restricting the number of password attempt [52].

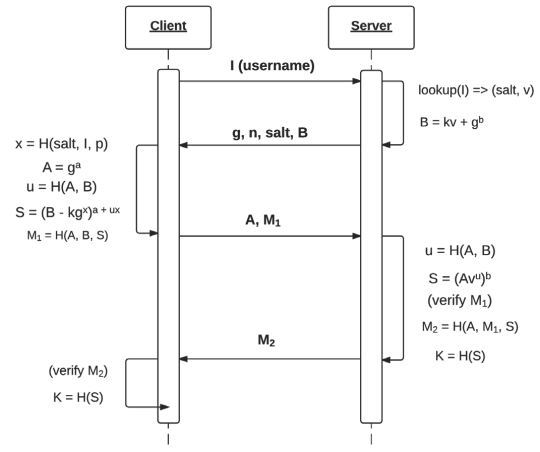


Figure 4.3: Optimized SRP Protocol Handshake

Notation:

$$
\begin{aligned}
x &= \text{private key derived from user salt, identity, and password} \\
salt &= \text{randomly generated salt} \\
I, P &= \text{user identity, i.e. username and password} \\
H() &= \text{Hash function} \\
g &= \text{primitive root modulo n (a.k.a generator)} \\
v &= \text{user password verifier} \\
u &= \text{random scrambling parameter (public)} \\
k &= \text{multiplier} \\
a, b &= \text{ephemeral random private keys} \\
A, B &= \text{corresponding public keys} \\
S &= \text{premaster secret} \\
K &= \text{cryptographically strong session key} \\
M1, M2 &= \text{evidence messages (or proofs)}
\end{aligned}
$$

Active Attacks

Replay Attacks: SRP is immune to this kind of attacks, due to the randomness that influence the key generation.

Man-in-the-Middle Attacks (MITM): If the password-equivalent x is known, an attacker can act as the client. If the verifier v is known, it can masquerade as the server. But, to perform a MITM, both values are required, meaning that we first need to compromise both client and server.

Active Dictionary Attacks: If an attacker manages to steal a user's verifier (e.g., gaining access to a server's password file), the attacker can attempt a dictionary attack to recover that user's password. This possibility is clearly mentioned by the author of SRP in [25, 52].

The latest attacks on the SRP based systems are discussed in [49, 89, 125]. Researchers recommend to use more advanced hash function and a modulus value n of at least 1024-bit. Thus, we conclude that the security of SRP is based on the security of its primitives.

4.5 Proposed Solution

In this section, we present a secure authentication service that systematically addresses the functional and non-functional requirements discussed in the sections 4.2.1 and 4.2.2. Before delving into the details, we define concepts, preliminaries and protocol used in the proposed solution.

A user account (username and password) must be created and stored on the MDCS server before the user starts using the system. Even though the user only uses username and password to authenticate himself or herself, the way the user authentication is implemented on the server, differs from one authentication method to the other. Besides, the client application role on user authentication also changes from one authentication method to the other accordingly. For example, if we consider basic authentication, the client is expected to send username and password to the server in clear text or through a secure tunnel. Whereas in case of digest authentication, the client challenges the server instead of sending the actual user's password. So, when the MDCS system adopts a new method of authentication, it should examine the advantages of the new method and required changes to integrate it into the existing system. Based on the discussions in the section 4.4, we identified the SRP protocol as the preferred client - server authentication method because of the following list of advantages[10]:

- safe against snooping
- immune to replay attacks
- exchanges a session key in the process of authentication
- can provide mutual authentication
- resists the dreaded off-line dictionary attack based on exchanged messages offers perfect forward secrecy
- can tolerate a compromise of the verifier database on the host

First, we discussed the changes needed on the MDCS server in order to incorporate the SRP based authentication method. Later, we presented how the SecureMDC framework on the client, authenticate the user using the SRP protocol. Since the SRP protocol does not provide a message exchange format, we chose the JSON Web Token (JWT) for handling the SRP handshake. Here, we provide a proper definition of JWT as follows:

[10]SRP Advantages, [Last Accessed: June 2015], `http://srp.stanford.edu/advantages.html`

JSON Web Token (JWT): is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JavaScript Object Notation (JSON) object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or MACed and/or encrypted. [62]

N.B.: When we use the word SRP anywhere in this disseration, we are referring to SRP-6a, the latest and improved version of SRP-3.

4.5.1 Users Sign-up on the MDCS server and Account Distribution

Figure 4.4 and 4.5 show how users are signed up to MDCS system. Many of MDCS systems use a user management software component on the server-side for signing up users into the system before conducting data collection. This part is a manual process and performed by the project manager. MDCS system users can be data collectors, form manager, data viewers, field supervisors, coordinators, and any other actors within MDC eco-system. This individual user profile data is recorded through the user manager console and each user is assigned a username (usr) and password (pwd) for accessing the system.

Furthermore, resources access privileges are assigned to each account. These user accounts are distributed through a side-channel. The side-channel can be an e-mail, an SMS, in person, a phone call, a letter, or some other means. The choice of the side-channel depends on several factors such as users geographical location, communication network accessibility, users ability to use these channels, security, and others. Hence, the choice of the distribution channel varies from system to system, and it is not covered in our work. However, we believe that this is an important piece of research to be investigated in the future. Especially when distributing user account in large scale projects.

Some of MDCS force the user to change the initially distributed password, and some others do allow the user to keep initially distribute password.In any case, once user receives the account, any further interaction with the server is protected.

Figure 4.6 and 4.7 compare user sign-up flow between the existing MDC systems and with the introduction of SRP protocol respectively. After the sign-up form is filled, the current systems take the user password and randomly generated a salt value as an input to a hashing function such as SHA-256 as shown in figure 4.6. The computed hashed password together with the salt is stored in the data storage. When using SRP, what is stored is a password verifier v instead of the hashed password.

Even if the SRP protocol standard outlined in RFC2945, suggest to create the private key, x, as $SHA1(salt|SHA1(Identity|" : "|Password))$, it is possible to generate the value x in different ways. For instance, it can be computed as $H(salt, password)$ or using some key derivation function such as Script or PKCS#5:PBKDF2 as outlined in RFC2898. Therefore, the verifier creation is mostly left to a particular implementation. Figure 4.7 shows how the server can simply compute x as $H(salt, usr, pwd)$. Once the users account are created and distributed through a side channel, users authenticate themselves through the SRP handshake.

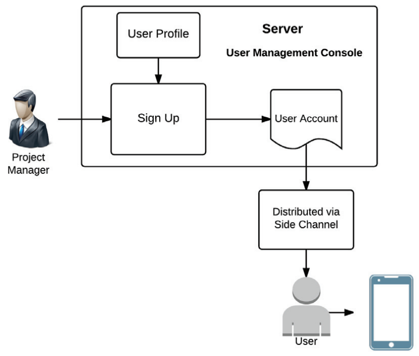
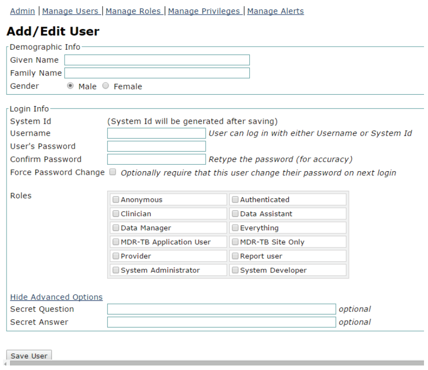


Figure 4.4: Users Sign-up and Account Distribution



Admin | Manage Users | Manage Roles | Manage Privileges | Manage Alerts

Add/Edit User

Demographic Info

Given Name	
Family Name	
Gender	[✓ Male] Female

Login Info

System Id	(System Id will be generated after saving)	
Username		*User can log in with either Username or System Id*
User's Password		
Confirm Password		*Retype the password (for accuracy)*
Force Password Change	☐ *Optionally require that this user change their password on next login*	

Roles

☐ Anonymous	☐ Authenticated
☐ Clinician	☐ Data Assistant
☐ Data Manager	☐ Everything
☐ MDR-TB Application User	☐ MDR-TB Site Only
☐ Provider	☐ Report user
☐ System Administrator	☐ System Developer

Hide Advanced Options

Secret Question		*optional*
Secret Answer		*optional*

Save User

Figure 4.5: OpenMRS User Registration

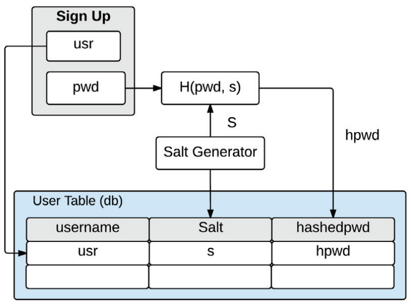


Figure 4.6: Current Users Sign-up Process

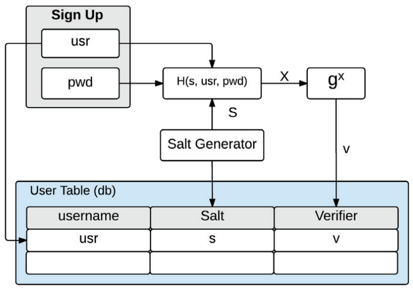


Figure 4.7: Users registration with SRP

As a summary, we assume that the user's credential for accessing the system consist of a username (usr) and a password (pwd), provided securely to legitimate users. Typically, the username is assigned by MDCS, while the password is chosen by the user after the initial authentication using the distributed password. Password strength policy is enforced when the user chooses the new password.

4.5.2 General Architecture of the Proposed Solution

In order for the proposed solution to work properly, we are making the assumption that the application is distributed and installed on mobile devices in a secure manner. The secure application distribution ensures that the application code that runs on the client side is not tampered with during distribution or installation. We have provided a thorough analysis on the application provisioning in Chapter 7.

Figure 4.8 shows the general authentication service architecture of the SecureMDC framework. It incorporates main components including the authentication security module, an account store, a key store, the SRP and the JSON Web Token protocols.

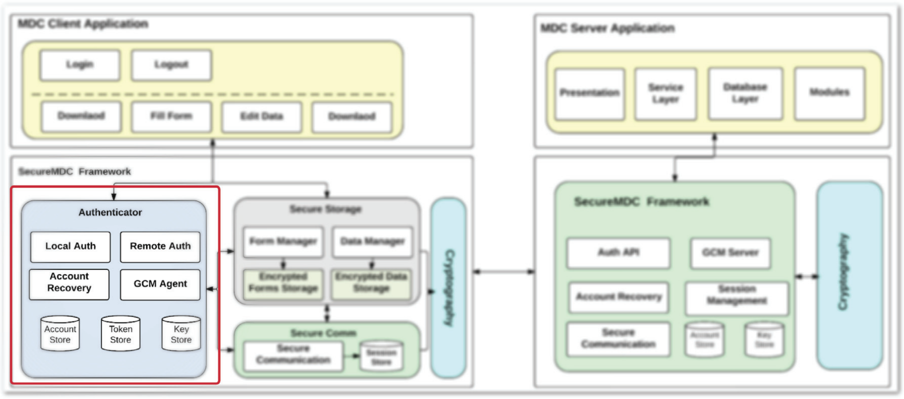


Figure 4.8: Authenticator: Local and Remote User Authentication Module

The authentication service itself is designed to be platform and application independent and provides an authentication layer on top of an existing application layer. The aim is to be able to provide adequate secure authentication service based on the requirements and restraints of any given application or project. The intended area of use is mainly securing MDCS in low-budget settings. User Authentication on the mobile device is organized into two phases: User registration and local authentication. The user has to register on the phone the first time, and this requires remote authentication on the server with the server password. In the next section, we discuss the user registration.

4.5.3 User Registration

Unlike a web-based service where data is frequently exchanged between server and client, MDCS client applications are designed to hold users data for a certain period. The collected data may, in fact, stay for days or months on the device. Moreover, due to limited budgets multiple data collectors can use a single mobile device. Therefore, the solution should fulfill the following requirements:

1. Data access restriction between registered users on a device

2. Local authentication on the device

Figure 4.9 shows the Authenticator module with its main components used to authenticate the user locally and remotely. The numbers labeled in the figure are used to describe the user registration steps and we present it shortly.

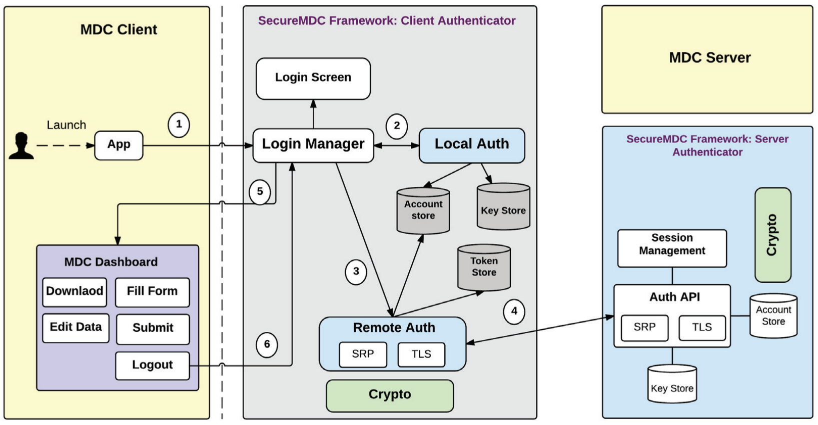


Figure 4.9: User registration flow Diagram

The **Local Auth** component in the figure 4.9 is responsible for handling user authentication on the device. This component checks if a user does exist on the device before initiating the server authentication through the **Remote Auth** component. If the user does not exist, the Remote Auth component starts a server authentication procedure through the SRP protocol and verifies the server response. If the authentication is a success, the **Remote Auth** component creates the credentials used to authenticate the user during offline access. If the server authentication fails, it allows the user to try to log in again.

The whole process of authenticating a user to the server for the first time is called *user registration* on the device. At the beginning, the device has no information for authenticating the new user and needs help from the remote server to verify and authenticate user credentials. Thus, connectivity to the server is required. Here, we present steps needed to register a user on a mobile device. The user registration involves mutual authentication between client and server, creating credential storage on the device for offline authentication, changing new password, and mapping the user profile to the device identity (s)he is trying to register with on the server. All these steps are performed using the Remote Auth component of the Authenticator module without involving both MDCS server and client application as shown in the flow diagram in the figure 4.9. Next, we use the numbers labeled in the figure 4.9 to explain the required steps for user registration.

1. When a user starts the MDCS client, the client redirects the user to the Authenticator module. A login manager component of the Authenticator launches a user login form. At this point, the Authenticator takes complete control of the user authentication process and gets the user credential directly via the login form.
2. The Local Auth component of the Authenticator, looks up its local credentials storage for the current user credential. If the user does exist, it proceeds with offline authentication, otherwise, returns false.
3. The login manager then initiate the Remote Auth component to perform server authentication using the SRP protocol over HTTP.
4. The Remote Auth component runs the SRP protocol for mutual authentication and key exchange. The Remote Auth verifies the server response and allows the user to change the old password with a new one. Subsequently, the component registers the user on the device by creating the necessary offline credentials used to authenticate the user locally.
5. Once the user registration is completed; the Authenticator returns the control and the authentication result back to the MDCS client. If the authentication is success, the MDCS client allows the user to access the client services such as form downloading, form filling, editing, and submitting to the server.
6. At the end, the user can initiate the logout function to inform the Authenticator. Accordingly, the Authenticator logs out the user and clean user related credentials and data properly.

In the next sections, we delve into each of the above user registration steps and present how we approach and develop a working authentication module for feature phones and smartphones based MDCS, both for the Java ME and Android platforms. The interaction and handling control of the authentication process between the MDCS client and the Authenticator module is also discussed.

Step 1: Passing control over the Authenticator Module from MDC Application

Passing control from the MDCS client to the Authenticator module and vice versa differs from Java ME platform to Android.

In the Java ME platform, there are limited ways of passing control. Understanding how the system thread works in Java ME platform was critical in order to be able to make a responsive application that will not deadlock. The system thread won't be able to do its job, such as updating the current screen until it is not in use. For a more in depth explanation see the article "Networking, User Experience, and Threads" [11]. Because of this, the Authenticator Module has an abstract method called userMenu(), the purpose of this method is to give control back to the application once the Authenticator has logged a user in. The logging in steps are run inside a number of threads, this will be discussed in the next section. Once these threads have logged in a user, they call the userMenu() method and the application can run its code. The secureclient threads will then wait until the application calls logout() at which point they will restart the log in process. There would be nothing wrong with using a normal method instead of the userMenu() callback method. One such method could be logInScreen() for instance, which would return once the user has been logged in. However, the downside of this approach is that the programmer would be responsible for correctly running this method in a separate thread to avoid application deadlock. This would be a nuisance, go against good API design as proposed by Josh Bloch [12] and result in boilerplate code [13].

Android platform maintains the Java ME thread handling mechanism with some built-in remedies for better ways handling thread related operations. When an Android application is launched, the system creates a thread of execution for the application, called "main" or "UI thread". The thread is responsible for screen updates and other system related tasks such as dispatching events.

If we compare the main thread handling between Java ME and Android, in the case of Java ME, when we tried to make a waiting screen for user feedback while the application was communicating with a server, the application would freeze for the duration of the communication and then for a fraction of a second display the waiting screen before setting the next screen. This happened because the lengthy operation of network access was being run in the main thread, so while the system had been instructed to set the new screen, it had to wait for the server communication to complete in order to be freed and thus able to order to do so, at which point the server communication would be done. As a remedy, we could hand over the intensive work to a worker thread, but when the intensive work was completed, it was not possible to manipulate the main thread from the worker thread in order update the screen with some result.

Similarly, Android's single thread model strictly follow the following two rules:

- Do not block the main thread
- Do not access the Android UI toolkit from outside the main thread

Unlike Java ME that lets applications freeze when there is an intensive operation on the main thread, Android wait few seconds (up on this writing, 5 seconds) and the application process is killed. A user is notified with "application not responding (ANR)" error. For a more in depth explanation on Android Processes and Threads see this [14].

[11]Jonathan Knudsen. Networking, user experience, and threads. [Last Accessed: Dec 2014], http://www.oracle.com/technetwork/systems/index-156145.html

[12]Joshua Bloch. How to design a good api and why it matters. Talk/video: [Last Accessed:June 2011], http://www.infoq.com/presentations/effective-api-design

[13]Wikipedia - Boilerplate code, [Last Accessed: August 2011], http://en.wikipedia.org/wiki/Boilerplate_code

[14]Android Processes and Threads, [Last Accessed: January 2015], http://developer.android.com/guide/components/processes-and-threads.html

Android also offers public API to access the UI thread from other threads but it comes with more responsibility for handling all threads life cycles and furthermore the code might get complicated and hard to maintain when threads related operations grow within the application. In order to alleviate this problem, Android provides "AsyncTask" public API to execute intensive operation on the background asynchronously. Once the operation is completed, the AsyncTask handles updating the main thread with a new result.

Furthermore, on the Android platform, the proposed secure solution can be developed as service application running on a separate process. Hence, the MDC application can access the service through Interprocess Communication (IPC) which is the underlying apps communication mechanism in Android platform. Android supports a simple form of IPC mechanism through intents [15] and content providers [16] for asynchronous and synchronous communication respectively. So, the security services can be accessed through intents asynchronously while, for faster synchronous call execution, Android offers an alternative solution to create a communication channel between two application running on different processes. This is achieved using a Binder Framework[17]. Remote Procedure Calls (RPCs) is provided through Android's IPC based on Android Binder framework. The RPC mechanism using binder enables us to call a method or an operation that is running on another process from the MDCS application process and get a result of the operation to the MDCS application. Since Android IPC mechanism involves the operating system, a simple RPC call requires disassembling the call and its data to the level the operating system can understand, transmitting to the remote process address space, reassembling and executing the call on remote process, and return the call result back to the caller process on the opposite direction. The Android platform handles all these underlying complexity. This binder based IPC mechanism is more flexible and faster to establish a communication channel between the applications.

Android centralized user's account management system (a.k.a Account Manager) is one of many system services that uses a binder as an IPC mechanism. The Account Manager handles storing and managing user credentials on the device. In order to accommodate different authentication schemes for online services such as Dropbox, Google, Twitter and Facebook, the Account Management framework does not provide any specific implementation. Rather it is open through a pluggable Authenticator module. The Authenticator module is an implementation of a specific authentication service and implemented by a particular service provider. Therefore, we can implement our own Authenticator Module that fulfill the SecureMDC Authenticator Module requirements.

The Account Manager provides access to the centralized account registry via a public API. Any application can delegate the Account Manager to authenticate users online account with the remote server and store credentials on the device system storage. The credential store is outside the application data store and is not accessible from any application on the device.

All components in figure 4.10 except for the AccountManager are running as an

[15]Android Intents, [Last Accessed: April 2015] `http://developer.android.com/guide/components/intents-filters.html`

[16]Android Content Providers, [Last Accessed: April 2015], `http://developer.android.com/guide/topics/providers/content-provider-basics.html`

[17]Android Binder, [Last Accessed: March 2015] `http://elinux.org/Android_Binder`

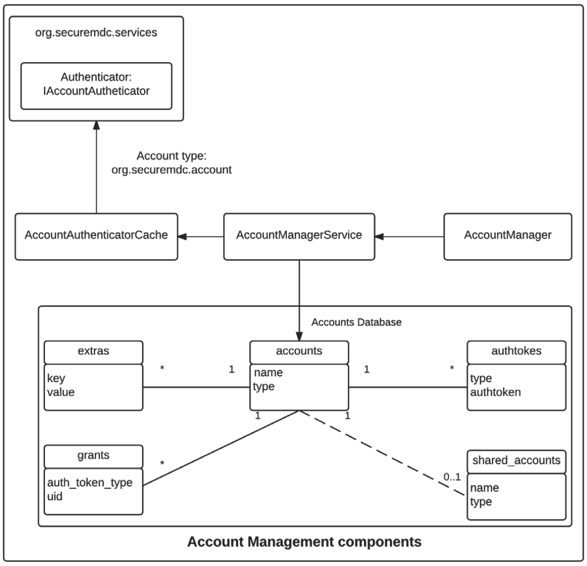


Figure 4.10: Android Account Manager Components

authentication service in a separate process. This authentication service is exposed to the MDC application by the AccountManager via a binder. The Authentication-ManagerService is the core component of the account management framework which handles persisting account data in the account storage. The Authenticator Module is the actual implementation of SecureMDC Authentication and is identified by a unique account type "org.securemdc.account". Finally, the AccountAuthenticatorCache identifies the Authenticator modules by scanning the packages that define the Authenticator modules and make them available to the AccountManagerService.

The sequence diagram in figure 4.11 shows the interaction between the MDC application and the Authenticator module via a binder. The MDC application is running on process A and access the Authenticator module which is running on process B. The method call addAccount with the passing parameters are decomposed into transaction and data before it reaches the binder driver which is responsible for handling process to process communication. The transaction and the data are reassembled on the other end, and the method call is performed on the Authenticator module. The Authenticator module returns back the result on the opposite direction. The Android platform provides all the underlying complexity of data marshaling, unmarshaling, transaction, and thread handling. From the MDCS application perspective any method call feels like a local call, but it is executed on a remote service using RPC.

At the end, the returned result from the Authenticator may contain a token when the authentication is a success or an error if the authentication fails. The MDCS application verifies the response and allows the user to use the application.

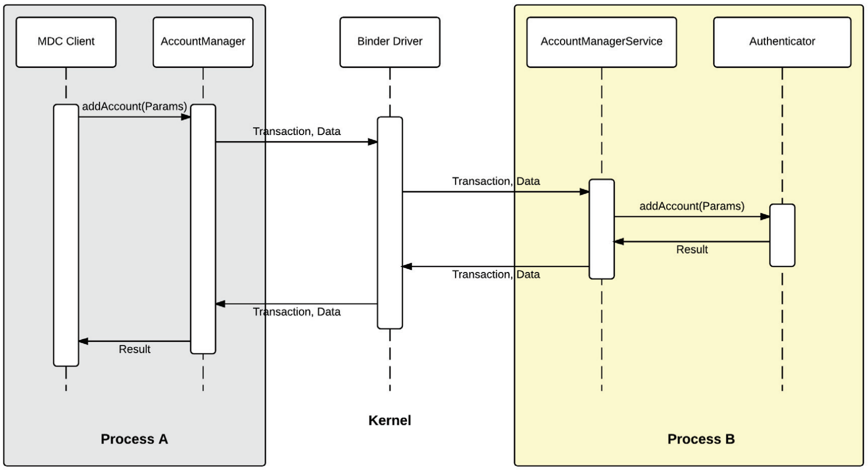


Figure 4.11: Android Account Management Sequence Diagram

Step 2: Does a User Exist on Local Account Storage?

Once the control is passed over to the Authenticator module in step 4.5.3, the Authenticator module launches the login screen to capture user credential in a secure manner. Then, the Local Auth component in figure 4.9 takes the unique username and checks if the user does exist in the account storage. If user does not exist, the Remote Auth component is notified to authenticate the user credentials with the remote server using SRP protocol.

Step 3 and 4: Client-Server Mutual Authentication

When we began this work in late 2010, we proposed a home grown protocol based on public key cryptography to establish mutual authentication between client and server [74]. We first assumed that the MDC application had been configured and installed properly on the mobile device. When the application opened for the first time, the server URL would be entered to request and retrieve the server public key. However, the user could type the URL incorrectly or a man in the middle could manipulate the server response. Therefore, the protocol required the client to authenticate the server public key. This is done by challenging the server to decrypt a secret key. To be able to perform the decryption, the server must use a shared secret that was distributed in advance to the users together with their user name and password. If the server can decrypt the secret key, it will be able to encrypt its response consisting of a unique application id to identify the specific device running the application in future requests, and a seed to improve the quality of the keys generated by the random generator on the mobile phone. The public key digest is used as a proof that the server sending the response is the same one that initially supplied the public key.

However, since the public key is inside the application manifest file we can expand on this idea and calculate the digest of the whole Java ME JAD file (Java Application Descriptor (JAD) as it is discussed in the chapter 2 under the section 2.4.1), containing the public key and thus verify not only the key, but an arbitrary set of Application manifest properties that we want to make sure are not tampered with. The digest is computed on the server and sent as part of the server authentication response. The client can then calculate the digest for the application manifest present on the device, and compare this with the one from the server. The hashing is done by concatenating a string consisting of the values we want to verify in a predefined order. The order is the same on client and server. This alone however leaves the system vulnerable to spoofing attacks, if we have the properties "a=1","b=2", and "c=3" and we concatenate them in the same order, there is nothing stopping an attacker from leaving "b" and "c" blank, and setting "a=123". To prevent this, we prefix the value with its length + ';'. Any empty values would have the length of 0. The previous example would then yield the value "3;1230;0;" which would not be the same as "1;11;21;3". The concatenated string is then hashed and the hash from the server compared to the hash from the client values.

A drawback of the Pre-Shared secret approach, is the difficulty in distributing many activation codes, especially if there are thousands of collectors, and the difficulty of entering a complicated and long key on a mobile phone. So, a third option, which is also used today, is to deliver the phones to the collectors pre-configured by someone responsible for this specific task, so that installation can be done manually. In this way, we can get a genuine public key that can be used for a mutual authentication together with user's authentication credential.

In some previous work [44, 73] we argued why HTTPS cannot always be considered as the best solution for mutual authentication and secure communication in low-budget data collection projects. The reasons were mainly three: bad support/implementation in older phones, criticisms towards the Certificate Authority Trust Model (see for instance [112]) and the cost of the certificates. A typical feature phone would, in fact, have a limited list of root certificates which could not be modified by the user, and that was not standardised across models and manufacturer. This would make it difficult for a project to choose a CA from which to buy an SSL certificate that could be supported by all the handsets deployed in the project. On the other hand, not being able to modify the list of root certificate would give a guarantee that self-signed certificates could not be accepted, providing higher security. The result however was that most MDCSs would simply not use HTTPS at all. With smart phones, the support for HTTPS improved dramatically, but, especially in Android, self-signed certificates and CA signed certificates are treated in the same way, with the consequences discussed in the previous section on app distribution chapter 7.

We looked then for other protocols that are not dependent on certificates, and realised that a password-based approach would be best. One of the most reliable such algorithms is the Secure Remote Password Protocol (SRP), a Password Authenticated Key Exchange (PAKE) protocol based on a pre-shared password for mutual authentication and key exchange, standardised in RFC 2945 [18]. It allows mutual authentication and secure key exchange, while being resistant to on-line brute-force and Man-in-the-

[18]RFC 2945, [Last Accessed: December 2014], `https://www.ietf.org/rfc/rfc2945.txt`

Middle attacks (MITM). In addition, it can be used in conjunction with TLS [19] to create a secure transport layer without the need of certificates or new protocols. In previous MDCSs this approach would have been ideal since every collector would get a password based account to access the server anyway, but they would lacked the necessary APIs to support it. Besides, with new authentication technologies like OAuth, that can be used to authorise access to some server resources without sharing the user's credentials, a password based approach might not be necessary. ODK for example is using Gmail accounts to grant access to their servers, without having to issue a new user-name and password for each collector, as long as they have a mail account with Google.

Thus, we started looking at another strategy called certificate pinning. With the help of strict-host-key-checking feature, an OpenSSH client can be configured to grant or reject a connection request by verifying incoming request key against known keys list stored in the key store. This idea has been adopted in certificate or public key pinning. Pinning is an emerging concept of associating a given client with a list of known public keys or certificates. The client then verifies an incoming connection request against the list and grant or reject accordingly. After Man-in-the-Middle attack on Gmail's SSL through a compromised DigiNotar certificate (DigiNotar was one of well know CAs before bankrupting) [99], Google and other organisations have been actively engaged on making pinning as a part of their products. As a result, there are public APIs to implement pinning solution on mobile apps and web browsers. There is one fundamental problem with pinning though. There is no secure way of distributing the certificates or public keys to a client at the beginning of a project in a distributed environment. Basically, there are two ways to distribute the certificate or public key to the client. It can be incorporated with the application before distribution install or it can be fetched after the application is installed. OWASP recommends the former - before app is installed [99]. However, if we choose the second distribution scheme proposed in the previous section, where no project specific configuration can be added to the app, we have to go for the later approach. That is, to fetch the keys and certificates after the application is installed. We found SRP protocol to be a potential candidate to accomplish this task.

Assuming we downloaded a genuine app, we propose to use SRP based on a one-time password delivered to the user by SMS or email, and use this protocol to authenticate the server for the first time and download its certificate. Successively, we use the certificate pinning strategy offered for example by openSSH and use HTTPS for future communications and perform the chosen user authentication on this secure channel.

In general, given that the application has been correctly and securely installed and configured, and the user has a unique identifier and a pre-shared secret, e.g. username and password or a one time password, we want to achieve the following requirements: Confidentiality; Mutual Authentication; Protection against reply and MITM attacks; Data Integrity; Key generation and management; and Compatibility with the existing applications.

SRP is currently resilient to many attacks and no vulnerability proof exist. SRP was proposed by Thomas Wu and its latest version is SRP-6a. Due to the strength and the simplicity of the protocol, we can use SRP to fulfill all security requirements except data integrity, since it is used for authentication and not secure data transport.

[19] SRP-TLS, [Last Accessed: April 2015], `http://tools.ietf.org/html/rfc5054`

Compatibility is an issue as there are surprisingly few APIs offering the complete SRP protocol on both client and server side. We are working on this issue. The two SRP round trips needed to perform mutual authentication are shown in Figure 4.12. The final result is a shared symmetric key K, which can be later used in a TLS based session for example, or to encrypt the server certificate. According to the project setup, SRP might be used only once to download the server certificate, or every time in conjunction with TLS and the user password. Whether user authentication is based on [20] user certificates, passwords or one time codes is also open. In any case, when using TLS based communication, all our requirements would be met.

The SRP scheme requires two request-response cycle to accomplish mutual authentication and exchange keys. As shown in Figure 4.12, the client initiate the communication by sending a request with user's unique identifier, I. Upon the request receives, the server generates its public key, B, by using a verifier which is a result of g, Hash(I, salt,password), a constant value k, and a random number b. The client uses the server response B, g, n, and the user salt to compute its public key and a proof, M1, for mutual authentication. After the server verify the proof, it perform a similar computation and respond with its proof, M2, to client. Finally, the client and server generate the session key, K, using the pre-master secret Hash(S).

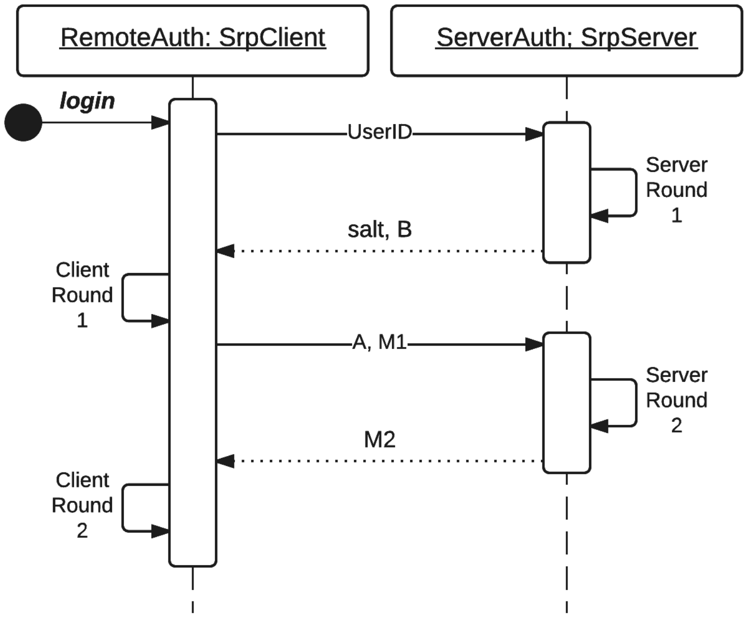


Figure 4.12: SRP-6 Protocol

In our approach, the SRP scheme is used only once when the client side application

[20]The OAuth 2.0 Authorization Framework, [Last Accessed: June 2015], `http://tools.ietf.org/html/rfc6749`

is used for the first time. Once the two-way authentication is completed and shared a session key is created, the client retrieves the server certificates/public keys through the established secure channel and use the keys to create TLS link for later transactions. For further interaction with the server, the client does not need to initiate SRP, instead it uses the pinned certificate to establish a standard TLS channel and the OAuth protocol, or a usual password based login, to achieve all requirements of a secure communication.

Step 5: Passing control back to the MDC Application

When the user authentication is completed, the secure framework returns control to the MDC application together with the authentication result. If the authentication is a success, the result may contain a token that the MDCS client uses to interact with the secure storage module for form and data related activities. The token may have a set of key/value pairs such as user identity, expiration time, token id, and so on. The token may also have a signature field if the Authenticator Module signed it. One may wonder what happens if the MDC app compromises the token. The Authenticator module can then revoke it and furthermore, since we are assuming the MDC app and the secure framework are signed by the same key, the secure framework always checks the signature of the caller before it processes the request. Hence, we can protect the user data even when the token is lost or compromised.

The Authenticator also creates the necessary credentials that is used to authenticate the user during offline. If the authentication is successful, the Authenticator creates a user keystore in platform protected area (further discussed in chapter 5). In the keystore, a storage key to encrypt stored data ($STKey$ in figure 4.13) can be created for the specific user, either by the server or the mobile phone itself, and the user may be asked to choose a new mobile password to access the encrypted storage. A copy of user master key is kept on the server in clear, since this is a private server we trust, and allows for recovery. A solution for untrusted server such as a cloud-based server is discussed in chapter 6. A master key MK is created from the mobile password and used to encrypt the $STKey$. A Hash-based Message Authentication Code (HMAC) is generated by combining MK and the encrypted $STKey$. Local authentication can now be performed using this HMAC.

Step 6: Logout

The logout process is an important step in cleaning the data and any user related account from memory. The Authenticator module expects the user to invoke this service and the Authenticator takes the charge and cleanup the keys, invalidate token, and inform the storage module to encrypt or re-encrypt available data in a user working area. The user working area actually contains user requested forms or data in-memory. We can see the scenario where the MDC app process is killed or the user forgot to invoke the logout function. Since both MDC app and the secure framework are signed by the same key and use the same process, the former one can easily be handled. However, for the latter, the secure framework monitors the user's activity and if the user does not communicate for a certain period of time, the secure framework takes immediate actions such as auto-logout. Internally, this is handled through the Android IPC framework.

4.5.4 Local Authentication

Local authentication on mobile device is a gateway for accessing any data stored on the device memory. Thus, when offline authentication is enabled, the Authenticator module protects data encryption keys and other user related credential on the device. Thus, when designing a solution for local authentication, we had to account for some typical scenarios in mobile data collection as described in section 2.3.1 and 5.3. In particular that multiple users should allowed to use the same phone and that Internet access might not be always be available. This means that mobile devices can no longer be considered private or personal to a user and that most of the data collection might have to be done offline. From a security perspective this translates into the following concerns:

1. Confidentiality (encryption)
2. Authorization (users can access only their own data)
3. (Off-line) authentication
4. Password and data recovery
5. Password changes should account for the possibility of using the same credentials to authenticate on different phones
6. Data should be reasonably protected also if all information stored on the phone is available to an attacker
7. Breaking the storage encryption should not compromise the rest of the system

In this section, we go through different possible solutions, discuss why they do not meet all the requirements, and gradually improve them until we get to a satisfactory one. Notice that, although at a high level we are guaranteed some security properties because we use standard algorithms which are proven to be secure, the security of actual implementations depends on the correct implementation of the crypto libraries used, the correct usage of the algorithms and the security model of the platform. Such libraries might be Bouncy Castle [21] for Java ME, Spongy Castle [22] for Android or OpenSSL [23], just to mention some freely available APIs.

Here we assume that user credentials consist of a unique user name and a password, and that to each user (on a given device) is assigned a different encryption key, so that a successful authentication grants direct access to a token, the encryption key, and the data encrypted with it, but nothing else.

- *Straightforward solution*: The user is authenticated by computing a hash of the password and comparing it to the hash stored in clear on the phone. The same hash is also used as the user's encryption key (or to derive it). This allows separate encryption for each user (requirements 1 and 2), off-line authentication (requirement 3), but the authorisation mechanism prevents unauthorised access only as

[21]Bouncy Castle, [Last Accessed: June 2014], `http://www.bouncycastle.org/`

[22]Spongy Castle, [Last Accessed: November 2014], `http://rtyley.github.io/spongycastle/`

[23]OpenSSL, [Last Accessed: May 2015], `http://www.openssl.org/`

long as data are accessed through the application. If an attacker can copy the phone memory, then the stored hash can be used to decrypt the data directly, just as it can be used to recover the data if the password is lost (requirement 4).

- *Slightly better solution*: An easy improvement to the above solution is not to store the hash of the password. Authentication is then performed by verifying that the key derived from the password can correctly decrypt the data. This can be done by employing some kind of integrity check on the data, like appending a digest to the data before encryption, or adopting algorithm that can reliably detect errors caused by the use of a wrong key. But the safest and the cleanest way is to compute a keyed-hash message authentication code (HMAC). This approach would satisfy also requirement 6 if the key is generated with a reasonable number of iterations and therefore more time-consuming to break, but makes the recovery of the password and the data impossible unless a copy of the derived key or the password itself is stored somewhere and it is accessible through some other form of authentication.

- *Complete solution with a private server*: Recovering the password or the derived key is not a trivial problem. This information would have to be stored somewhere and the user should have an alternative authentication mechanism to access it, creating a circular problem. In any case, when the encryption key is derived from the password, even though we had a way to recover the data, a password change would require re-encrypting the whole storage since also the encryption key must be changed. This problem can be solved by creating an encryption key that is not directly derived from the password, but is instead encrypted separately with a password based derived key. Combining the previous solutions we would satisfy almost all our requirements, except for 4, 5 and 7, as long as the same password is used both for accessing the phone and the server. In fact changing a password on the server from one phone, would mean that the old password must still be used to login locally on other phones, creating potential synchronisation problems. Also, if one phone encryption is broken and the password recovered, the attacker could impersonate the user to the server and gain access to the rest of the system. One approach that could give a satisfactory solution from a security perspective, is to separate completely the local authentication from the server authentication. In this way, requirement 4 would be satisfied by storing a copy of the encryption key on the server, and if the mobile password is forgotten, the user could login on the server, retrieve the key and reset the mobile password. Requirement 7 would be satisfied since no trace of the server password could be found on the mobile phone, neither explicitly (the hash of the password), nor implicitly (some keys are still derived from the password, so it is still possible to guess the password, generate a key and verify whether it is correct just as the authentication mechanism does).

Figure 4.13 shows a solution that satisfies all given requirements based on the previous discussion. During login procedure, a user passes username and password. The login manager then computes the master key and the HMAC as it is shown in figure 4.13 and verifies against with the stored HMAC. If the authentication is successful, the login manager returns a token that the MDCS client uses to interact with the secure storage module for form and data related activities.

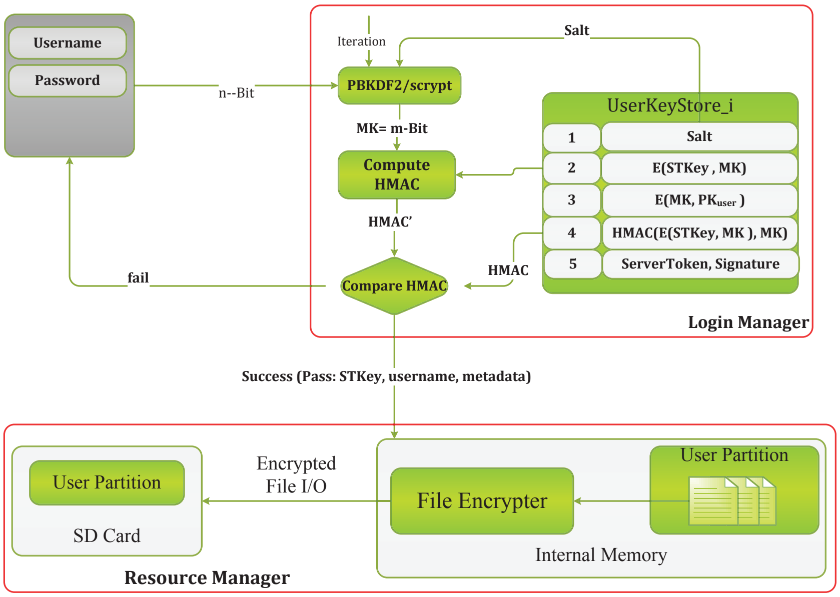


Figure 4.13: Local Authentication. The notation $E(Data, Key)$ means that the $Data$ is encrypted with Key.

If authentication fails enough number of times, the encryption key of the user will be deleted from the phone, hence forcing a recovery procedure involving remote authentication on the server. Also, keeping the keys in a store separate from the data (a keystore), and different user stores separated from each other, would add flexibility when implementing this solution.

4.6 Discussions and Conclusions

How difficult it is to break the data encryption depends on the strength of the user password and the key derivation algorithms used. There is not much that can be done to enforce a very strong password while keeping it secure. It is common knowledge that users will just write it down somewhere or forget it. A good compromise could be to require an alphanumerical password of a given minimum length, or a passphrase, while using algorithms that could slow down as much as possible a brute force attack, but without compromising usability. For this purpose, simple MD5 hashes or even a salted SHA-1 hash might not be enough, so we recommend using a password-based key derivation function such as PBKDF2 (Password-Based Key Derivation Function)

[24] and scrypt [25], with as many iterations as possible, based on the computation power of the phone.

Initially, Android was designed for a single physical user in mind. This design choice has been changed in favor of supporting multiple users per device in the later versions. As we previously discussed, however, the multi-user approach introduced by Android, and, therefore, its local authentication approach, does not meet the MDCS security requirements we identified. It is also worth mentioning that this feature was first introduced in Android 4.2 (Jelly Bean) for tablets and extended to smartphones only in Android 4.4 (Kitkat). This means Android devices prior to this version, do not provide any multi-user support.

As mentioned in [79], data exchange standard is one of among many factors that can overcome the barriers to integration and interoperability between MDC systems. Even though the good work of OpenRosa Consortium initiative to bring standardized data exchange and API are halted (for unknown reason), we also believe standard solutions are the only way to consolidate the fragmented MDC systems for better outcome. As of this writing, we are not aware of any group who does a similar work as the OpenRosa Consortium. We contributed this piece of work as starting point for a comprehensive and all-inclusive standard authentication solution. We identified the common functional and security requirements of most MDC systems and systematically addressed the local and remote authentication. We followed a standard, a well-tested protocol for remote authentication, namely SRP, and the local authentication is seamlessly integrated to two of the partners project, i.e. openXdata and ODK. The solution in the openXdata system is in production use, and we are extensively testing the authentication solution provided to the ODK system.

[24]Password-Based Cryptography Specification Version 2.0, [Last Accessed: January 2015], `http://www.ietf.org/rfc/rfc2898.txt`

[25]Scrypt, [Last Accessed: May 2015], `http://www.tarsnap.com/scrypt.html`

5 SecureMDC SecureStorage: A Secure Data Storage Module for MDCS

This Chapter is based on the candidate's previously published work at the International ACM Conference on Management of Emergent Digital EcoSystems (MEDES). The full paper is available at [46].

5.1 Introduction

Insecure data storage is probably the most concerning aspect of mobile devices, especially when a large amount of health related data is collected. The reason is that, unlike desktop computers, mobile devices are much more likely to be lost or stolen, or to be easily accessible in a short period of time if left unattended. It is enough that most application data is stored unprotected on the memory card, which can easily be taken out of the phone and read without problems on another device. This is why it is necessary always to encrypt sensitive data before storing them locally on a mobile device. With encryption, however, come also a lot of other unavoidable problems. How to generate strong cryptographic keys? How to store them in a secure way? Which algorithm to use? Does the device offer adequate support? Similar problems also arise on the server side, at least when the server is not under direct control, like for example in the cloud case.

It is not a surprise then that the Open Web Application Security Project (OWASP) ranked insecure data storage at second place in the top 10 2014 mobile security risks [94]. OWASP suggests to not to store data on mobile device as a cardinal rule to minimize risks, but as offline capability is one of our main requirements, providing a secure local storage becomes a major challenge in Mobile Data Collection Systems (MDCS). Besides, a secure storage is not only used to store and protect collected data, but also authentication information, key material, downloaded form definitions, certificates and more. This implies that a complete solution for secure storage must be very flexible and offer a wide spectrum of services to both the user, the application and the other modules of the framework. It must also provide means of recovering encrypted data if passwords or keys are lost, or means to destroy sensitive content if the device is compromised or stolen. For this reason this chapter, which will discuss all these different aspects of the secure storage solution can be considered the foundation of the framework and stands for most of our research activity.

Again, we focus on low budget mobile phones with low hardware and software specification and propose a solution that is flexible enough to be integrated into existing mobile client applications. The Java ME based solution has been extensively tested and incorprated into a production MDCS, namely openXdata. Thanks to its high-level and flexible design, we also managed to port it to Android based MDCS clients like ODK Collect, with a few minor adjustments.

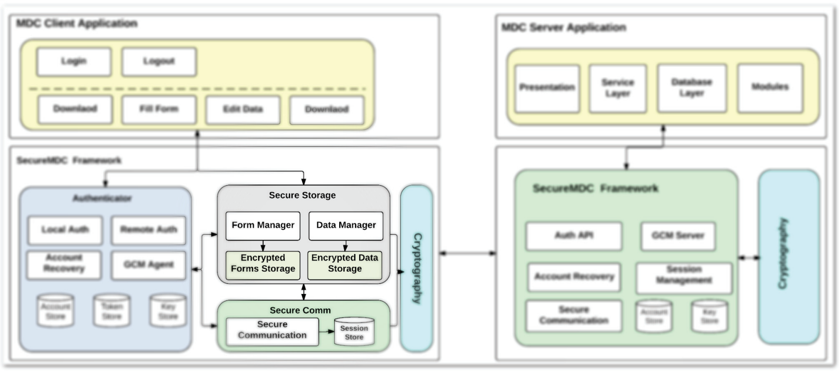


Figure 5.1: Secure Storage Module for Data Protection on the Device

The chapter starts with an assessment of the storage security of existing MDCS. What we found was that most basic security concerns had not been addressed in a satisfactory manner or had been completely ignored. It continues with a list of requirements that a secure storage should satisfy and an extensive review of existing solutions that can be used to provide a more secure storage. Finally, we present our solution. First its design, then its implementation and integration with productions MDCS and at the end its performance evaluation.

5.2 Review of Secure Storage Solutions in Existing MDCS

Our findings regarding the protection of the data while being stored on the mobile device, unlike the client-server communication, give reasons for serious concern. Among all MDCS we analyzed, only one actually encrypted the data on the phone, and even in that case, we found some weaknesses in the solution that was used. From the answers we got from DataDyne [26] and Mobenzi Researcher [85] customer support, we know that they do not encrypt the data stored on the phone, and their security actually relies mostly on two things. The first is that the Application Management System (AMS) on the phone prevents unauthorized applications from reading from another application memory store, and the other is that forms that are completely filled in, are automatically uploaded whenever a connection to the server is available. The AMS can work in some measure to sandbox applications on the phone, but the main problem with mobile phones, is that it is quite easy and straightforward to get a copy of all the data stored on the phone, and, once outside the phone, the data is completely unprotected [32]. In this case, uploading frequently can mitigate the problem, but in *remote* data collection, it is not unreasonable to assume that connection to the server might not be available even for days. Also, if the phone is lost or other sensitive data like user credentials are stored permanently on the phone, encryption should be in place to guarantee adequate protection.

To test the other clients, we tried ourselves to extract the data stored on the phone by means of a backup software, and analyzed the source code where available. We found that most clients store their data in clear, but with different formats. OpenXdata [17, 97] stores the serialized Java objects, but data elements are still recognizable, while CommCareHQ [22] and Magpi (formerly known as EpiSurveyor) [72] store all data as XML in clear text. Magpi, also, stores also passwords in clear text if one chooses to have the login form pre-filled. The only notable exception is Nokia Data Gathering. This client provides password-protected data encryption. However a quick look at the implementation [82] revealed that the encryption key is a direct MD5 hash of the password, and it is stored in clear on the phone. Hence, one can retrieve both the key and the encrypted data, hence rendering encryption useless.

Open Data Kit toolkit (ODK) has an optional security feature for storage and transmission protection, but still has some problems to be solved. The claim of this solution is to protect data while at rest on the device, during transmission and to protect data on the cloud. What ODK offers, is to download a public key attached to the form definition, and then upload each single form to the cloud encrypted with a random symmetric key, which is in turn encrypted with the aforementioned public key. This assures very strong data security both at rest on the client and on the server. Secure communication might even be superfluous since data is already encrypted. The downside is the overhead generated by all the encryption required for each form and the little flexibility the collector has since forms cannot be edited after being encrypted. Besides only the owner of the public key can decrypt and possibly redistribute the collected data. Apart from the usability limitations described above, this technique has some other disadvantages in terms of efficiency since it involves one expensive asymmetric encryption/decryption for every submission. By maintaining compatibility with the schema of ODK, we want to propose an encryption scheme that makes the system more usable in terms of flexibility, allowing many viewers, and efficiency, reducing the number of heavy operations, without lowering security. Together with this, we should also consider security problems since, as clearly stated on the ODK website [92]:

"There is nothing preventing a malicious adversary from the wholesale replacement of a finalized form with falsified data or the synthesis and submission of extra data."

5.3 Requirements and Risk Analysis

The main challenge is that, if multiple collectors share a device, it can no longer be considered private or personal to a user. Besides, most of the data collection might have to be done offline. From a security perspective, this translates into the following concerns:

- Data Confidentiality (encryption)
- Authorization (users can access only their own data)
- Data Availability and recovery
- Data Integrity and quality
- Breaking the storage encryption should not compromise the rest of the system

The followings are some of the threat in MDCS.

- Lost or stolen device
- Malware / malicious application
- Unattended device
- Bypassing client restrictions
- Malicious user on network

5.4 Available Solutions

This section covers existing solutions to provide the building blocks needed to put together a secure storage solution. We give only a brief overview for the Java ME platform as very little was available for it, while Android is the main focus of the section. We look at key management solutions first and full disc encryption. Then we discuss solutions to enforce security policies like data encryption, length and complexity of passwords and remote data wipe at the device level (MDM solutions) or at the application level (MAM solutions). Finally, we go through the available solutions that allow single applications to encrypt their data locally like SQLChipher, Bouncy Castle and Facebook Conceal APIs.

5.4.1 Java ME Storage Solution

With the exception of the optional JSR177 library [98] that provides cryptographic support, but is implemented only on very few handsets, Java ME does not offer any kind of storage/crypto library. Hence, any solution for encrypting stored data would have to be implemented from scratch building on some external cryptographic API. Most of the examples we found in the literature, and in the MDCS we reviewed at the beginning, are very straightforward. For instance, a key is created from the user password (or worse pin code as in [59, 118]) through some hash function, and it is used to directly or indirectly encrypt the data with symmetric encryption in the Java ME persistent storage called the Record Management System (RMS). Although the user password must be somehow involved in the process, we think that the storage scheme and the algorithms used should be more advanced, in order to protect data more effectively while allowing password recovery and multi-user management. Typical solutions for password recovery, like using the collectors' private email accounts or SMS to send new temporary passwords or activation links, are also more suitable for Web services than MDCS. An extensive review of possible solutions for secure storage in Java ME application can be found in [32].

5.4.2 Android Key Storage Solutions

With the advance in the capability of mobile phones to handle various tasks of mobile users in day to day activities, more sensitive information is being stored or transferred over phones. Account passwords, personal information about users, financially sensitive information like bank accounts or credit card numbers are some of the sensitive pieces of information that needs secure handling during transaction as well as in storage if the different applications store that on the device. Encryption of data is the obvious solution to protect data, and its security depends on the way encryption keys are handled. In Android phones, one can either use the Android key storage service or the Bouncy Castle library for key storage. The security of mobile devices highly depends on the mechanism to keep the integrity and confidentiality of keys. To that end, due consideration of securing keys is required and in this section we make an in-depth overview of the necessary security mechanisms and security solutions in use by Android devices.
The major security mechanisms to protect cryptographic keys in Android are:

1. Android Access Control: conventional access control on files which gives access to files per app.

2. Trusted Execution Environment (TEE) trusted zone technology provided for ARM based hardware that separates the environment into the normal world and secure world.

3. Password Protected Storage: a mechanism in which passwords are required to protect keys and the password can be stored in the app, or the user can supply it.

These mechanisms are used by varies solutions to protect storage of cryptographic keys in Android. The two commonly used Keystore types in Android are the Bouncy Castle cryptographic library for Java and Android Keystore. Bouncy Castle for Android is a limited version the standard library with the assumption that some of its functions are not required for Android devices [23]. When it comes to the Android Keystore, there is Keystore service that starts when the device boots moreover, it's possible to implement either hardware based secure storage, by communicating with device drivers directly from the service, or software based secure storage.

5.4.2.1 Android Keystore Implementation

The implementation of Android Keystore varies from device to device and even differs from the version of Android running on similar devices as well. One major difference between devices is the hardware support for trusted zone technology that is a specialization among devices based on ARM design. Before getting through the various implementations, we briefly an describe overview of the TrustedZone Technology.

5.4.2.2 TrustedZone Technology

The Android service depends on the type of phone in use which determines the usage of ARM Trustzone [7] features [23] ARM Trustzone features apply only to those devices whose processor is built as per processor design from ARM [23]. This is a hardware-based feature that allows the processor cores to run in two execution environments - normal world execution environment and secure world execution environment. The normal environment it refers to the devices' OS and normal apps and the secure environment refers to apps that handle sensitive data. With ARM trustzone feature, there is a clear boundary between processes from the normal world and secure world environments. That is made possible by adding a security bit that tells peripherals to which environment the application they are talking to belongs.
Such separation is also implemented by the NS-bit (Non-Secure-bit) in the Secure Configuration Register of the processor that indicates the environment the processor is working on. NS-bit can only get its value set by a trusted component called Trusted Monitor. A value of "0" implies the processor is operating in the secure world and "1" implies normal world [23].
The security features provided by TrustedZone Technology are hardware-based, but that does not mean that it can address all security concerns. For instance context switching between the normal and secure world is implemented by the software running in the secure world and data communication between the two worlds is implemented by the software running in both worlds. This hardware-based separation feature somehow creates a virtual environment that make running two OSs possible one normal world OS and another secure world OS. The environment for the secure world is referred as Trusted Execution Environment (TEE) [48] and applications running on it are called trustlets. An important point to note about trustlets is that there is a possibility that they might be used by untrusted user and the TEE has to make sure there won't be any data leak even among trustlets [23].

5.4.2.3 Keystore Implementation Types

The paper [23] identifies the below mentioned five Keystore implementation types in Android and make a detailed analysis.

1. Bouncy Castle using a stored password without TEE
2. Bouncy Castle using a user provided password without TEE
3. AndroidKeyStore using the TEE on Qualcomm devices
4. AndroidKeyStore using the TEE on TI devices

5. AndroidKeyStore using software fallback without TEE and user provided password

N.B: The types mentioned in 3 and 4 vary because of the devices which make the implementation of the same solution differ as per the device. To analyze the above-mentioned Keystore implementations, the research has taken two important parameters into consideration:

1. Important security requirements
2. Attacker models

Security Requirements Three security requirements are identified that are found to be important when analyzing key storage solutions:

1. App-binding: if the key can be shared among apps or a single app will solely use it.
2. Device-binding: if the key can only be used on certain device
3. User-consent required: if the key can be used without the consent of the user or there always have to be an explicit user consent

Attacker Models Knowing the possible attacks on a system helps to evaluate if a recommended security solution can address those issues or not and will also guide in the design and implementation of new solutions. To that end, the paper identified three major attack models on secure key storage.

1. Malicious app attacker - usage of an installed app to gain access to secure key storage.
2. Root attacker - an attacker with root credentials which the attacker might get using exploits or with the ability to run an application with root permissions.
3. Intercepting root attacker - a root attacker with an additional capability of intercepting or capturing user input or inspecting the memory of the device.

After conducting some experiments, Cooijmans T. et. al [23] draws the following conclusions:

- If TEE is present on the device, the Keystore on Android device provides device binding against root attacker. However, the combination of TEE and the Keystore does not guarantee app binding in the presence of root attacker.
- Interestingly, the software backed Bouncy Castle key storage provides *"stronger"* security guarantees than the hardware-backed Android KeyStore using the TEE. This is because to unlock the key storage, the Bouncy Castle requires user-provided password, but the Android Keystore does not.

The software or hardware backed Keystore is a promising security feature for data protection guarantee on the device. In our implementation, we leveraged Android provided Keystore in a hacky way. Since, the Android Keystore does not support user-provided password, we created a secure layer that perform the encryption and decryption based on user-provided password at the application layer before it is written or read to/from the Android Keystore file respectively.

5.4.3 Android Full Disk Encryption

Since Android 3.0 (aka Honeycomb), Android introduced Full Disk Encryption (FDE) together with device administrator policies (a core component to Mobile Device Management, see section 5.4.4) to address security issues within Android platform. The security enhancement is mainly motivated by pushing Android technology into the Enterprise world. [33] FDE is a transparent encryption method based on dm-crypt, which is a Linux kernel feature that works at the block device layer (block devices are hardware devices distinguished by the random access of fixed-size chunks of data, called blocks)[1]. According to the Android Compatibility Program Definition, Original equipment manufacturer (OEMs) are obliged to support FDE if the device has lockscreen. Consequently, all subsequent release of Android device provide FDE. The problem is that even if FDE has been available in most devices, FDE has to be enabled by a user or by device policy of managed devices. There was a recent effort from Google and public announcement to enable FDE by default in all FDE supporting devices upon when the device is booted the first time. However, later, Google made a statement saying that due to FDE performance issue, OEMs are not required to make FDE default.[2] It is also worth mentioning that Google products like Nexus provides FDE by default.

FDE implementation may vary from one OEM to the others. For instance, some device may support Hardware-backed key storage component such as TPM or others rely on software-based key protection solution using pass-phrase. FDE is expected to encrypt every piece of data writing to the disc transparently. If the key management is based on the software, FDE key generation, and encryption method is shown in figure 5.2

The disk encryption key (a.k.a master key) is encrypted with another 128bit AES key (aka Key Encryption Key, KEK) derived from a user-supplied password. In Android 3.0 through 4.3, the key derivation function used was PBKDF2 with 2,000 iterations and a 128bit random salt value. The resulting encrypted master key and the salt are stored, along with other metadata like the number of failed decryption attempts, in the footer structure occupying the last 16KB of the encrypted partition, called a crypto footer. Storing an encrypted key on disk instead of using a key derived from the user-supplied password directly allows for changing the decryption password quickly, because the only things that need to be re-encrypted with the key derived from the new password is the master key (16 bytes) [33].

In order to make it harder to brute-force disk encryption passwords, Android 4.4

[1]The Block I/O Layer, [Last Accessed: March 2015], `http://www.makelinux.net/books/lkd2/ch13`, Android FDE, [Last Accessed: March 2015], `https://source.android.com/devices/tech/security/encryption/`

[2]Engadget - Google will not force Android encryption by default,[Last Accessed: April 2015], `http://www.engadget.com/2015/03/02/android-lollipop-automatic-encryption/`

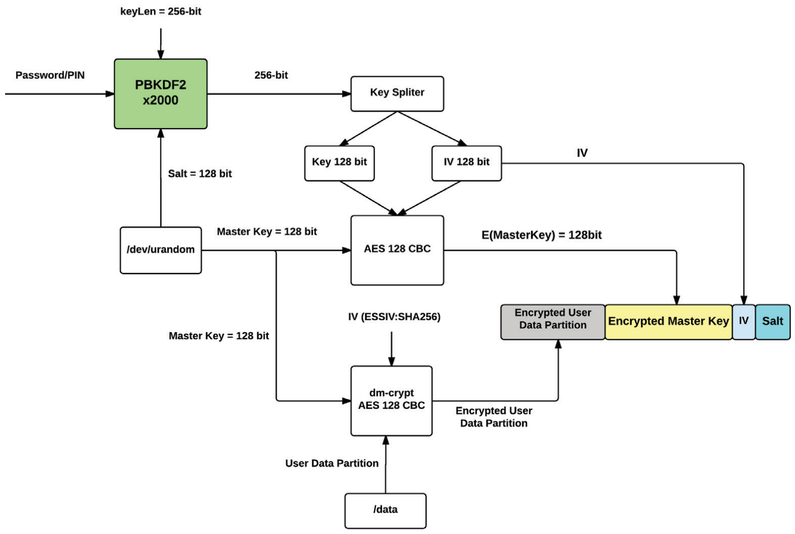


Figure 5.2: Android Full Disc Encryption Diagram

introduced support for a new key derivation function called scrypt [33, 101]. Scrypt employs a key derivation algorithm specifically designed to require large amount of memory, as well as multiple iterations (such an algorithm is called memory hard). This makes it harder to mount brute-force attacks on specialized hardware such as ASICs or GPUs, which typically operate with limited amount of memory. Android 4.4 devices automatically update the key derivation algorithm in the crypto footer from PBKDF2 to scrypt and re-encrypt the master key using a scrypt -drived encryption key [33]. As it is also stated in [33], brute-forcing PBKDF2 is almost 50 times cheaper (that is, faster) compared to scrypt.

However, Android disk encryption only protects data at rest; that is when the device is turned off. It does not protect from a malicious application running on the device.[33].

5.4.4 Mobile Device Management System (MDM)

Mobile Device Management (MDM) is software tool for managing mobile devices. It is introduced to administer mobile devices used to access corporate services and data. Several organizations have been using the popular Blackberry's MDM enterprise solution[3] for restricting and controlling corporate data access using mobile devices. Other mobile platform providers including Apple and Google have joined and made their enterprise solution available to the MDM vendors. Unlike Blackberry, Apple works with

[3]Blackberry Enterprise Solution, [Last Accessed: May 2015], http://us.blackberry.com/enterprise/solutions/emm.html

third-party MDM providers such as Airwatch[4], Good Technology[5], XenMobile by Citrix[6], and so on. Apple also works with organizations who are interested to develop their own MDM solution and distribute the application in-house. For this purpose, Apple requires new Enterprise Account registration with a fee (upon writing this $299). Apple made device administration API very flexible with several features.

On the other hand, Google provides a sort of device administration API through Android with limited features compared to Apple MDM APIs. Android device administration APIs are available freely for developers with a normal developer account (which costs USD25 upon first-time registration). Even if the mainline Android platform lacks a comprehensive MDM APIs, an OEMs (Original Equipment Manufacture) like Samsung customized the mainline Android platform and built a secure enterprise solution for organizations who are interested a device and application management solution for their services. Samsung Knox [114] is an enterprise solution tailored for Samsung Android devices and certified by NSA [113]. Samsung has proprietary API for device administration and works with third party MDM vendors like Apple does. Samsung security-related API have not been yet a part of the stock Android, and Samsung charges for its API usage. Since there exist several Android device OEMs, focusing on a solution that targets a specific brand, like Knox, does not work in the context of MDCS. We focus on a solution that works for all Android devices. In the next section, we explore how the MDM works and discuss its main components. Later, we discuss its pros and cons compared with our security requirements.

Before we delve into the details, it is worth mentioning the challenges in understanding how the MDM API works, its underlying protocol, and workflow. Apple MDM does not provide public documentation how the system works unless we registered for enterprise account that requires an existing company account and fee. Android, on the other hand, provides little documentation on how to create MDM solution that is managed remotely. This part of the work is inspired by "The iOS MDM Protocol" by David Schuetz [27], who have done an outstanding work in unpacking the underlying iOS MDM protocol.

5.4.4.1 Android MDM Components

In general, Android based MDM system consists of the following three components (shown in figure 5.4):

1. MDM Agent: an application installed on a managed mobile device such as smartphones, and tablets.

2. MDM Server: Device and Policy Management Application running on the Server.

3. Message Delivery Channel: A downstream or upstream communication channel by which the server and client reach each other in order to exchange messages and

[4]Airwatch Enterprise Mobility Management Platform, [Last Accessed: June 2015], `http://www.air-watch.com/`

[5]Good Technology - The Secure Mobile Platform for Business, [Last Accessed: June 2015], `https://www1.good.com/`

[6]Citrix XenMobile: Enterprise Mobility Management solution, [Last Accessed: April 2015], and `https://no.citrix.com/products/xenmobile/overview.html`

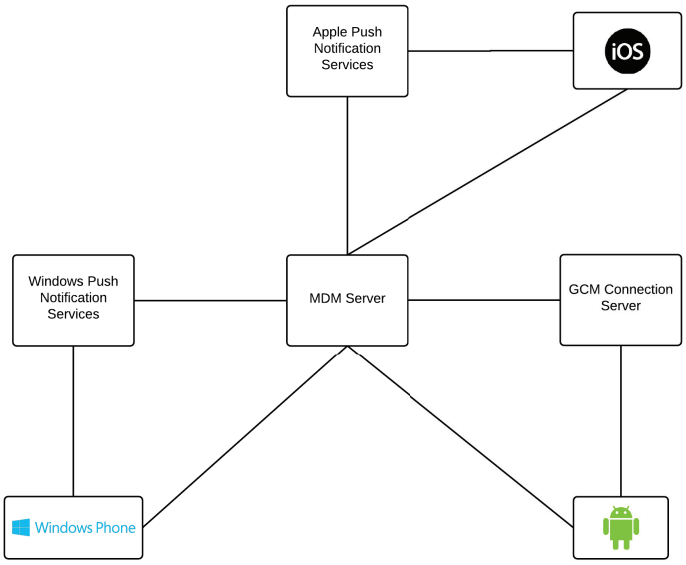

Apple Push Notification Services
iOS
Windows Push Notification Services
MDM Server
GCM Connection Server
Windows Phone

Figure 5.3: Mobile Device Management (MDM) with Push Notification Services

execute commands (aka push notification service such as Google Cloud Messaging (GCM))

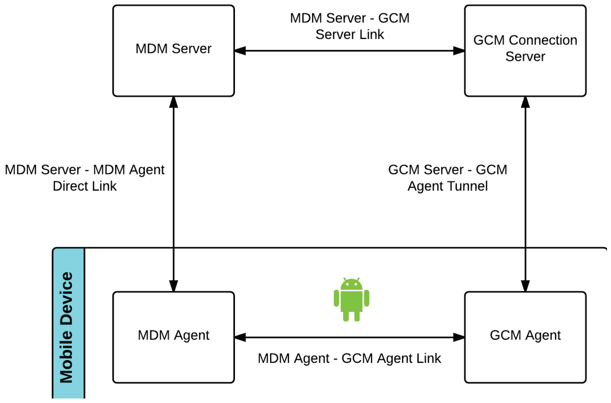

MDM Server
MDM Server - GCM Server Link
GCM Connection Server
MDM Server - MDM Agent Direct Link
GCM Server - GCM Agent Tunnel
Mobile Device
MDM Agent
MDM Agent - GCM Agent Link
GCM Agent

Figure 5.4: Android based MDM Components

5.4.4.2 MDM Agent

The Android MDM agent is based on a device administration API. It is introduced in Android 2.2. The administration API is designed to enforces a set of policies on organization or users owned mobile devices and restrict organization services access using mobile devices according to the enforced policies. The followings are some of the policies that administration API provides:

- Password related policies such as minimum length, alphanumeric, numeric, or complex password, timeout, and maximum failed attempts which lead to device wipe and factory reset
- Enable Storage Encryption (introduced on Android 3.0)
- Disable camera (introduced on Android 3.0)
- Password recovery through remote account reset
- Device locking remotely
- Device data wiping remotely (factory reset)

These policies are declared in a simple XML file as shown in Listing 5.1.

```xml
<device-admin
  ↪ xmlns:android="http://schemas.android.com/apk/res/android">
  <uses-policies>
    <limit-password />
    <watch-login />
    <reset-password />
    <force-lock />
    <wipe-data />
    <expire-password />
    <encrypted-storage />
    <disable-camera />
  </uses-policies>
</device-admin>
```

Listing 5.1: Example: Android Device Administration Policies

Once the user installed and activate the MDM agent, the Android system enforces the policies listed in the XML file. The Device Policy Manager within Android platform takes care of enforcing and managing the policies. For instance, listing 5.2 shows how the Policy Manager enforces the policies.
Once policies are in place, the system manages the policy, allows policy changes at runtime, and notifying the MDM agent when there are policy related events through a set of callbacks. Accordingly, the MDM agent can be implemented to execute some tasks based on the events from the system.

```
//System Device Policy Manager class - responsible for managing the
  ↪ policies
DevicePolicyManager mDPM;
//The Device Administation component is responsible for listening
  ↪ policies related events from the system
ComponentName mDeviceAdminSample;
...
//Maximum period of user inactivity that can occur before the device
  ↪ locks
long timeMs = 1000L*Long.parseLong(mTimeout.getText().toString());

mDPM.setMaximumTimeToLock(mDeviceAdminSample, timeMs);
```

Listing 5.2: Example: Policy enforcement using Android Device Policy Manager

5.4.4.3 Message/Command Exchanging Channel

Android device administration API works on local context, in other word, Android does not provide remote MDM management or provisioning solution. Even if remote device locking or data wiping requires remote command execution, Android does not provide a protocol to execute those commands remotely. Instead, the responsibility for finding a remote management solution is left to MDM implementers.

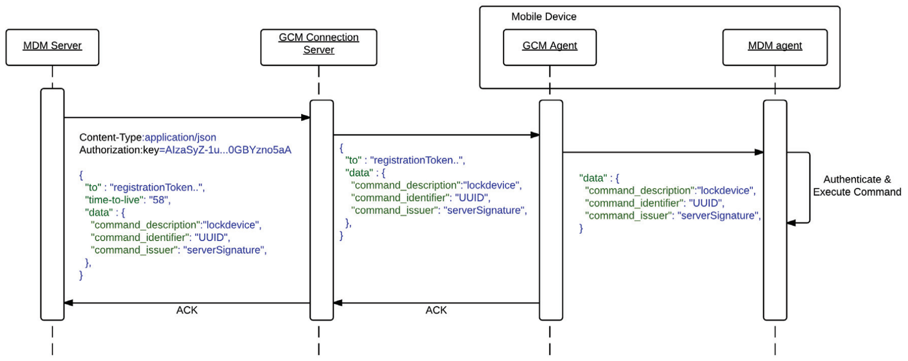
As a result, several MDM vendors are leveraging Google Cloud Messaging (GCM) to synchronize policies and send commands data from the server to clients through GCM infrastructure. Even if the GCM infrastructure is developed for notification message delivery or send-to-sync messages, GCM can be utilized to exchange remote commands as data and the client interprets the data and execute them accordingly. These commands are very sensitive and critical and securing the exchanging channel is imperative.

The GCM infrastructure consists of GCM server and GCM agent (aka GCM framework or client). The GCM server is a cloud based services which connect the MDM server to the MDM agent via the GCM agent on the device. The GCM client is a system level application running on the device that works as an agent to GCM server. The GCM client manages to establish a secure tunnel with the GCM server and manages bi-directional messages delivery from MDM agent to MDM server and vice versa.

Each MDM agent application instance installed on the device are required to register with GCM server before sending or receiving messages via the GCM infrastructure. This step requires getting a senderID (aka projectID) and API key from Google API console manually and setting the senderID in MDM agent application. The API Key saved on the MDM server that provides the MDM server authorized access to Google services.

Since Google Cloud Messaging 3.0 (announced at Google I/O 2015[7]), when the user install the MDM agent on the device, it triggers the GCM connection server to

[7] Google I/O - 2015, [Last Accessed: June 2015], https://events.google.com/io2015/



```
Content-Type:application/json
Authorization:key=AIzaSyZ-1u...0GBYzno5aA

{
  "to" : "registrationToken..",
  "time-to-live": "58",
  "data" : {
    "command_description":"lockdevice",
    "command_identifier": "UUID",
    "command_issuer": "serverSignature",
  },
}
```

```
{
  "to" : "registrationToken..",
  "data" : {
    "command_description":"lockdevice",
    "command_identifier": "UUID",
    "command_issuer": "serverSignature",
  },
}
```

```
"data" : {
  "command_description":"lockdevice",
  "command_identifier": "UUID",
  "command_issuer": "serverSignature",
}
```

Figure 5.5: Example of MDM remote Command Execution using GCM

generate an instance ID of the installed MDM agent instance through GCM agent. This step requires connectivity, and if the application is installed offline, the instance ID is generated when the device comes online. We have not found public documentation on how GCM agent and the GCM server authenticated each other or how the instance ID is generated. However, on the Google Instance ID page [8], it is stated that there is public/private key cryptography involved together with instance ID. The private key is stored on the device while the public key is registered with the GCM connection server. This may imply that during the instance ID generation, the GCM agent on the client generates the public/private key pairs and forward the public key together with other parameters such as AndroidID, MDM agent application package name, application version, application signature, and so on. The GCM connection server returns the instance ID that is related to the public key and the parameters sent from the GCM agent.

The instance ID consists of two parts: a unique identifier (username) and a security token. Once the GCM agent gets hold of the instance ID from the GCM connection server, it can return the unique identifier (ID) if the MDM agent requests. However, the ID does not qualify for sending or receiving messages via GCM infrastructure. Instead, the MDM agent needs a security token (hereafter called registration token) which is the second part of the Instance ID. The security token is generated dynamically upon request and registered on the GCM connection server. There is no indicative documentation on where the security token is generated. However, we assumed that the token generation is taking place on the device since the private key is stored locally and the token is synced with the GCM connection server in order to allow third party application to send messages to the MDM agent such as MDM server. The registration token is compared as OAuth token except the fact that OAuth token and registration token represent a delegation on behalf of a user and application respectively.

Next, the MDM agent forwards the security token to the MDM server in a secure tunnel. The MDM server uses this token together with the API Key for sending messaging to the MDM agent through GCM connection server. The message can carry MDM related commands, data, or simple notification.

[8]GCM Instance ID, [Last Accessed: June 2015], developers.google.com/instance-id

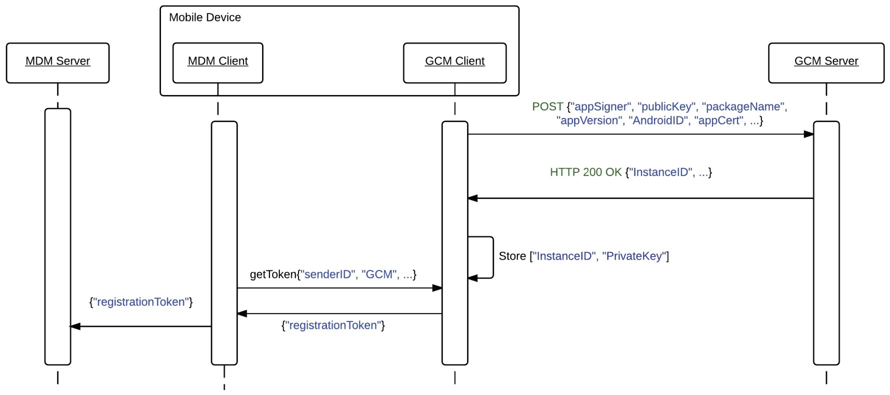

Mobile Device
MDM Server
MDM Client
GCM Client
GCM Server
POST {"appSigner", "publicKey", "packageName", "appVersion", "AndroidID", "appCert", ...}
HTTP 200 OK {"InstanceID", ...}
Store ["InstanceID", "PrivateKey"]
getToken{"senderID", "GCM", ...}
{"registrationToken"}
{"registrationToken"}

Figure 5.6: Enabling GCM for Android MDM through Registration

Figure 5.7 shows how the MDM server sends a query to the GCM connection server and get a set of status information about the remote client. The MDM server can query the following information using the registration token.

- Application name: the MDM agent name such as package name.
- Application version
- Connection status such as when is the last time the device has been seen online, the type of connection the device is using such as WiFi or cellular network.
- Attestation Status: notify the server if the MDM agent instance is running on a rooted device.

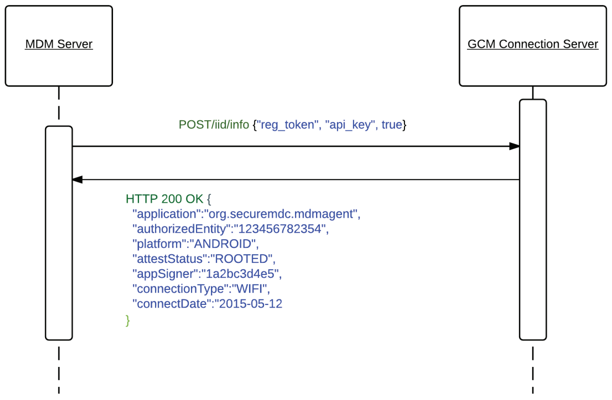
In summary, the MDM solution have the following pros and cons as we compared it with our security requirements:

Pros

- Manage installed applications on the device remotely.
- Monitor the user activities on the phone.
- Enforce organization policies.
- Remote wipe, detect rooted device

Cons

- Very little control over how the device administration framework executes the commands. Implementation is OS dependent.
- Multi-user support is not possible since the device is locked using a single user account.



```
HTTP 200 OK {
  "application":"org.securemdc.mdmagent",
  "authorizedEntity":"123456782354",
  "platform":"ANDROID",
  "attestStatus":"ROOTED",
  "appSigner":"1a2bc3d4e5",
  "connectionType":"WIFI",
  "connectDate":"2015-05-12
}
```

Figure 5.7: MDM Server query the GCM Connection Server

- Privacy concerns: if users are using their device, they might not be interested locking down their device and have a complete control.
- Selective encryption or wipe is not possible.
- MDM encryption is based on device disk encryption that may have performance issue.
- Android disk encryption always requires passphrase to unlock the device which has usability issues.
- There is no standard remote device management protocol or provisioning solution.
- Android device administration API lacks features when it is compared with another platform such as Apple MDM framework.
- Mostly used by corporates and MDM vendors does not provide documentation on how the MDM is developed or used APIs (mostly proprietary).
- some research have shown the weakness of Android disk encryption.

Some of these MDM drawbacks can be alleviated using Mobile Application Management (MAM) which discussed in the next section.

5.4.5 Mobile Application Management System

Bring Your Own Device (BYOD) is one of the main factors for having Mobile Application Management system as a solution. The idea behind BYOD is to allow employees within an organization to use their mobile device to access organization restricted services and resources on their device. Since most employees already have their own mobile device, it is convenient to provide access through their phone without giving organization owned phones. MAM is designed to run organization applications within a secure container of employee device. The secure container makes a partition between

users private data and corporate data and works only within the context of corporate applications and data. In theory, using secure container, users can practice whatever they want on their device without being monitored or losing control over their device. MAM is based on the concept of running applications in a contained and monitored environment which is called secure container. The secure container is a crucial part of MAM architecture and we described as follows.
These are some of the secure container features:

- Data storage protection performed at the application level
- Does not rely on OS security features being activated such as pass-phrase
- Allows security policies to be enforced at the application level

Airwatch, Good Technology, Maas360[9], XenMobile by Citrix are some of the MAM vendors in the market. We are not aware of any open source MAM solution or a public research on MAM with the exception of a research conducted by by Ron Gutierrez [10]. It is hard to get hands-on experience on how MAM works and do some concrete research because it is difficult to get documentation or do research on MAM products. Most of MAM vendors allows a trial period, but they require a lengthy sign-up process with a corporate email address and other information. We had tried to get a trial version using our academic email account, but they did not respond to us. With the help of Ron Gutierrez talk and other publicly available related information, we briefly present the secure container (a.k.a containerization) as follows.

The secure container is made through application wrapping as shown in figure 5.8. Secure functionalities are injected into existing applications using wrapping utility. A secure layer is enforced at the application layer and provides data at rest protection, authentication before data access, and policy enforcement. The developer is not required to make code changes in order secure the existing application if the wrapping toolkit is applied. As an alternative, it is also possible to incorporate the secure container in the application development lifecycle using SDK provided by MAM vendors such as Good Technology. If the developer decided to use the SDK, it may require a great deal of understanding of the API and underlying working principles before use. However, the SDK approach provides more flexibility and features. In general, the use of wrapping toolkit can be convenient to the developers with no understanding how the secure container works or security API.
The application wrapping process may use method swizzling, a common method for code injection in the iOS application, or code injection in Android using reverse engineering tools. There are some open source tools used to manipulate Android APK files such as ASMDEX Library[11]. It is possible to inject secure code into the main application. Thus, data security components including secure storage, authentication, secure communication or any other security components can be inserted smoothly into the application. Here are some of the list of the secure container data at rest protection principles [111]:

[9]Maas360 MAM Solution, [Last Accessed: May 2015], `http://www.maas360.com/`

[10]Contain Yourself: Building Secure Containers for Mobile Device, [Last Accessed: April 2015], `https://www.youtube.com/watch?v=siVS2jmPABM`

[11]ASMDEX: a bytecode manipulation library, [Last Accessed: April 2015], `http://asm.ow2.org/asmdex-index.html`

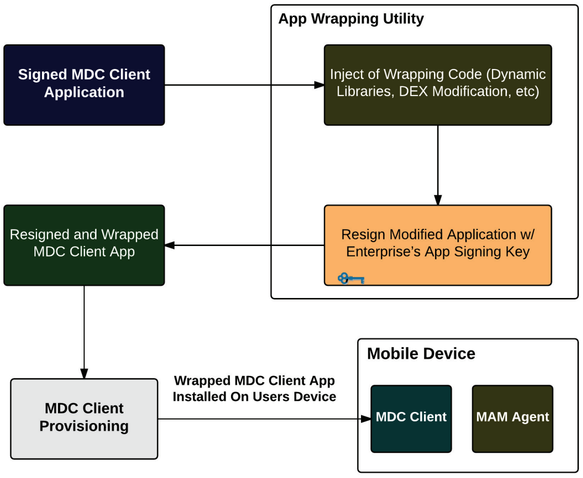

App Wrapping Utility
Signed MDC Client Application
Inject of Wrapping Code (Dynamic Libraries, DEX Modification, etc)
Resigned and Wrapped MDC Client App
Resign Modified Application w/ Enterprise's App Signing Key
Mobile Device
MDC Client Provisioning
Wrapped MDC Client App Installed On Users Device
MDC Client
MAM Agent

Figure 5.8: Mobile Application Management (MAM): Application Wrapping Processes

- All data stored by the application must be encrypted seamlessly.
- Strength of crypto cannot rely on any device policies.
- Crypto keys must be retrieved upon successful authentication.

In summary, even though MAM is intended for enterprises, it is worth looking into since it has several advantages especially simplifying secure solution integration into an existing app. It may require significant work and collaboration to develop MAM solution to the open source community. However, the core functionalities such as secure data storage, key management, and user authentication are designed in a similar way as the MAM vendors. Therefore, these security components can be re-used to build a comprehensive MAM solution. Many of the vendors use open source or custom crypto library, IPC mechanisms, and wrapping a secure API around existing libraries such as Java I/O API or Apache libraries. Often, security is an afterthought and MAM understand that limitation and provides seamless integration and policy enforcement. However, it is highly recommended to incorporate security in the application during software development life cycle.

5.4.6 SQLCipher: Secure SQLite Database

SQLCipher is an elegant secure solution for data stored in the SQLite database. SQLCipher is a security extension of the lightweight SQLite database with a transparent data protection mechanism using symmetric key encryption. The default implementation uses OpenSSL libcrypto and provides of AES 256-bit block cipher encryption with CBC mode. The SQLCipher team focuses on securing user data where part of the key material is provided by the user [12]. Figure 5.9 shows how the SQLCipher changes the insecure SQLite database into a secure database by introducing a transparent security layer. The security layer basically intercepts the common SQLite write operation and apply data encryption on the fly before it is written to the database and decrypt when there is read SQLite operation. Apart from passing the encryption key material such as passphrase, the mobile application does not know the underlying transparent security layer and uses the same SQLite interfaces to interact with the database. However, under the hood, the SQLCipher encrypts the entire database file. The data in the database are divided into chunks called pages and encryption/decryption is applied on these pages. The default SQLCipher page size is 1024bytes. When the database is created for the first time, it is assigned a randomly generated database salt value with a 256bit size. Each database file has single salt value assigned and its value is written into the first 256bit offset of the database file. A Message Authentication Code (MAC) and random initialization vector (IV) are written at the end of each encrypted page as shown in the figure.

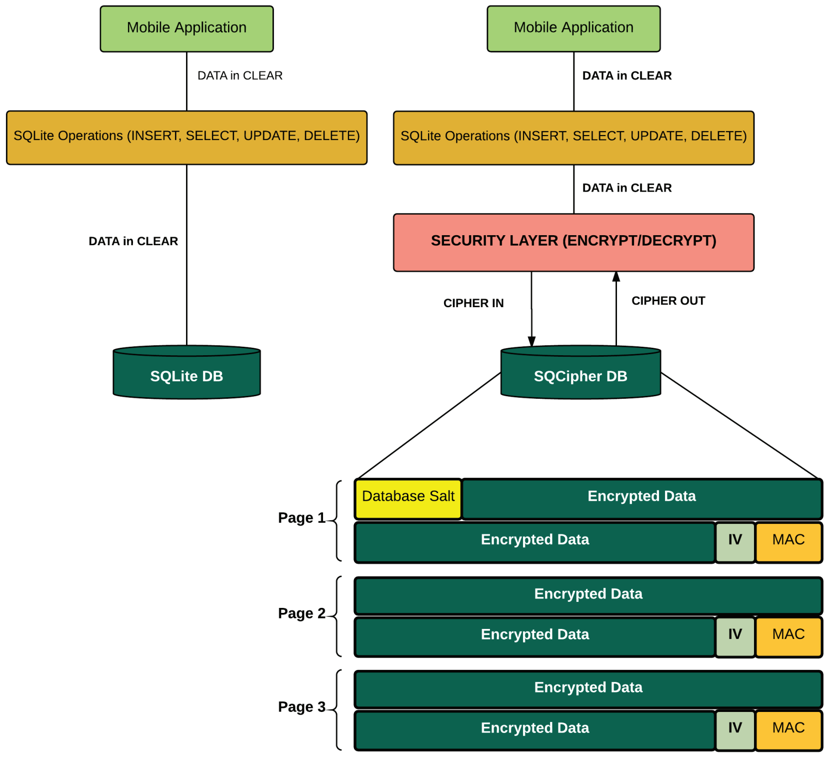


Figure 5.9: SQLite vs SQLCipher

[12]SQLCipher, [Last Accessed: April 2015], https://www.zetetic.net/sqlcipher/

SQLCipher uses PKCS #5 v2.0, PBKDF2 (Password-Based Key Derivation Function 2) to derive the master encryption key. A mobile application is responsible for managing and passing the key material, such as a passphrase, to the key derivative function. In addition to the passphrase, the key derivative function uses a 256 bits randomly generated database salt and an iteration value (a default value of 64,000 for iOS devices) and derive the master key. Next, the HMAC key is derived using the same PBKDF2 algorithm using the master key, database salt, and an iteration value of 2. This is done to get a different key for message integrity protection and data encryption/decryption key. Figure 5.10 shows how the data is encrypted and the MAC value is computed afterwards.

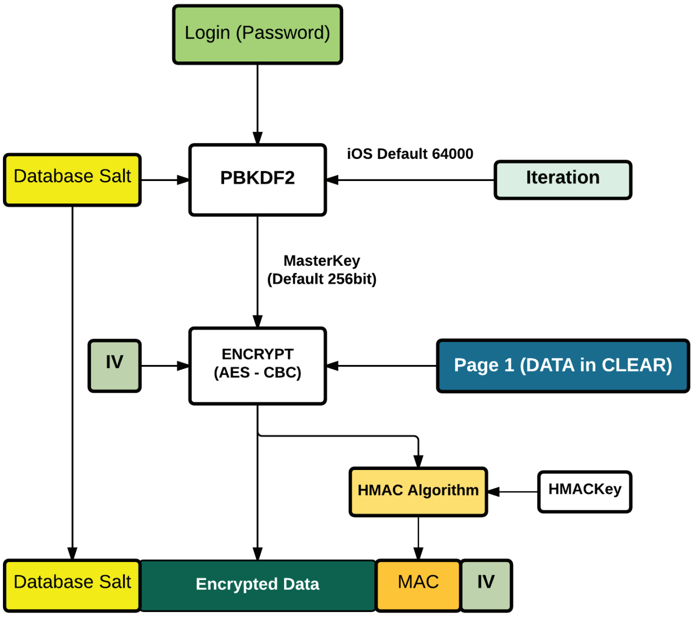


Figure 5.10: SQLCipher Encryption Flow Diagram

When there is an SQLite read operation, data is decrypted page by page as shown in figure 5.11. The MAC is computed using user input passphrase and checked with the MAC value in the database. If these values are equal, it proceeds with decrypting that specific page. This means that the MAC is used to check the integrity of the encrypted data and also used to verify the input passphrase is correct.

To summarize, the SQLCipher has been customized for several platforms including Android, iOS, Windows phone and many others. The SQLCipher community edition is a free and open source segment of SQLCipher, which released under BSD-style license with minimal restriction. The SQLCipher also released in commercial edition and enterprise edition. A set of important features is missing from the free edition that are incorporated in the commercial edition. Apart from these features and flexibility,

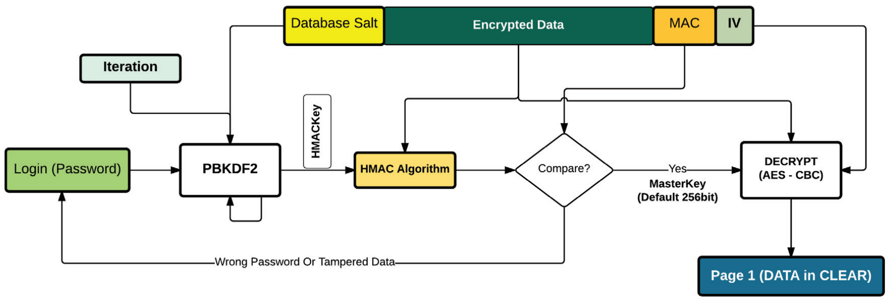


Figure 5.11: SQLCipher Decryption Flow Diagram

all editions are designed based on the same principles. Therefore, we identified the following pros and cons of SQLCipher:

Pros

- Requires few lines of code to integrate SQLCipher and provide full database encryption.
- Perform well since SQLCipher is written at the native layer.
- Data is encrypted page by page and decryption also performed page by page which means the entire database is not decrypted when some data is needed

Cons

- The master key derived from the key derivation function is used directly to encrypt the actual database. If the passphrase is compromised, changing the master key with a new one requires decrypting the entire database and re-encrypt with a new key. This can be solved by decoupling the master key with the actual data that need to be encrypted. The data is encrypted by a randomly generated data encryption key (DEK) and the master key encrypt the DEK. In that way, if a user wants to change a passphrase, the SQLCipher changes re-encrypt the DEK with the new master key.
- Despite a key management is a critical component of a secure storage solution, SQLCipher does not provide any. The application is, therefore, responsible for implementing any possible solution for managing the keys and policy enforcement for passphrase selection if the application allows a user to set a new passphrase on the device.
- A number of mobile data collection applications, do not store multimedia data such as image, video, and audio in SQLite database. Instead, these types files are stored in the file system and keep the pointer to these files location in the SQLite database. Hence, with SQLCipher approach, these multimedia are left unencrypted. The application needs to find another secure solution to protect the multimedia files which is additional work for the application provider, and the two solutions might have compatibility issues.

- SQLCipher does not provide data or account recovery procedures. The application should implement data and account recovery procedures on top of the SQLCipher.
- SQLCipher a performance impact overhead of 5 to 35 percent of using the standard SQLite database. The performance impact can be reduced through the use of database indexing. If there is no index in place, a normal select statement needs to search the entire encrypted database. In other words, every page needs to be decrypted and checked sequentially. However, the data leakage when the indexing is used not addressed or researched well. It is also mentioned that, if there are duplication in the database, the performance impact might be higher due to the fact that reading or updating a specific row data has to performed several times. Therefore, the data in the database should be normalized well. Furthermore, the use of transactions is also highly recommended to reduce the performance impact.

5.4.7 Conseal - Facebook Secure Storage API

Conseal a secure storage API is a great piece of contribution from Facebook to the open source community. Taking old Android version mobile devices constraints such as low memory, battery usage, and slow computing capability, Conseal API aims to encrypt and decrypt large files on the disk in a fast and efficient manner. The API incorporates default implementation for encryption and decryption using the OpenSSL crypto library. Since the OpenSSL crypto library that shipped with Android platform does not include AES with GCM mode, Conceal ships with this and other selective crypto algorithms from the standard OpenSSL library. Figure 5.12 benchmarks show how the Conceal API outperform Android Java and Bouncy Castle on encryption, decryption and MAC computation [13]

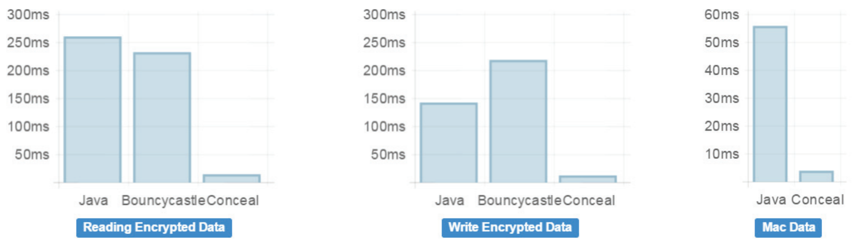


Figure 5.12: Performace Comparison between Android Java, Bouncy Castle, and Conceal API

Since Android 4.4 (aka KitKat), OpenSSL has become the default crypto provider for Android devices and several improvement has been made to the library. OpenSSL is implemented at the native layer, and it is accessible from Java layer through Java Native Interface (JNI). The fact that OpenSSL is native, it makes a significant performance difference. However, devices prior to Android 4.4 ship their own crypto library with

[13]Conceal API, [Last Accessed: April 2015], https://facebook.github.io/conceal/

applications such as Conceal API or they use Android Built-in Bouncy Castle library. Therefore, Conceal API is still gain advantages on older devices. Since our focus is also resource-constrained devices, and Conceal API is released under BSD license, instead of re-inventing the wheel, the secure storage module leverages the Conceal API for multimedia files protection. However, the secure storage module only loads the Conceal API for devices prior to 4.4, otherwise, it uses the built-in OpenSSL library.

5.4.8 Android Internal Storage vs External Storage

Unless otherwise specified, when data stored in shared preferences, internal storage, and SQLite database, the data is considered private, and Android platform provides some level of protection. By default, when data is private to the application, other application are not allowed to access it, and the data are removed when the user uninstalls the application. Despite, system level protection is provided to the application data, there are potential ways of getting into the application data and access it. Android platform does not provide any level of data protection while data is stored in the external storage. Sometimes it is confusing when some devices such as Nexus shipped without removable or expandable external storage. Basically, the external storage can be removable such as SD-card or non-removable (internal) storage. In conclusion, when sensitive data stored internally or externally, a comprehensive data protection should always be in place.

5.5 Proposed Solution

In this section, we discuss how our secure storage module provides a secure storage to the MDCS client. First, we present the common services provided by MDCS applications without secure solution and discuss how we secure these services. When the user authentication is a success, the user lands in a dashboard with a set of services as showing in figure 5.13. These services include form downloading, form filling, data editing, data or form deleting, searching, and last but not least data uploading or submission. In a typical MDCS client, form downloading, and data submission services involve client-server communication whereas the others such as form filling, editing, and deleting are local operations.

The MDCS Application and the Secure Storage module communicate via Android IPC framework as shown in the figure 5.13. The secure storage module controls the forms and data in a securely. Next, we present how the secure module handles each operation and change them into secure operations.

As it is shown in figure 5.13, it is common that most MDCS provides services such as form downloading, form filling, form data editing (locally), and uploading to the server. Form downloading service is used to retrieve forms from the server and store it locally. The "Fill Blank Form" service present the saved form, allow the data collector to fill-in data, and save as complete which is ready for upload or as incomplete to edit it when the time is convenient. The upload or submission service provides functionality to upload completed forms to the server.

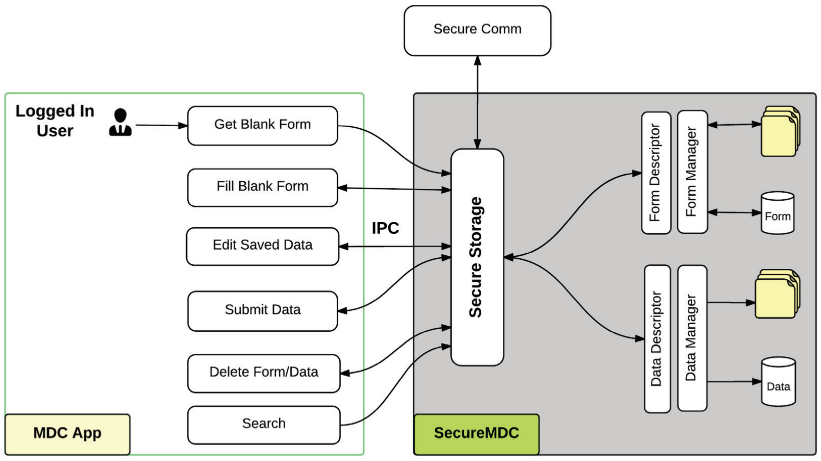


Figure 5.13: MDC States After A User Logged In

5.5.1 Secure Form Downloading

Current MDCS applications are designed to initiate an HTTP request to the server and gets a list of available forms. The user then chooses one or more forms and the forms are downloaded in the form of xml, JSON, or in binary formats. As alternative, it is also possible to load the forms manually via bluetooth or USB connections on the phone without requiring connectivity to the server. However, the manual loading processes is not suitable for a large scale project. Even in small scale project, it will be cumbersome loading forms in every phone available and moreover, it is difficult to distribute updated or new forms for projects rolled out remotely. The former solution provides flexibility to the users to choose a list of certain forms but this method requires two round trip request-response and involves the user in order to accomplish the task. This has impact on usability and power consumption. As a side note, we highly recommend automating the form downloading process through push notification services without involving the user and with minimum power consumption as possible. The recommended method can further be utilized through workflow procedure on the server. A workflow procedure for MDC has been conducted by our former colleague and further reading is available at [102]. These are more of application optimization issues and we left it to the application developers. Next, the candidate discusses how the download services is secured through the Secure Storage Module.

The standard form download procedure shown on the left side of figure 5.15 is now uses SecureMDC framework as shown on the right side of the figure. When there is a form downloading request from MDCS client, the SecureMDC framework wraps the request and send it through a secure tunnel provided by the secure communication module (discussed in detail in secure communication Chapter 6). Once form download protocol completed and forms start downloading, the secure storage module begins encrypting the forms.

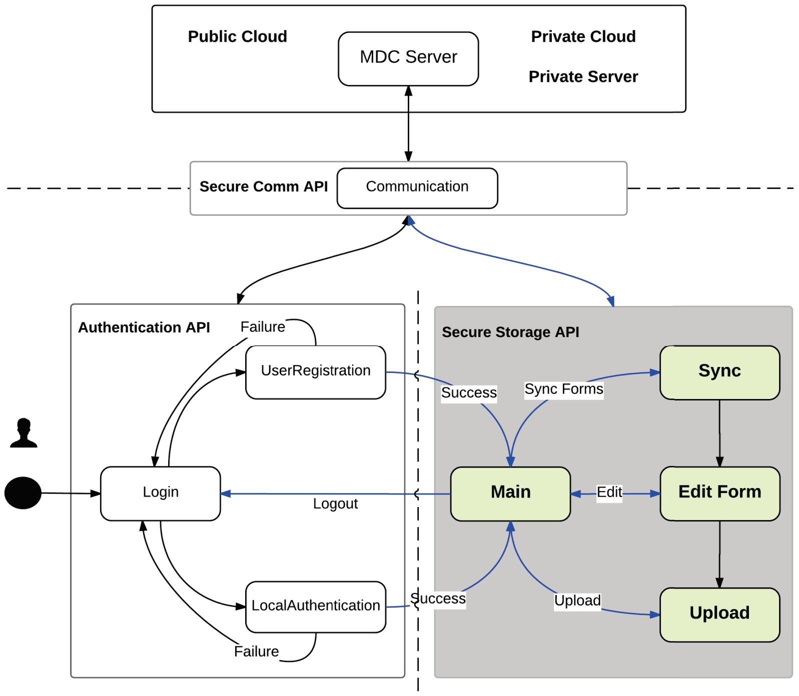


Figure 5.14: MDC Client General State Diagram

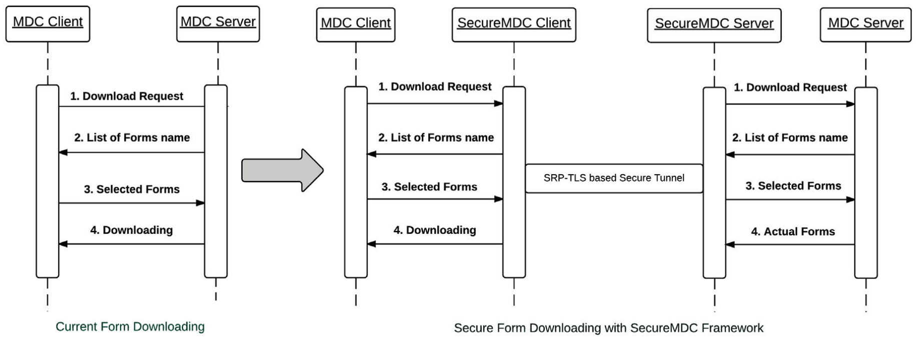


Figure 5.15: Insecure vs Secure Forms Downloading Procedure

The overall interaction between the MDCS client and the secure storage component is shown in fig 5.16. When the user authentication is completed, the MDCS application gets a token that is used to communicate with other services such as secure storage. The form downloading process is invoked through "Get Blank Form (aka Download

Form)" service. Then, the secure storage component verify the caller identity and executes the standard form downloading procedure in a secure manner. The secure storage component uses the secure communication to establish a secure tunnel and download the forms.

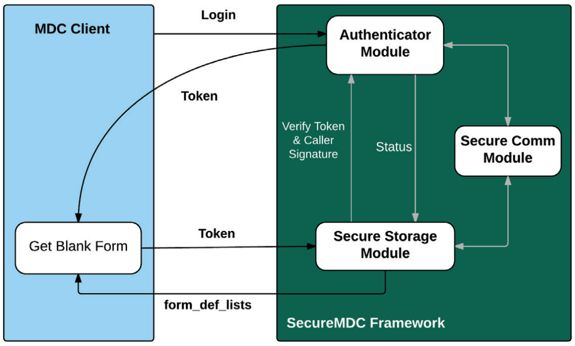


Figure 5.16: Secure Form Downloading Procedure

When the secure storage component receives the forms, it can store them in two ways. The first option is to encrypt all received forms as it is and write them to the storage as shown in the figure 5.17. The figure describes how the forms are downloaded through the SecureMDC framework and encrypted with user key when the forms arrive on the device. The pros to this approach is the secure storage component does not require any knowledge about the forms to be encrypted, but this option also has a drawback. When the user invoke "Fill Blank Form" as shown in the figure 5.14, the user has to provide with a list forms to choose. If the secure storage component does not have any information about the encrypted forms, the forms have to decrypted iteratively and make the list for the user to choose. This can be translated into the user having to wait until the form definitions are decrypted. Besides the process consumes more battery power. As a result, the user might be dissatisfied with the product.

As an alternative, we can minimize the performance impact by adding some extra information about the forms that need to be encrypted. Figure 5.18 shows how these extra information is added on the fly when the forms are passed to the SecureMDC server. If we pass over form description together with the actual forms, the secure storage component can use the form descriptor to map with an actual form. The form descriptor is a set of metadata with key value pairs that clearly describe a form definition to the user. Once it is downloaded, this part of data is stored in the form descriptor datastore of the secure storage component. The form descriptor is decrypted when a user invokes "Fill Blank Form" without decrypting the actual form. As a result, this approach is fast, flexible and consume less battery power than the former approach.

The form descriptor can be generated by MDCS client, or MDCS server or secureMDC framework. Since making the secure framework agnostic to any MDCS is the central objective of the work, we decided to generate the form descriptor on the MDCS server. We considered the MDCS client, but the security framework needs to rely on the MDCS client input which is hard to have control over using interprocess

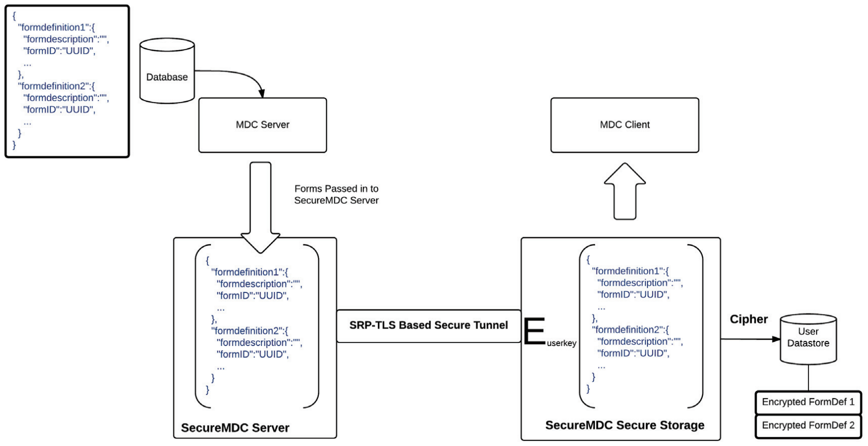


Figure 5.17: Secure Form Downloading Procedure without Optimization

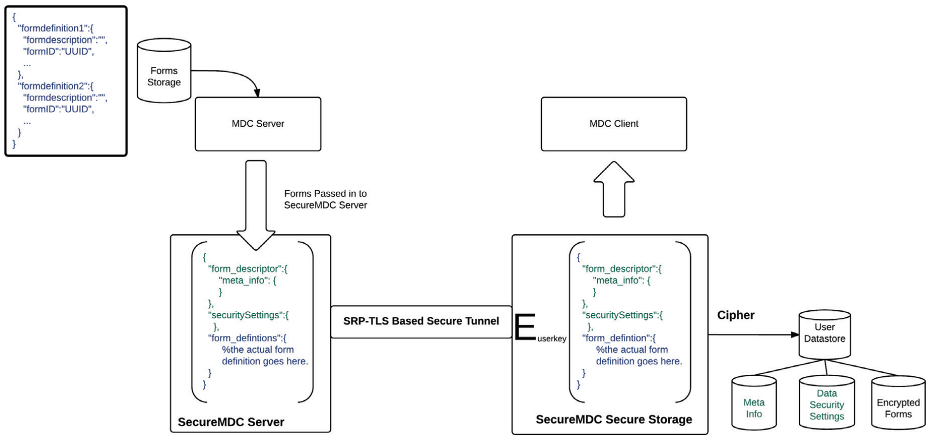


Figure 5.18: Secure Form Downloading Procedure with Optimization

communication. We can also gain some performance by processing the form descriptor on the server side. Moreover, we incorporated security related fields categorized into the "securitySettings". This set fields can be used to distribute security related critical information such as the public keys of the main actors in MDCS including the form manager and data viewer, public key algorithm types, a set of form fields that need to be encrypted (when field level encryption is enabled - N.B.: field level encryption is still research in progress). Some of these fields are shown in the Listing 5.3. The use of these public keys and the importance of secure distribution are discussed in Chapter 6. Listing 5.3 shows how the sample fields inside the form descriptor and the security settings added into a normal form definition.

```
"form_definition_1":{
  "form_descriptor":{
    "meta_info": {
      "form_id": "uuid",
      "form_version":"string",
      "desciption": "text",
      "encryption":"enabled/disabled",
      "encryptionType":["whole", "field_level"],
    }
  },
  "securitySettings":{
    "alg":"RSAES-PKCS1-V1_5"
    "fields_to_be_encrypted":["givenName", "middleName", "familyName",
     ↪  "gender", "birthdate", "address", "email"],
    "searchable_fields":["givenName", "middleName", "familyName"],
    "data_viewer_public_key":"DEREncoded",
    "form-manager_public_key":"DEREncoded",
    "user_manager_public_key":"DEREncoded",
    "project_admin_public_key":"DEREncoded",
  },
  "actual_form_defintion":{
          //the actual form definition goes here.
  }
}
```

Listing 5.3: An Example of a Form Definition with Form Descriptor and Security Settings

Once the secure storage component received the form descriptor with the actual form definition and the security settings (the form definition can be in XML, JSON or any other supported data exchange formats), it processes and stores it as follows. AES block cipher encrypts the form definition with authenticated encryption mode, GCM as shown in figure 5.19. The GCM encryption takes the form definition, an encryption key (default 256bit), additional authentication data (aka. AAD), and an initialization vector (IV).

A 96-bit IV and the AAD are generated by using secure random generator. Generating a good random binary values takes time, and we use the AAD as an input to the hash algorithm to minimize the delay. A standard hash algorithm is used to generate a unique encryption key using the user storage key which is unlocked during user authentication process, and the AAD as inputs. Once the form definition is successfully encrypted, our next step is to update the form descriptor with newly created attributes and values. Therefore, the form definition encryption related values including AAD,

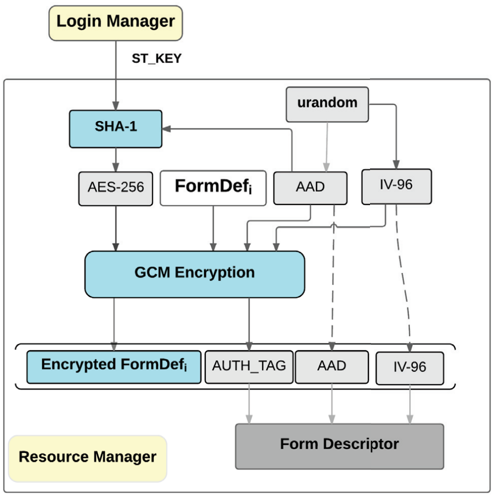


Figure 5.19: Form Definition Encryption with AES GCM Mode

IV, the authentication tag (Integrity Check Value) are stored in the form descriptor as shown in listing 5.4. There are a couple of ways of storing the encrypted form definition and form descriptor in a persistent storage. We can store them in the SQLite database by putting the form descriptor attributes and values in a given row and the encrypted form definition as an additional column in that row. Alternatively, we can store the form descriptor in a different table than the encrypted forms definition and keeping the encrypted form id in the form descriptor as shown in listing 5.4. Since there is already a logged in user who invokes form downloading service, the form descriptor stays unencrypted until the user logged out. Perhaps, we can leave the form descriptor in clear since it does not contain sensitive information that compromise the secure framework. However, since the form descriptor is a small set of metadata, it is possible to encrypt when the user logout and decrypt it when the user login.

For untrusted server such as a cloud-based server, we incorporate the public keys of the admin, the form manager, and the data viewer into the downloaded forms. Then, the master key of individual users on the device is encrypted with these public keys. The encrypted values of the user's master key with the public keys are stored in the form descriptor as shown in figure 5.4. This approach enables data sharing with the different actors in MDCS ecosystem. This part of the work is discussed more in depth in chapter 6. We have a different approach when a private MDCS server is used (private server is a traditional non-cloud based server, which the MDCS provider have physical access and complete control over the server). In the private server case, a data encryption key to encrypt data on the device is created for an individual user, either by the server or the mobile phone itself. A master key is created from the mobile password using strong key derivative function and used to encrypt the data encryption key. A copy of user master key is kept on the server in clear since this is a private server we trust and allows

```
"form_descriptor": {
  "meta_info": {
    "form_id": "uuid",
    "form_version":"string_text",
    "desciption": "text",
    "unique_fields":"[]",
    "encrypted_form_def_id": "uuid"
  },
  "dataSecuritySettings": {
    "encrypted_form_def_id": "uuid",
    "additional_authenticated_data": "AAD",
    "authentication_tag":"authTag",
    "aes_key_size":"size_in_bits",
    "iv_size":"iv size",
    "iv_value":"iv value",
    "hash_algorith":"sha-256",
    "signature":"RSA/Ellptic/DSA Signature of the sender",
    "time-stamp": "Date/Time",
    "key-exchange-Alg":"SRP, RSA-OAEP",
    "encryption-alg":"A256GCM",
    "encrypted_field":["givenName", "middleName", "familyName",
                             "gender", "birthdate", "address", "email"],
    "encrypted_searchable":["givenName", "middleName",
                                         "familyName"],
    "strict_searching":true,
    "is_single_object":false,
    "encryption_status":"enabled/disabled",
    "encryption_type":["whole", "field_level"],
    "data_viewer_key":"base_64_encoded",
    "form-manager_key":"base_64_encoded",
    "user_manager_key":"base_64_encoded",
    "project_admin_key":"base_64_encoded"
      },
    "form_defintion":{
          //encrypted form definition goes here.
    }
}
```

Listing 5.4: Form Descriptor updated fields and values after the form definition is encrypted

for recovery. This approach is discussed in details in the chapter 4 under the section 4.5.4

The choice is made to use symmetric encryption. Every data-Collectors individual user storage key, is generated on the server or a client during device activation. Pros and cons of key generation on server and client are discussed later. This key is of course not stored in clear on the mobile, but in turn encrypted, this time using the result of a password-based key derivation function. If there is a need for strong forward secrecy using a symmetric key, we present a solution that is proposed by a security research team in [76]. However, the symmetric key based strong forward secrecy comes with a trade-off that could be a limiting factor for adopting such solution.

5.5.2 Secure Blank Form Filling

Form filling is one of the services that the MDCS client provides. A user access a blank form and fill-in through a standard user interface designed to gather different types data types such as multimedia files, text, multiple choices, single-select, and other question types. At the end of the form filling process, data can be saved temporarily or marked as finalized, meaning that they are ready to be sent to the server and do not require modifications.

When the user requests to fill a form, the MDCS client display a list of available forms on the device. The displayed information is coming from the field descriptor part of the form definition persisted in the secure storage. The key value pairs in the field descriptor are shown in Listing 5.4. These fields are descriptive and more importantly, it uniquely identify the encrypted form definition. Figure 5.20 shows how the MDCS client accesses the encrypted form definition. The MDCS client passes the user token that is obtained during user authentication, and the encrypted form id which is passed to the application together with the form descriptor. The secure storage module verifies the token and the caller ID by contacting the Authenticator module (as it is discussed in chapter 4) before it begins decrypting the requested form definition. The verification may include checking token expiry time, signature, user ID, and other parameters. Then, a copy of the form definition is decrypted in-memory and made available to the application that can thereafter access and manipulate it.

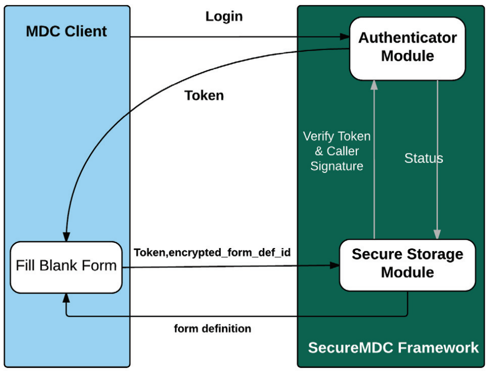


Figure 5.20: MDC Client Form Definition Request

When the client finishes filling the form, the secure storage module is triggered to take action on the collected data. The secure storage module provides two options before the data encryption process is invoked. The first is to store collected data in an incomplete state, and the user can edit the form at a later time. The second one is to store collected data as a completed and ready to be submitted. The former one requires the data to be accessible by the user when they need it. In case of later approach, the secure storage can use perfect forward secrecy which limit any further data manipulation on the data.

When the form data is stored as incomplete or finalized state, the secure storage component generates a data descriptor that is equivalent to the form descriptor. The form data and form definition are stored separately.

When the user needs to edit an incomplete form, the MDCS client passes the token and the encrypted form data ID, an equivalent of the encrypted form definition ID. When the user finishes editing the form data, the secure storage makes an update on the saved encrypted form data only when there is a change on the form data.

Locally, form data can be loaded in SQLite database instances and saved as XML, JSON or other formats that are ready to be sent. This also allows to transfer them manually using the SD-Card. Multimedia files that are related to a specific form data are treated separately and decrypted on demand which means when an encrypted form data is decrypted, the secure storage component does not decrypt the media files unless the user needs it.

The way the MDC client fills in a given form remains unchanged. The difference when using the SecureMDC is that the client does not persist or have access to persisted data directly, but it must go through the framework. On the other side, the secureMDC framework does not have prior knowledge on how the MDC client treats the form definitions.

5.6 Perfect Forward Secrecy (PFS)

To our best knowledge, the concept of perfect "Forward Secrecy" was first introduced in [54] and later it is used in [31] in the context of session key exchange protocols during a transaction [12]. A formal definition for "Forward Secrecy" is not provided in [31, 54]. However, in [31], the basic idea is that a compromise of the key protecting the current session cannot lead to the compromise of previously communicated messages or actions. In other words, with perfect forward secrecy, it guarantees that all past communicated data are protected. The same concept can be applied to data stored on the mobile device using symmetric key or public/private key approaches.

With public/private key approach, collected data are encrypted with a randomly generated symmetric key and the symmetric key that encrypt the data is encrypted with the public key of the server. The server public key is distributed to all clients in advance.

The SecourHealth framework [76] provides perfect forward secrecy (PFS) using symmetric encryption as a novel feature. The downside of the PFS using symmetric key is explained in chapter 8. The SecourHealth perfect forward secrecy approach works as shown in figure 5.21.

When the user registers on the device for the first time, a hash function is used to derive the first form encryption key, K_{sfs0}. The hash function takes a master key derived from the user password and a random seed sent from the server. After the form encryption key is derived, the application deletes the seed from the memory so that only the server keeps the seed associated with the user. The first form data is encrypted using the form encryption key. Then, the first form encryption key serves to calculate

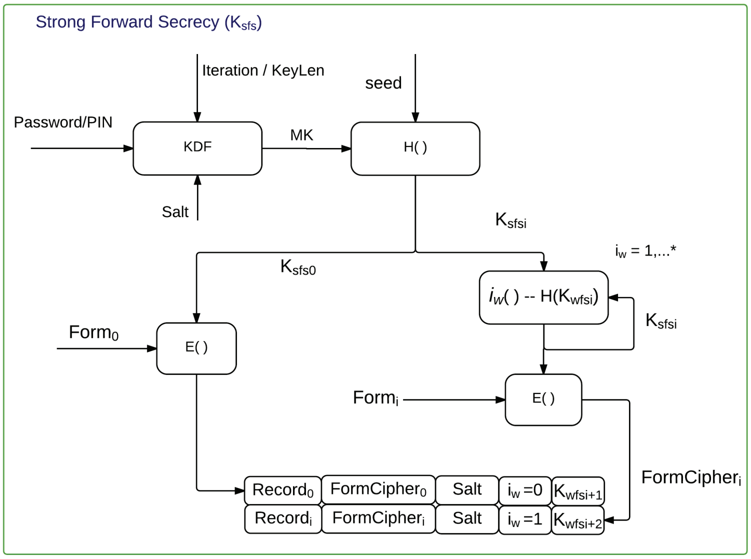

Strong Forward Secrecy (K_{sfs})
Iteration / KeyLen
seed
Password/PIN
KDF
MK
H( )
Salt
K_{sfsi}
i_w = 1,...*
K_{sfs0}
i_w( ) -- H(K_{wfsi})
K_{sfsi}
$Form_0$
E( )
$Form_i$
E( )
$FormCipher_i$
$Record_0$ | $FormCipher_0$ | Salt | i_w =0 | K_{wfsi+1}
$Record_i$ | $FormCipher_i$ | Salt | i_w =1 | K_{wfsi+2}

Figure 5.21: Perfect Forward Secrecy using Symmetric Key Encryption

the second form encryption key by passing through a hash function. Each subsequent forms are encrypted using a form encryption key derived from the previous form key. When the user log out, the next key is derived and stored in persistent storage ready to be used when the user login again in a later time. This means that the current form key found on the device has not been used to encrypt anything yet, and previous keys cannot be derived directly from this one. The only way to do that would be to generate the whole sequence of keys starting from the seed, but that is only available on the server.

We liked the idea of using symmetric encryption due to overall performance gain as every encryption/decryption is performed using symmetric algorithm both on the client and the server. We considered of using the SecourHealth PFS feature when the forward secrecy needs arise. However, currently, we are considering cloud-based MDCS, and we believe that the use of cloud will continue to grow. The SecourHealth PFS on the other hand requires storing of storage encryption key on the server database. This contradict to the security requirements for cloud storage as it is discussed in chapter 6 and chapter 9 under section 9.1.1.4. So, we found the public/private key approach acceptable for the time being.

5.7 Secure Storage Implementation and Experiences

In this section, the candidate experiences on the secure storage implementation for Java ME and Android based MDCS are discussed.

5.7.1 Secure Storage Implementation in openXdata (Java ME)

Although the solution proposed in the section 4.5.4 satisfies all the requirements we identified, it does not mean that it is the best in practice. From a usability point of view, it is very heavy for a user to remember two different passwords as we proposed in the chapter 4 under section 4.5.4, and from an implementation perspective, it might mean to re-design the whole application. Hence, in collaboration with openXdata, we designed a simplified solution that uses only one password and still satisfies most requirements except for the complete decoupling between client and server authentication. However, this was a risk that they were willing to take in order not to compromise usability. The details can be found in [46] and we will not report them here as they deal with many technicalities specific to the Java ME platform, which is not the focus of this chapter.

5.7.2 Secure Storage Implementation experience on Android Platform

The solution sketched out for openXdata is general enough to be easily ported also to the Android platform, but with different implementation challenges that we will discuss here. The first and foremost task during secure storage implementation for Android platform is to figure out what Android platform security model provides us to facilitate smooth implementation. In this section, we cover the following issues and recommend possible remedies.

1. Does Android platform package provides built-in Crypto APIs?
2. Can Android provide a truly random key generation tool on the device?
3. Is there a secure place to store keys and user credentials?
4. Are there any Android features to handle multi-user support per phone?

Android platform includes two major open source crypto library providers, OpenSSL (C and Assembly implementation) and Bouncy Castle (Java implementation). Bouncy Castle has been shipped with Android platform in a cut-down and crippled state due to device memory constraints. As a result of this, the Bouncy Castle API has been re-packaged and re-named as Spongycastle to avoid class loader conflicts with the old built-in Bouncy Castle API. The Spongy Castle must be included with an Android project as an external library while OpenSSL is supported and maintained as a native library. OpenSSL is also available in iOS which make solution porting easier in the future. Developers must, however, know how to use these libraries properly, or security might be compromised[14]. According to the Android team [33], there was a plan to remove BouncyCastle and replace it with the OpenSSL, but after Android 4.0, i.e. API level 15, Android has updated the Bouncycastle library from version 1.34 to version 1.46 and it is accessible through java.security.* APIs. In Android 4.2 and onwards, the default Bouncy Castle based SecureRandom implementation is changed to OpenSSL-based SecureRandom generation due to deterministic behavior of Bouncy Castle implementation. Thus, we strongly recommended OpenSSL as a crypto provider to build a secure solution in Android platform [33].

[14]http://android-developers.blogspot.com.au/2013/08/some-securerandom-thoughts.html

As of this writing, vulnerability on Android SecureRandom generator has been identified, and Google officially acknowledged that and announced a short term fix [15]. According Google's report, due to improper initialization of the underlying PRNG, both system-provided OpenSSL PRNG and Java Cryptography Architecture (JCA) based key generation, signing, and random number generation may not receive cryptographically strong values. The remedy is to explicitly initialize the PRNG with entropy from /dev/urandom or /dev/random and re-evaluate if a user needs to use a new key or random values when it uses JCA APIs including SecureRandom, KeyGenerator, KeyPairGenerator, KeyAgreement, and Signature. It is also recommended to generate a random AES key upon first application launch and store it somewhere secure such as internal memory.

The other issue with Android is where to store keys and credentials in a secure manner in the device. The SIM card (Subscriber Identity Module) is a type of smart card and can be considered as a secure key store. Android provides APIs to access the SIM element but, as of this writing, there is no Android crypro API exposed to perform cryptographic operations in the SIM. Basically, the SIM element is owned by the network operator and they have all privilege and control over the SIM. An application signed with operator signing key may have access to the SIM card and use it as secure element. Another way to store keys and credentials in Android is through its built-in Keystore daemon. Since Android 1.6 (Donut), the Keystore daemon was introduced for maintaining cryptographic keys for system-level apps such as VPN and Wifi connection. There was no public API for accessing the Keystore from third-party apps until Android 4.0 (a.k.a Ice Cream Sandwich (ICS)). A KeyChain public API is added in ICS for accessing the Keystore from third-party apps and store keys and certificates. Every Keys and certificate added into the Keystore are associated with their owners through the app user ID and stored in the /data/misc/keystore partition which is outside the application user space. From the security perspective, this a good place to store credentials. However, there is only one Keystore instance exist per mobile phone and all applications must share this Keystore instance. On ICS and later, the Keystore is unlocked when the device is unlocked through a pattern, a Pincode, or a password, and it is accessible to all apps. However, on pre-ICS devices, unlock screen was separated from Keystore unlocking and the Keystore unlocking is performed through an intent. Despite the fact that Keystore daemon provides a secure store, it does not maintain multi-user credential. It is worth mentioning that third-party applications can leverage system level protection for its credentials through Keystore and enhance application usability by introducing a single login interface for accessing the device. A limited built-in multi-user support per device was introduced in Android 4.2 tablets (a.k.a Jelly Bean). Each user partition contains its own accounts, apps, files and other application related data. Users are added in the shareable device manually following system provided wizard. Although application and user related data are separated, there is only one installed application instance that exist between different users. For example, if user A installed application x, user B are not allowed to install application with the same package name as application x. User B is also not allowed to downgrade application x, but it is possible upgrade it. Individual user Keystore values are separated under /data/misc/keystore/user_0 and keystore access permission is enforced by the system

[15]http://android-developers.blogspot.com.au/2013/08/some-securerandom-thoughts.html

using user ID. Therefore, as long as a multi-user environment is not fully supported, our proposed solution is still a valid alternative to handle the phone sharing problem. A secure ODK client based on this scheme is being implemented.

5.8 Evaluation

In the previous sections, we outlined the secure storage module for Android based mobile data collection systems. In this section, we discuss the evaluation of the secure solution using the following three metrics: security requirements, performance, and usability.

5.8.1 Evaluation of Security Requirements

Summarizing the security requirements, MDCS Systems require data protection while data is at rest on the device memory and while data is traversing across the network. We classified the secure solution into three architectures: secure delivery, secure storage, and secure communication. The secure delivery architecture makes sure that the client application is installed in a secure manner (discussed in detail in chapter 7. This includes code signing for application integrity and owner authenticity, establishing a secure tunnel that protect the application and its manifest attributes from eavesdropping, and verification steps which confirms the integrity of the setting attributes including the service provider or tenant public-key. We also mentioned that one can achieve the same level of secure distribution of service provider public-key and other attributes using the challenge-response protocol as described in chapter 4. Both solutions are resistant to possible eavesdropping, data tampering, and replay attack.

Secondly, The secure storage architecture is designed to ensure data protection from possible attacks and unauthorized access while data is at rest on the device storage. The storage is encrypted using a strong individual user key (randomly generated storage key), and a key derived from user password protects the storage key. We minimized the risk of data loss by separating local access from server access using a different password. The most obvious benefit from this is that if the device should become compromised, an attacker would gain nothing more than the information on the device. Given they manage to crack the local storage, the password they would gain is used only locally and thus the server is safe. The use of a storage key at individual users level instead of a single key for all user creates a strict access control mechanism between different users in a single device. We also make sure that only authenticated user has access to the application. Finally, the secure architecture guarantees data availability through data recovery mechanisms. Even if the user lost his/her password, we recover the data and allow the user to set a new password.

Lastly, the secure communication architecture is responsible for secure data transaction from the client to the server. The secure transaction includes but is not limited to, confidentiality, integrity, reply attacks, sender and receiver authenticity, and mutual authentication. Data confidentiality is guaranteed by a public-key/secret-key cryptosystem. The public-key cryptosystem is used for key exchange and to securely transfer user authentication credentials. HMAC is used for data authenticity and we use different key for the HMAC than the session key. The reply attack are prevented by using a unique message identifier or a sequence number. Once the session key is exchanged, the secret-key cryptosystem provides data confidentiality and the session-key is renewed within a certain time interval which make it harder for passive and active attackers. The secure communication part of the SecureMDC framework is presented in chapter 6. In the next subsection, we discuss the performance of the cryptosystems.

5.8.2 Performance Evaluation

The following performance evaluation is conducted on the Java ME based feature phones. One of the cheapest feature phone with low device specification (both CPU and memory) is used for the testing. The findings were acceptable as discussed below. The performance evaluation for Android is underway. Since Android device outperforms feature phones in terms of device specification, we are expecting a better performance result.

Cryptographic algorithms are the most expensive operations and running them in resource constrained devices consume computational resources and drain the battery. In order to find a right balance between throughput speed and computation, we conducted a test on the reference low-end mobile device. We used the openXdata model phone, Nokia 2330c-2 [90] for benchmarks and testing. This phone has a fair performance with a processor speed of 4.7MHz, 128KB non-volatile memory, and 4MB of RAM while still being inexpensive (less than 50$). We performed the benchmarks on the block ciphers and stream cipher algorithms. First, we measured the encryption and decryption time for different size of data and we calculated the throughput speed of both operations.

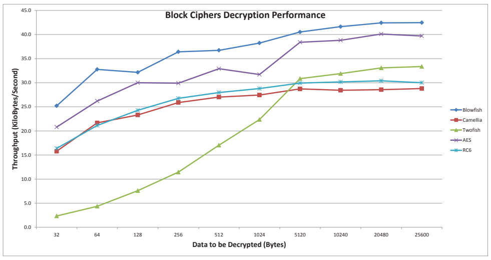


Figure 5.22: A chart displaying the block cipher speed results

As we can see from the figure 5.22, cryptographic operations are very costly on the slow processor. Blowfish and AES have better throughput speed than others with 42kb/s and 40kb/s respectively for the data size of 25KB. Figure 5.23 shows the stream ciphers throughput speed against block ciphers. For a given data size, the stream cipher algorithms outperform the block ciphers, for instance, RC4 with key size of 128 gain a throughput of 90kb/s whereas AES only 40kb/s.

If these results are acceptable or not, depends on the context in which the API would be used. How large are the data sets that are stored? What connectivity conditions will the application operate under? In the end it comes down to the individual project to determine if these results are acceptable. One thing we can however conclude is that the

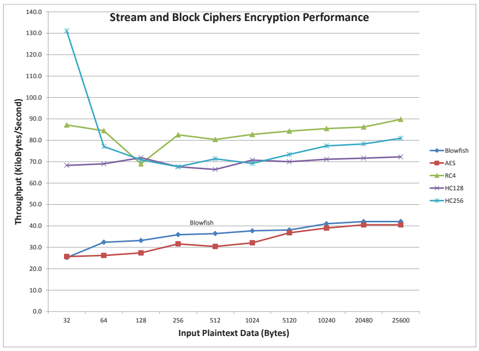


Figure 5.23: A chart displaying the block ciphers vs stream ciphers speed results

throughput speeds on the used device (Nokia 2330c) are reasonably high. Under optimal conditions a GPRS/EDGE [38] as of 2007 speeds up to 22 kb/s can be reached for upload. We are conducting an intensive testing on SecureMDC framework performance on the Android devices and the result will be presented in the future. However, from the preliminary testing result shows that the SecureMDC framework performs well.

5.8.3 Usability Evaluation

Here, we would like to emphasize that the usability evaluation is conducted from the developer point of view when Java ME based MDCS is used. However, from users perspective, we made the following statement.

We evaluate the secure solution from two usability aspects: developers and users. Users are data collectors. The introduction of the secure solution in the existing MDCS system requires nothing from the users except remembering their password and keep it safe. All security logics and services are hidden from the users in the transparent way. If we look at it from the developers point of view, in order to incorporate the secure solution the programmer needs no knowledge of the underlying security logics and cryptographic. For instance, the secure storage integration with the existing client requires only basic understanding of standard Java ME record store operations, and this is also true for secure communication integration. We finished integrating our secure storage solution into the existing openXdata and ODK clients smoothly without requiring significant changes on the client code base.

5.9 Conclusions

Our primary aim in designing these solutions, besides fulfilling the security requirements we identified, has been that they should be easy to integrate with existing systems, without requiring substantial retrofitting. The secure storage scheme is therefore quite modular.

We first implemented our design for Java ME and then adapted it to work also on Android based MDCS. Switching to smartphones brought us both flexibility and challenges at the same time. The attack surfaces, the security model, and built-in security features vary from considerably from feature phones to smartphones. Because of the lack of default security in Android based smartphones, different secure storage solutions have been developed such as SQLCipher, Facebook Conseal API, and other third party solutions. However, none of them could completely solve all our problems, mostly because they assume a single user per device and usually do not provide key management solutions. Another advantage of smartphones over feature phones is the availability of mobile application management solutions (MAM), which allows one to wrap insecure application and make them secure. The wrapping process incorporates many services including secure storage and remote data management. Even if such MAM solutions are close to our approach, they outperform us in terms of simplicity of integration. The drawback is that most MAMs are provided by big corporation such as IBM and Citrix, are not open-source and usually are developed in collaboration with device manufacturers and are therefore granted platform privileges that normal application or API cannot leverage. Should such a solution be developed as an open and free to use or even incorporated into the standard Android platform, it could be a valid alternative to our framework. Until then, we believe the solution we have developed has fulfilled the goals we had set.

We have developed our own prototype MDCS using the secure storage module and other Secure framework components, and tested it on various Java ME based feature phones with different settings in order to collect experimental data. The results are encouraging, since the performance with the default security settings was acceptable also on very low-end phones, and the openXdata integration proceeded smoothly. The secure solution has been integrated into a production openXdata code base. We are conducting intense test on ODK with the secure storage solution.

6 SecureMDC: Secure Transmission and Cloud Storage

This Chapter is based on the candidate's previously published work at the Nordic Symposium on Cloud Computing and Internet Technologies (NordiCloud '13). The full paper is available at [44].

Secure cloud storage is a hot topic nowadays. However, most solutions are user-centric, i.e., they focus on private data owned by a single individual. In the case of Remote Mobile Data Collection, we have many collectors continuously uploading data to a central cloud storage on one hand, and several data analysts or decision makers that require this data in real time on the other. In this chapter we investigate the challenges related to this model and discuss possible solutions.

6.1 Introduction

As already described at the beginning of this thesis, the general architecture of these systems consists mainly of two components: a server where data forms can be created, distributed and later collected; and a mobile client where forms can be downloaded, filled in and uploaded back to the server when connectivity is available. The server application, thanks to cloud based platform services like Google App Engine [51] and Amazon EC2 [2], can run an open source server on a third party infrastructure with minimum effort and cost. However, despite the advantages of these new technologies, there is a natural concern for the security of the data while at rest on these third party servers and the mobile devices themselves. The data collected is often of sensitive nature and its confidentiality needs to be guaranteed during all the stages of the collection process. As we discussed in the previous chapters, security has never been a priority in the development of data collection applications, and although some solutions have been proposed for the protection of data stored on the client side [73], the use of cloud storage poses new threats to data security that have not been adequately addressed yet.

The aim of this chapter is to present an analysis of the challenges in securing data in the cloud from the point of view of a MDC project, identify critical problem areas and discuss the pros and cons of different solutions that might solve or mitigate some of the issues.

The remainder of this chapter is organised as follows. In Section 6.2, we discuss the processes related to a data collection project and its structure. Section 6.3 describes the security and functional requirements of MDC. Section 6.4 provides an overview of the threat model we have adopted. In Section 6.5, we discuss some existing secure cloud storage solutions. Section 6.6 explains thoroughly the proposed solution in the context of RMDC. Finally, we conclude in Section 6.8 with some discussion and future work.

We assume the reader is familiar with basic cryptographic and security concepts as symmetric and asymmetric encryption, key derivation, authentication protocols, access control and confidentiality and algorithms like RSA and AES.

6.2 Typical MDC System Organisation

To understand the challenges in securing cloud data in a MDC project, it is essential to analyse the data flow in various stages of the collection process and define precise user roles in the system. Some of this has already been covered in the previous Chapters, but we further formalize it here.

Projects typically consist of form creators and data collectors. The former create the *form definitions* to be used to collect data in the field and have access to the collected data. The latter download the form definitions in the field and actually collect and upload the data. However, other roles might exist on the actual system like site administrators and data viewers, which also have the right to edit, delete or read all data. This role overlapping creates problems when trying to define a security architecture where it must be clear who has the right to do what. Thus, we begin by defining a semi-hierarchical structure, where we decouple data access from system administration.

- **The site administrator :** is in charge of server maintenance, user management and data recovery. This administrator does not need to be a physical person, but just a role in the system, that does not need to access form definitions and collected data.

- **The form managers :** are in charge of creating form definitions and have access to the data collected using such forms. In practice, form managers are the data owners.

- **The data collectors :** are trained specifically for a specific project and sent into the field to actually collect the data at different locations. They usually do not have access to the collected data after upload to the server.

- **The local administrators :** provide technical support on site. Data collectors might forget their password or lose their phone. Since we are talking about *remote* data collection, in places where network connectivity is scarce and unreliable, each location might then have a local supervisor with some administrator rights on the system, that can provide this type of support while coordinating the data collection process locally. They do not have direct access to any form definition or collected data.

- **The data viewers :** are usually analysts that are only interested in reading the collected data for analysis purposes.

The collection process itself starts by setting up the server with the list of predefined collectors, form managers, local administrators and data viewers, which all receive a username and password to get access to the server. The project organisation remains pretty much unchanged over time.

The form managers design the form definitions to be used with each project, and the collectors can download these definitions on their mobile client to start collecting data. When data collectors register to the server for the first time, they are presented with the list of possible form definitions they can download. A predetermined work-flow might decide in which order the forms should be downloaded. The form definitions are then downloaded and the data collection starts. Once the forms are filled, they are regularly uploaded to the server and deleted from the phone. At this point the form managers and data viewers can synchronise with this central repository to download the most recent data to their personal machines for further analysis.

Some access control should be in place to be able to define which form definitions and collected data is accessible by different form manager and data viewers. Finally, we assume that the collected data are static after upload, i.e. editing is restricted on uploaded data.

6.3 Security and functional requirements

Requirements for secure MDC systems have already been discussed elsewhere [73, 74] and in the previous chapters. Here we want to focus on the cloud storage, and the main functional requirement we take into consideration is data availability. In this context with availability we mean the ability to share collected data with predefined data viewers and form managers and the guarantee that added security will not result in potential data loss, i.e., that a recovery alternative always is available even if the data owner loses access to the system. Also, although being able to update already collected data is not a functional requirement just yet, a collector should be able to edit data as long as it resides on the phone.

In terms of plain security the requirements are mostly standard ones:

- Strong data protection both on the mobile device and on the central storage
- Access control to allow only authorised user to upload and download data
- Mutual authentication between server and clients
- Secure communication channel

The challenge is to satisfy these requirements while allowing for sharing and recovery, with the constraint that a potential solution should use ready available technology and be easily compatible with existing MDC systems, and should not dramatically increase power and memory consumption.

6.4 The Threat Model

In our discussion, we consider the adversary model of the *"honest but curious server"*. That is, the cloud platform where our server is running might monitor our input and outputs, both from network and the local memory, but it will not compromise our application code or change the data stored in our database.

In other words, assuming that what is stored on the cloud might be exposed to an attacker, we want to minimise the potential damage that such a leakage could cause.

In particular, what happens if the database with the collected data is stolen, or accounts compromised?

6.5 Available Solutions

6.5.1 Secure Cloud Storage

In a report from the Fraunhofer Institute for Secure Technology [14] a series of cloud storage services are analysed from the point of view of their security. Of all the security features they look at, we are most interested in confidentiality and recovery i.e., data protection on the cloud and data recovery process when a user loses credentials. It is clear that the safest solution when it comes to confidentiality is to encrypt the data locally before upload with a key which is not shared with the storage provider, as offered in TeamDrive, Wuala, Mozy and CrashPlan (to which we can add Mega [80] and nCrypted [88]). The main drawback of this approach is that if the user loses the encryption keys stored locally, also the data is lost. A compromise is to derive the encryption keys from the user password, so that, even if the password is never sent to the provider, users can re-generate the keys as long as they remember the password. This however can compromise confidentiality itself if the password is not strong enough.

The problem is that all these storage services are based on a user-centric model, and if users do not trust the provider enough to share a recovery key with it, then it is ultimately their responsibility to think of a recovery strategy and back-up their keys. Also the idea adopted by nCrypted of having an additional user public/private key pair and storing the encryption keys encrypted with the public key on the cloud, plus a back-up copy of the private key encrypted with the user password , does not help much. If the computer where the private key is stored crashes and the user forgets the password, the data is lost. Besides, if the password is weak, the private key copy on the cloud might be compromised.

Another problem in using existing cloud storage services in a remote data collection project, is that each collector and form manager would have to have two accounts: one to the server where form definitions can be created and downloaded, that might run on Google App Engine or Amazon EC2, and the second account on the secure cloud storage provider where data would be collected and shared.

Finally, group keys are typically used when multiple users have to write in a shared folder. However, when using hundreds of mobile phones that can easily be stolen or lost, combined with likelihood of weak passwords that are easy to remember and type in handheld devices, shared group keys might easily be compromised, and with them all collected data.

Among the current MDC systems, only OpenDataKit (ODK) [16] and its derivations [93] provide some secure cloud storage, but it is very limited. The mobile client encrypts each form to be submitted to the server with a unique symmetric key generated locally. This key is in turn encrypted with the form manager public key that was attached to the corresponding form definition used to collect the data.

The main advantage of this solution is that strong encryption is used, and data is protected both on the cloud and on the phone. The form manager will have to download the encrypted data on their personal machine, where dedicated software called Briefcase will decrypt the data with the corresponding private key. The big disadvantages are that: only one person can decrypt the data, once a form is saved for upload on the phone it cannot be modified, and losing the private key on the form manager computer would mean to lose all data on the cloud. This approach also generates a lot of extra data to be uploaded and uses up phone resources to perform heavy encryption. Related problems are the lack of server authentication when the form definition is downloaded, and no protection of the keys stored on the phone is guaranteed. All data is, in fact, stored on the SD card where anyone with physical access to the phone can read it and modify it.

6.5.2 Secure Communication

In some previous work [44, 73] we argued why HTTPS cannot always be considered as the best solution for secure communication in low-budget data collection projects. The reasons were mainly three: bad support/implementation in older phones, criticisms towards the Certificate Authority Trust Model (see for instance [112]) and the cost of the certificates. A typical feature phone would, in fact, have a limited list of root certificates which could not be modified by the user, and that was not standardised across models and manufacturer. This would make it difficult for a project to choose a CA from which to buy an SSL certificate that could be supported by all the handsets deployed in the project. On the other hand, not being able to modify the list of root certificate would give a guarantee that self-signed certificates could not be accepted, providing higher security. The result however was that most MDCSs would simply not use HTTPS at all. With smart phones, the support for HTTPS improved dramatically, but, especially in Android, self-signed certificates and CA signed certificates are treated in the same way, with the consequences discussed in the previous section on app distribution.

In the next section we will propose a way to leverage the characteristic organisation model of MDC projects in order to eliminate some of the weaknesses in the existing solutions for secure cloud storage services in our settings.

As long as HTTPS is used to protect data transmission, only a valid certificate recognized by the mobile device is needed. Rest of the key management is taken care of by the SSL/TLS protocol on which HTTPS is based.

The Hypertext Transfer Protocol Secure (HTTPS) is a protocol designed to send HTTP requests and responses in a secure manner through the SSL/TLS [58] protocol and it is a standard way of securing client-server transactions. In fact, all MDCS we reviewed can actually use it. However, we have also mentioned that HTTPS might not be the best solution given all the constraints discussed in the previous section. Here we explain why. The security of HTTPS depends mostly on two things: valid certificates signed by a trusted Certificate Authority (CA), and a correct implementation of the protocol. Both of these conditions are not equally easy to satisfy on all mobile phones.

The first problem is the certificates. While MDC systems that offer both software and service, have a centralized server and can secure the traffic to and from all the clients by using a single certificate, each project that wants to set up its own server will have to buy its own certificate. This is not cheap, especially for certificates signed by the most well-know and well-supported CAs (Certificate Authorities). Besides, the list of supported CAs pre-installed on each phone is controlled by the Original Equipment Manufacturer (OEMs) and it is not standardized. Thus, in order to guarantee compatibility, certificates signed by different CAs might be necessary, thus adding to the cost. A cheaper alternative might be to use self-signed certificates, but they do not provide the same security level as CA signed certificates, and not all mobile platform accept them except Android. See also [137] for a discussion on this subject.

Secondly, even assuming that the project is willing to take on the burden of using CA signed certificates, there is no guarantee that the implementation of the secure protocol using them will be equally secure on all devices they deploy [122].

Another source of concern about the use of HTTPS for remote data collection, is the long handshake needed to establish a secure connection. Although we do not have conclusive evidence from field data, a quick test showed that such a handshake can take up to 12 seconds to be performed on a 3G network, and it does not seem like this would be optimal for a working environment with scarce and intermittent connectivity. We should also think that the HTTPS protocol was designed to be used on the World Wide Web where clients need secure communication for transactions with various unknown servers. In our case we know both who the server and the users are in advance, hence we could establish in advance how client and server should authenticate each other and how information should be exchanged. This would eliminate the need for the handshake and a secure communication could be achieved with a lighter and faster protocol.

Such lighter protocols have been proposed previously [59, 118]. They are based solely on symmetric cryptography and are built on different assumptions than those discussed here, but some of the underlying ideas are still valid for our purposes.

6.6 Proposed Solution

In a typical use case with an organization model as mentioned in Section 6.2, data collectors are the source of data and they are required to share the collected data with the form managers who created the form definitions, and possibly the data viewers. The main idea for sharing is the same as many cloud storage provider and ODK: encrypt the master key used to encrypt the submitted data with the public keys of those who can read the data. However, unlike user-centric models, in a MDC project we know in advance who are the data consumers are. This allows to distribute the public keys before data collection even starts, and therefore to enable sharing for multiple data readers in an asynchronous way. In addition we will use the same approach to publish securely other information that will allow for data recovery, and discuss some variations on this model to increase efficiency.

The details of the overall process are explained in the next sections, while an example is shown in Figure 6.1.

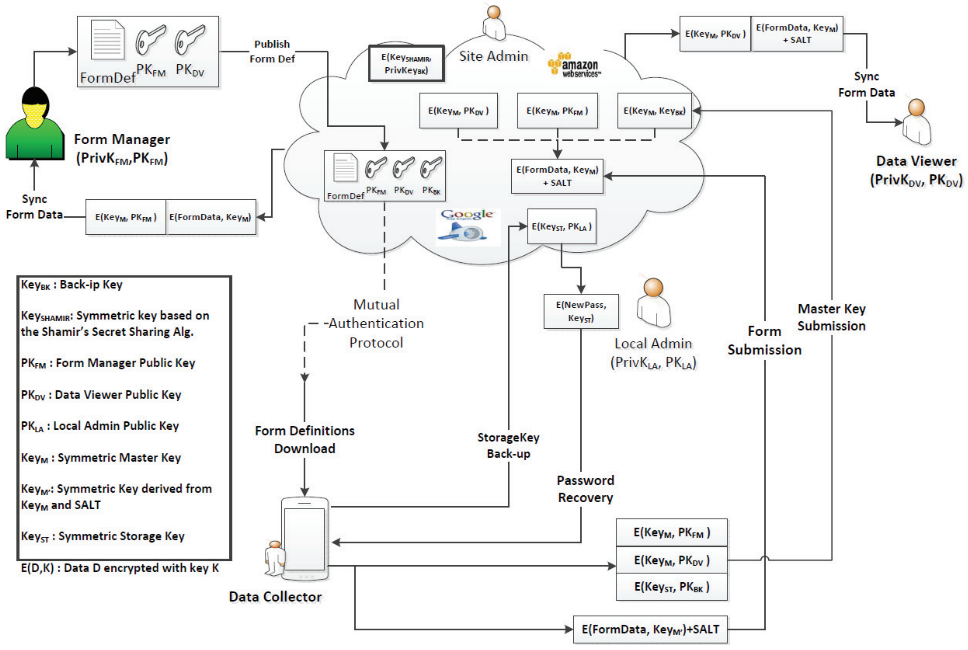

FormDef
PK_FM
PK_DV
Publish Form Def
Form Manager (PrivK_FM,PK_FM)
Sync Form Data
E(Key_M, PK_FM)
E(FormData, Key_M)
E(Key_SHAMIR, PrivKey_BK)
Site Admin
amazon webservices
E(Key_M, PK_DV)
E(FormData, Key_M) + SALT
Sync Form Data
Data Viewer (PrivK_DV, PK_DV)
E(Key_M, PK_DV)
E(Key_M, PK_FM)
E(Key_M, Key_BK)
FormDef PK_FM PK_DV PK_BK
E(FormData, Key_M) + SALT
Google
E(Key_ST, PK_LA)
E(NewPass, Key_ST)
Local Admin (PrivK_LA, PK_LA)
Mutual -Authentication Protocol
Form Definitions Download
StorageKey Back-up
Password Recovery
Form Submission
Master Key Submission
Data Collector
E(Key_M, PK_FM)
E(Key_M, PK_DV)
E(Key_ST, PK_BK)
E(FormData, Key_M')+SALT
Key_BK : Back-ip Key
Key_SHAMIR: Symmetric key based on the Shamir's Secret Sharing Alg.
PK_FM : Form Manager Public Key
PK_DV : Data Viewer Public Key
PK_LA : Local Admin Public Key
Key_M : Symmetric Master Key
Key_M': Symmetric Key derived from Key_M and SALT
Key_ST : Symmetric Storage Key
E(D,K) : Data D encrypted with key K

Figure 6.1: Proposed Solution for Secure Cloud Storage for MDC Systems.

But, before we delve into the details of the secure communication and cloud storage sections, we present first the custom secure communication protocol developed for Java ME application and described on our earlier publication [73]. We believe this gives an insight to the read on the history and road map since we began this research and how we ended up with a new and standard way of secure communication which is discussed in section 6.6.2.

6.6.1 Custom Secure Communication Protocol for Java ME Apps

Given that the application has been installed and configured, and we have some key to start encrypting our traffic as it is presented in section 7.1.1, the next step to consider is how the communication should be conducted from this point onwards. What we want to achieve, given that we do not use HTTPS, is:

1. Confidentiality

2. Mutual authentication
3. Minimal traffic time and amount
4. Protection against replay and man in the middle (MITM) attacks
5. Data integrity
6. Less computation key management
7. Compatibility with existing applications

We mentioned that in previous works only symmetric keys were used to secure communication, but this can be problematic when we have multiple users on different phones, where each uses a different key. The server must then store all possible keys, and retrieve the correct key from the Keystore every time a request arrives, based on the pair user-device. Besides, on the client side, these user specific keys are stored encrypted on the phone until the user logs in. What if the user has not registered yet or has lost the password and needs to perform a recovery procedure? Which key should then be used? There should be a device key that any user can use, but how do we encrypt it on the phone to keep it secure and at the same time available? All these problems would be solved if we could use a public key to initiate all communications with the server, and this is indeed our recommendation. We said that certificates are not good for our purpose, but if we managed to obtain the server public key securely as described in the previous section, then we can use it without the need for certificates.

This server public key based approach would mean that the server can decrypt all requests coming to it just using a single private key, without knowing whom they are from or what they are about beforehand. This has multiple advantages, besides a simplified key management from the server side and the fact that practically no sensitive information needs to be communicated in clear (like the user name and application-id to identify the key to use). Since it is always the client that initiates the communication with the server, we can send directly the user credentials to achieve mutual authentication (the server authentication based on the fact that, if the private key is not compromised, only the genuine server can decrypt and answer a request). So we do not need to run a dedicated authentication protocol, and also the risk of MITM attacks are minimized, since we do not receive the public key each time from the server, but we use the one stored on the phone.

Hence, requirements (1), (2), (5), (6) and (7) would be satisfied and the format of a client request and server response could then look like the one shown in Figure 6.2.

The TYPE field is used to identify the type of secure service request to the proxy server. The request could be user registration, or a password recovery, a data transfer (upload or download) or a session establishment. By knowing this, in most cases the server could complete the request operation directly, without requiring further exchange of messages, thus satisfying (3). Also, having the type of the request been encrypted as well, an attacker cannot distinguish different requests just by eavesdropping the traffic.

The MASTER-KEY and the NONCE are used to derive the encryption and authentication keys used in the server response, in the same way as the session and authentication keys were generated from the PSK in the previous section. Although a session

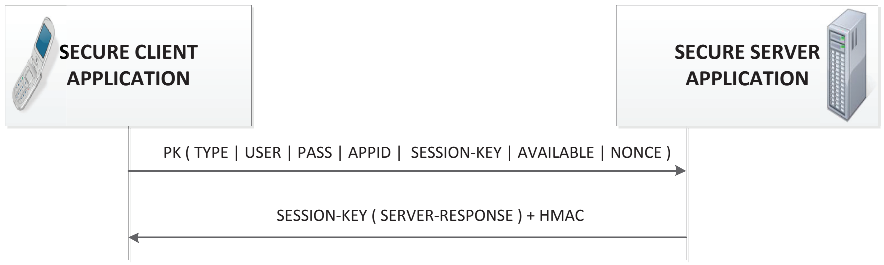

SECURE CLIENT APPLICATION
SECURE SERVER APPLICATION
PK (TYPE | USER | PASS | APPID | SESSION-KEY | AVAILABLE | NONCE)
SESSION-KEY (SERVER-RESPONSE) + HMAC

Figure 6.2: Example of a generic secure service request/response

key could be sent directly, deriving it from a master secret gives us more flexibility, as we explain in the next section 6.6.1.1.

The AVAILABLE FIELD is used for parameters that are specific for each request TYPE. For example, a User Registration request might contain a back-up copy of the key used to encrypt the users' data on the phone.

We should point out that the public key of the server is not secret and the parameters in the HTTP request are not negotiated, but retrieved from runtime memory after login and directly sent by the client. Hence the user's credentials must always be included to guarantee the authenticity of the client request or anyone could try to forge a request and obtain services or information from the server. The fields USERNAME, PASSWORD and APPLICATION-ID are there for exactly this purpose. The APPLICATION-ID is an identifier unique for each instance of the client application, i.e., for each mobile phone. This is needed to identify the specific user account, since the same user might have activated an account on different phones. It should be among the security settings sent to the client at initialization time, so that the server can guarantee the uniqueness of each Id. In fact, if we were to use the IMEI of the phone, like some existing MDCS, we would limit the range of phones that can be used, as this identifier cannot be extracted by a J2ME application on many lower-end phones.

When it comes to the possibility of replay attacks (requirement (4)), they should be neutralized by sending only requests that do not trigger changes on the server if idempotent, or expose more information than that sent in the first response. In fact, a client might have to re-send the same request again if the server response never arrived, and this request should be identical to the previous one, because it require resources to create and re-encrypt a new request just to change a sequence number or a nonce. Hence, a replay attack would be indistinguishable from an actual client request, but by sending exactly the same (possibly buffered) response every time, without doing anything else on the server, one would limit the damages this attack can do.

Notice that we use a public key only for the server, not for the clients. We do this because it would be problematic to distribute user specific private keys, and very difficult to keep them secure on the mobile phone, given the means at our disposal and the problems discussed in section 2.3.1. This excludes the possibility of signing the user's messages for integrity and accountability purposes, and for the server to initiate a secure communication with a specific user or device. Besides, although public key operations are still feasible on lower-end phones, given reasonable key sizes, private key operation would probably be beyond these phones capabilities.

Finally, the public key approach is used only to initiate the communication with the server, and perform simple tasks like user registration, password recovery and session establishment, that require only a single request and response, and minimal data. In fact, the amount of information that can be encrypted with a public key, at least in RSA iwth PKCS#1 padding, is the length of the key minus 11 bytes, and we would like to keep the key as short as possible to minimize the computational power required to perform encryption. For instance, a 128 byte RSA key would be enough to encrypt most types of requests, assuming that each field could contain 16 bytes.

Dealing with the transfer of large amounts of data, like upload or download of forms, is discussed in the next subsection.

6.6.1.1 HTTP Sessions and Data Transfer

A public key encrypted request can be used to establish a session, and obtain a HTTP session-Id from the server. The server maps HTTP session object to a user session key and send HTTP session ID to a client. A client uses a session-Id for further interaction to the server. The session-id will then be included in a cookie with each new request as it is normally done with HTTP sessions. However, unlike HTTPS, we are encrypting the traffic at the application layer, and the cookie will have to travel in clear. Still, hijacking the session would require the attacker to know both the encryption-key and the authentication-key currently used by both client and server, and replay attacks can be prevented by adding a unique sequential number to each request.

In order to minimize the need for establishing new sessions, we could renegotiate the session key periodically, without changing the session-Id. This should be done as a separate operation, since if either the server does not receive the request or the client the response, the two parties might end up using different session-keys. The operation would consist in sending a new session key encrypted with the current session key, and waiting for a confirmation from the server that the request was successful. The client will not send new data until it gets this confirmation, and if necessary, will re-send the same request. The server should not delete the old key until the first time the client sends data encrypted with the new one. The new keys will be derived from the original master key sent with the public encrypted request that established the session, and the new nonce. This solution, shown in step 6 of Figure 6.4, will give more protection even if the current session key is compromised and the attacker knows the new nonce.

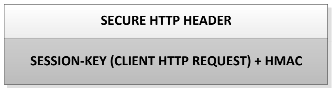


Figure 6.3: Secure HTTP request

Since a client using our encryption strategy might already have its own (unencrypted) communication protocol and use some particular HTTP request to transfer information, we encapsulate the whole HTTP request and send it through a secure tunnel, and possibly add a standard secure HTTP header in a fashion similar to that of

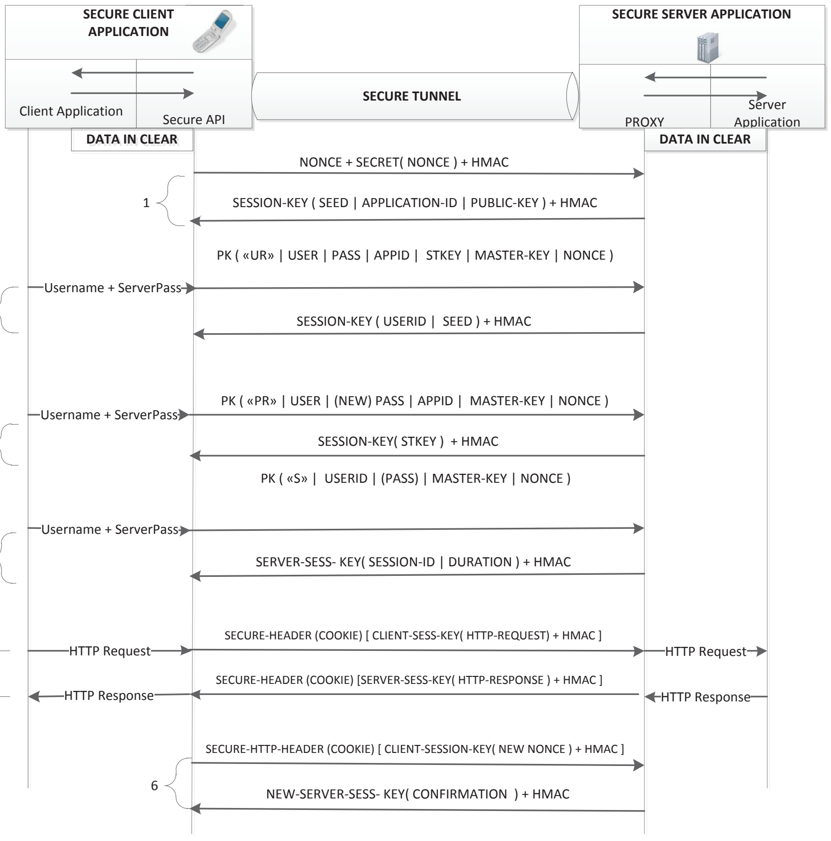

SECURE CLIENT APPLICATION
SECURE SERVER APPLICATION
Client Application
Secure API
SECURE TUNNEL
PROXY
Server Application
DATA IN CLEAR
DATA IN CLEAR
NONCE + SECRET(NONCE) + HMAC
1
SESSION-KEY (SEED | APPLICATION-ID | PUBLIC-KEY) + HMAC
PK («UR» | USER | PASS | APPID | STKEY | MASTER-KEY | NONCE)
Username + ServerPass
2
SESSION-KEY (USERID | SEED) + HMAC
PK («PR» | USER | (NEW) PASS | APPID | MASTER-KEY | NONCE)
Username + ServerPass
3
SESSION-KEY(STKEY) + HMAC
PK («S» | USERID | (PASS) | MASTER-KEY | NONCE)
Username + ServerPass
4
SERVER-SESS- KEY(SESSION-ID | DURATION) + HMAC
SECURE-HEADER (COOKIE) [CLIENT-SESS-KEY(HTTP-REQUEST) + HMAC]
HTTP Request
HTTP Request
5
SECURE-HEADER (COOKIE) [SERVER-SESS-KEY(HTTP-RESPONSE) + HMAC]
HTTP Response
HTTP Response
SECURE-HTTP-HEADER (COOKIE) [CLIENT-SESSION-KEY(NEW NONCE) + HMAC]
6
NEW-SERVER-SESS- KEY(CONFIRMATION) + HMAC

Figure 6.4: Example of protocol using the methods suggested in this section. The notation $key(data)$ indicates that $data$ is encrypted with key, in particular PK stands for (server) Public Key, while $Header[body]$ indicates an HTTP request.

S-HTTP [110]. Both the secure header and the encrypted data integrity (requirement (5)) is protected by Hash-based Message Authentication Code (HMAC) which is used to verify both the data integrity and the authenticity of a message, as shown in Figure 6.3. Since the HMAC protects also the secure HTTP header, the sequence number used to avoid replay attacks could be part of this header, rather than the encrypted body, that could be therefore buffer for possible retransmission. Notice that using a time stamp like in [110] is unfeasible unless mobile phone and server clocks are synchronized. In figure 6.4 we show an example of how the protocol we propose could work in practice.

This approach satisfies also requirement (7), since it allows a separation between the application request format and the cryptographic solution adopted, although the price to pay for this flexibility is an extra HTTP header. More efficient protocols could surely be designed if taylored for a specific application, but that would be more prone to error and not interesting for securing other existing MDCS.

6.6.2 Secure Channel

The primary choice in the definition of the architecture was the one related to the core responsibility of the system: how do we secure the communication channel? Depending on the choices made here, the other components of the system are defined. While we already decided to use SRP, it is still unclear how exactly this protocol will be used to provide a secure channel.

6.6.2.1 SRP vs TLS-SRP

SRP itself does not provide encryption, but a mutually authenticated and strong symmetric key that can be used directly or indirectly for that. The encryption, once we have a key, can be done for example with AES. This whole process of key exchange and encryption is not as trivial as it seems, because many other issues must be addressed, such as a mechanism of key rotation, nonces, integrity checks, message authentication and so on. Since these issues, not easy to deal with, are far from being new, existing solutions should be preferred to "re-inventing the wheel", and accordingly a SSL/TLS protocol is a very good candidate. The standard SRP key agreement process can even be optimised when coupled with the encryption, since evidence messages (M1 and M2) can be replaced by the encrypted payload itself. For these and other reasons, TLS-SRP should be preferred to any other custom solution.

It is important here to understand that the secure channel can be protected acting at different levels of the communication stack. The alternatives from this point of view are mainly two:

1. Application-level
2. Transport-level

In the first case the application-level logic is directly involved in encrypting/decrypting the channel, performing authentication and running the security protocols above of other other layers of unprotected communications. In the second case security is managed outside the application logic, offered by the layers underneath, in a completely transparent way. For example, what often happens on the server-side is that we use container-managed security. When the application requests an HTTPS resource, the container will manage the procedure of establishing, maintaining and destroying a secure session. The two levels have each one advantages and disadvantages. While, for instance, using application-level techniques implies an increase in the needed implementation work, it can reduce the configuration and deployment burden which is typical of the other kind of solutions. In this case for example we want a product which is easy to deploy for organizational staff without high technical skills, so the first solution seems better. On the other hand keeping the authentication at a lower level makes integration considerably easier, reason why this is the approach used by almost all products nowadays. Usually the encrypted channel is built on SSL/TLS libraries, relying on certificates for server authentication, and where necessary keeping only client authentication at applicaion-level. With SRP we achieve mutual trust, but at a cost: the encryption of the channel is based on user credentials, which have to be somehow submitted through a user interface. This, as long as there is no native support, means that we still have to make changes in the app logic. Also on server side there must be the possibility to register users and store their verifiers, often integrated with other forms of access control related to data and functions, depending on the identity of the currently logged user. In the end, even if we choose a transport-level solution such as TLS-SRP, we still have to modify a considerable part of the higher level tiers. In addition to this we should consider also the support for SRP and TLS-SRP protocols. If the first one counts a certain number of implementations, the second is still not widely supported.

The main problem in this case is that Google App Engine (GAE) does not support SRP cipher suites for TLS. This means that, despite on the client we could have implemented them ourselves, the server solution would still not be compatible with this platform. In other settings we could simply choose a different cloud solution, offering IaaS (Infrastructure-as-a-Service) instead of PaaS (Platform-as-a-Service), but for this case the requirements strictly impose the compatibility with GAE. For the general case, in which this requirement could be not so important, we studied a possible system setup to enable the secure channel over TLS-SRP. This solution, not elsewhere detailed, is shortly described in section 5.2.1.2, since it can represent a valuable alternative to the main one discussed in this thesis. The chosen solution, instead, is to work at application-level. The focus of the prototype is then to show SRP mutual authentication and key agreement. The specific technique for encryption, once the key is available, is not going to be detailed in the prototype. We can assume that every message sent after the authentication is encrypted using AES and sent together with a digest to check its integrity.

Also in this case we propose an alternative solution, in section 6.6.2.2.

6.6.2.2 Alternative solution using TLS-SRP

According to the current support status for TLS-SRP cipersuites, OpenSSL and GnuTLS are the two most mature providers. The proposed architecture involves the use of Apache HTTP Server as a front-end for the Application Server, using mod proxy. This allows to use either the module mod ssl or mod gnutls, respectively belonging to each one of the two providers. We briefly see an example showing how to configure and run an TLS-SRP session using OpenSSL 1.0.1e. As a first step the SRP verifier file must be created and a user registered, using the OpenSSL command-line tool:

> touch passwd.srpv > openssl srp -srpvfile passwd.srpv -add johnsmith@mdc.org Enter pass phrase for johnsmith@mdc.org: Verifying - Enter pass phrase for johnsmith@mdc.org:

Then, in the SSL configuration file we have to set the directive SSLSRPVerifierFile to point to this file. Additionally, to force the use of non-certificate TLS-SRP cipher suites:

SSLSRPVerifierFile /thepath/passwd.srpv SSLCipherSuite "!DSS:!aRSA:SRP"

We can also show a TLS-SRP session on localhost, using two terminal instances (client and server) and the OpenSSL command line tools. First we start the server component:

> sudo openssl s server -nocert -cipher SRP -srpvfile passwd.srpv -accept 443 Using default temp DH parameters Using default temp ECDH parameters ACCEPT

Then we connect the previously registered user from the client component:

> openssl s client -srpuser johnsmith@mdc.org -cipher SRP -connect localhost:443 CONNECTED(00000003) Enter pass phrase for SRP user:

If the password is correct both server and client will show details on the session, and allow the two terminal instances to communicate over the secure channel. The conclusion is that TLS-SRP is already there, and even if most of the servers and cloud platforms still do not offer its cipher suites, it is only a matter of time until it will be generally available. While websites and browsers would probably be less prone to adopt it, mobile apps backed by cloud services with pre-registered users are the perfect adopters.

6.6.3 Setup and Key Distribution

First, we need to assume that there is some kind of protection mechanism in place on the mobile client, so that encryption keys can be stored securely. For this, we adopt the security solution presented in chapter 4. That is, there exists a local authentication process on the mobile client where the user password is used to securely generate a key that encrypts what we call a *storage key*. This symmetric key is in turn used to encrypt other key material of the user on the mobile device. Successful decryption of the storage key, results in a successful local authentication and access to the encrypted user data. Second,there must be a way to build trust between client and server. Even though platforms like App Engine offer a SSL/TLS based secure connection, this only guarantees that we are connecting to an application running on App Engine, not specifically our server application. Therefore, we propose to adopt a communication protocol that provides mutual authentication, preferably without the use of SSL certificates. Reasons not to use SSL certificates are the cost of these certificates (for some discussion on this topic in the context of MDC, see [73]) and criticism towards the Certificate Authority Trust Model (see for instance [112]). Since all users already have a password-based account, the Secure Remote Password Protocol (SRP), standardised in RFC 2945 [139], would be a reasonable solution. This protocol allows mutual authentication and secure key exchange, while being resistant to on-line brute-force and Man-in-the-Middle attacks.

Once the mutual authentication is successful and a symmetric shared key is created through SRP, it can be used to encrypt or sign the public keys and form definitions downloaded from the server, so that their authenticity is verified also without certificates. Such keys can then be securely stored on the phone encrypted with the user storage key. The public keys can subsequently be used also to verify the integrity of form definitions or other data loaded on the phone through other channels, but signed by their source.

Finally, at set-up time, a special public/private key pair is created and published on the server: the *back-up key*. This key pair is used solely for back-up purposes. All the data encryption keys used in the collection process will always be encrypted with the back-up public key, while the private back-up key will be encrypted with a strong symmetric key and stored on the cloud. This symmetric key will then be split up and distributed to a number of participants in the project (most likely the project organisers), according to the Shamir's Secret Sharing algorithm [120]. The reason is explained in Section 6.6.5.4.

6.6.4 Form Upload and Sharing

Once the public keys and local authentication are in place on the mobile client, we can define a different approach to the encryption of filled forms than that used by ODK. As already mentioned, their current solution generates a new symmetric key per form; encrypt the form with this key and the symmetric key with the (single) public key that was attached to the form definition. Since we want to allow for more than one public key per form definition, in order to enable sharing with multiple data viewers and back-up, this approach would incur a huge overhead both in terms of transferred data and processing power. The most straightforward solution is to introduce a symmetric master key to replace the public key encryption and reduce this overhead. We have mainly two alternatives for that:

1. **Form Definition Master Key:** For each downloaded form definition, one master key is created.

2. **User Master Key :** A master key is created once at user registration time and used to encrypt all form submissions.

In both cases the master key(s) are encrypted with all public keys attached to each downloaded form definition (hence all data viewers and form managers allowed to see the form submissions) and the back-up key, and uploaded to the cloud. Locally the master key(s) will be protected with the user storage key and the local authentication scheme.

Compared to the ODK solution, we lose some security on the mobile client, because a copy of the master key will be encrypted with a password based key chain locally. On the other hand, we gain a lot of flexibility and efficiency. The collector can, in fact, decrypt and edit the form before upload, and much fewer public key encryption operations need to be performed. Stealing the phone and brute forcing the password would gain an attacker only the data encrypted on the phone and not yet uploaded.

The main difference between the two types of master keys is the flexibility of deciding who can see data collected with different form definitions. In the first case, the form manager can decide with whom to share the data related to a particular form definition, since different master keys are used for different form definitions. In the second, anyone having a public key attached to some form definition downloaded by a collector, will in principle be able to decrypt all the data collected by that collector. In terms of efficiency, the first case has clearly a bigger overhead in the worst case, both in terms of public key encryption performed on the mobile device and extra storage used on the server. Still, in both cases, a much smaller overhead than the ODK solution.

Once the master key solution is in place, we have in turn at least two options for the key encrypting a specific submission:

1. **Dedicated Submission Key:** a filled-in form instance is encrypted with a randomly generated key and the key is encrypted with a master key as defined previously.

2. **Derived Submission Key:** a filled-in form instance is encrypted with a new key derived from a randomly generated salt and a master key. The encrypted form is uploaded concatenated with the salt in clear.

In this case, the second solution is clearly more efficient on the phone since only one symmetric encryption is performed, while the extra storage used on the server is the same when a salt is as long as a key. The security is equivalent, since recovering the master key would grant access to all data in either case.

Note that all keys used to encrypt data stored on the cloud are not derived from the user password. Hence, brute forcing a user password, would not compromise directly data stored on the cloud.

Figure 6.1 shows an overview of our approach when a Form definition Master key is used, with a Derived Submission Key. For each different user and form definition, a different master key would be used. For details on the secure phone storage, we refer to [46]. We consider this solution the one with the best balance of efficiency and flexibility.

6.6.5 Recovery

The main problem with the claim that the cloud server is not able to decrypt the data stored on it, is that no back-up of the password or private key is available in case the user loses everything. However, in our proposed solution we do not have a single user and its data, but rather an organisation. We can leverage on this to meet the requirement of data availability with a recovery process, in most cases. Note that once a key or password is lost, it should be considered as compromised, and therefore not re-used a second time. However, if the cause of the loss is a malfunctioning of a mobile phone or computer, one might consider it still safe and re-use it rather than issuing a new one, with all the resulting problems.

6.6.5.1 Lost Mobile Password

If a user loses the mobile client password, data on the cloud is still accessible by the form managers and data viewers because the data encryption is not dependent on the password. To allow the data collector to log-in again in the application and continue the collecting process, a password recovery procedure is needed. Following the idea in [46], we save a copy of the storage key on the server. If the user can recover it, then the data on the phone is not lost, and it is enough to choose a new password to encrypt the storage key with on the mobile device, to re-enable the user account. Here is where the local administrators come into play. The system can detect the local administrator automatically through GPS location, or by pre-setting it on the server, and back up the storage key at registration time, with the local administrator public key. The user can then personally contact the local administrator to recover the account on the mobile phone. The process will most likely be a manual one, because any automatic solution would risk exposing sensitive information to the cloud server, and we have the advantage of personally verifying that it is the actual data collector that needs help, and not an attacker trying to gain access to the phone.

6.6.5.2 Lost Phone

In this case, the local administrator resets the password of the user on the server, who can then register with a new phone. However the registration process must be repeated:

all form definitions and public keys downloaded again, and new master keys created and uploaded. The old master key is considered compromised if a phone is lost. If not, a form manager might be contacted to try and retrieve the master keys. However this process might be so complicated and could introduce so many issues, that it might not be worth the effort.

6.6.5.3 Lost Form Manager/Local Administrator Credentials

If the password is lost or forgotten, an authentication protocol based on the public/private key pair can be performed, that grants access to the blocked account and a new password can be set. This implies that once a public key is published, it cannot be changed without contacting the site administrator. Otherwise an attacker who stole the password could also change the public/private key pair, making the recovery process impossible, thereby compromising the system.

If the private key is lost, the site admin should be contacted immediately so that the account can be locked and the password reset. The new password should then be sent via a side channel to the form manager that can generate and publish a new public/private key. Collectors should be warned of the public key change, and renew the master keys associated to the form definitions of this form manager.

6.6.5.4 Lost Site Admin Credentials

If the site administrator account is compromised, usually we would have a disaster situation similar to that of a user of a secure cloud storage provider losing both password and private key. Someone impersonating the site administrator might delete users and lock out form managers and data viewer. However, the data is not lost or compromised because the site admin does not have access to it to begin with, and we still have a recovery option: the back-up key pair. The private back-up key is encrypted on the server with a secret key shared among project managers, and it is therefore necessary for a given number of people involved in the project to collaborate in order to recover it. In fact, Shamir's algorithm allows to split a secret key in n parts, while deciding a threshold of at least $t \leq n$ participants needed to reconstruct the secret. Once the secret key is reassembled, the decrypted back-up key pair can be used to reset and recover the site admin account. The security of this key pair is therefore proportional to how many people are needed to reconstruct the secret, and the bigger the project, the greater effort might be necessary to severely compromise the data and the server.

6.6.5.5 Data Recovery

In all the situations mentioned above, there is always a way to recover all the encrypted data stored on the cloud. In fact, in the worst case scenario, a copy of all the master keys used for data encryption is always stored encrypted with the back-up public key, that can be recovered only with the collaboration of a finite number of person involved in the project. The only data that can be lost is that stored on the mobile phones that was not yet uploaded on the server, if the phone is compromised or lost.

6.7 Discussion

In this section we discuss whether the approach we describe actually satisfies the requirements in Section 2 and we compare it mostly with the ODK solution, since we target specifically RMDC systems.

6.7.1 Data Protection and Security

The encryption scheme we have presented in Section 6 provides strong security in the cloud, while still allowing for data recovery in most situations. We have seen how the collectors' password is not linked in any way to the data encrypted in the cloud, and no cryptographic operations are performed on-line, but only locally on mobile devices or users' private machines.

This means that having only the encrypted data from the cloud does not allow a simple brute force attack to succeed easily, since no password derived keys are used, and the cloud provider does not learn any sensitive information useful for decryption. In order to access (in clear) even only the data collected by a single collector, a considerable effort is required: the phone should be stolen, the collector password guessed, the master keys recovered from the phone; and the correct encrypted forms retrieved from the cloud. A local administrator might be able to carry out the first steps easily, but they would still have to gain access to the data on the server, which they do not have the permission to read, nor do the collectors. We consider the flexibility and efficiency one gains by using symmetric master keys rather than only public keys to encrypt submitted forms, worth this risk.

User accounts might be compromised if the user database with password verifier is stolen, and a successful brute force attack exposes the passwords used to generate the verifiers. If care is used when generating the verifiers, this should not be easy, but it is a possibility.

Hence, if the site administrator account is compromised, changes to the user permissions and accounts can be performed, but the site administrator does not have access to any master key, so data are still safe. This is the main reason to split the key to the back-up private key among many project participants, so that we do not have a single point of failure.

An elaborate attack would consist in compromising the site admin and a form manager account, and replace the form manager public key on the server with the attacker public key, although this should not be so easy as mentioned in Section 6.6.5.3. The collector would then encrypt the data they collect with the attacker's public key, who would now be able to decrypt it. However, this would also trigger an heavy process of key rotation for all collectors using form definitions coming from the compromised form manager, and this should not go easily unnoticed.

Finally, compromising a collector account, which is probably easiest since the password is often typed and implicitly stored in a mobile device, would not accomplish much. Data collectors do not have permissions to download data, although their master keys are stored on the phone. The worst that can happen in this case, is either that data might be forged by an attacker before upload or that data still not uploaded and stored locally is compromised. If encrypted data from the cloud are also available to an attacker, however, all data the collector has uploaded might be exposed, but this sensibly increases the difficulty of the attack.

Notice that compromising the data viewers or form managers machines where their private key are stored, would indeed compromise most of the collected data, but the protection of those keys is outside the scope of this chapter.

6.7.2 Availability

In Section 6.6.5 we analysed different situations where data recovery might be necessary and showed how, in most cases, data can actually be recovered.

By using the same sharing technique as most existing cloud providers, data can also be shared with data viewers and form managers, as long as their number does not increase too much. We have not tested this in practice, but it is clear that a limit of our approach is the number of public keys that can be attached to a form definition.

On the mobile side, collectors are now able to read and edit the form they filled in before uploading, even though encryption is performed on the data stored locally.

6.7.3 Efficiency

The difference in processing resources used on a mobile phone with a Form Definition Master Key and a User Master Key based approach, as discussed in Section 6.6.4, is not that big. The amount of public key encryption operations performed in the first case is roughly equal to the amount of operations performed in the second case, times the number of downloaded form definitions. This number is fairly small, considering also that form definition downloads are not frequent operations. However, in the first case we also gain greater flexibility to control with whom we share the data, providing a better and more secure sharing mechanism. In any case, the total amount of public key operations is significantly lower than it would be by adopting the ODK strategy, where we have one such operation per submission. Also, using a Derived Submission Key reduces the number of encryption operations per form submission to one symmetric encryption, again a much lighter solution than ODK.

The extra storage space needed on the server side is also drastically reduced compared to the ODK solution. Assuming that RSA keys are used, and one was to follow the current recommendations for minimum key size [128], ODK would use at least 1024 bits to encrypt the symmetric key of each form submission. By using symmetric encryption instead, we would need at most 256 extra bits per submission when using the AES with the maximum key size available, both when encrypting with a Dedicated and a Derived Submission Key. Besides, only the master keys will be encrypted with public keys, and their number is negligible when compared to the number of total submissions. On the mobile side the storage overhead is more significant, since also the master keys have to be stored, and more public keys are used than in the solution ODK offers. New keys are added to the user storage at each form definition download.

6.7.4 Integration

By using only standard cryptography primitives, the implementation of our proposed solution should be compatible with most smart phones and application servers. Some changes might be necessary to existing systems on the server side to accommodate a new authentication protocol that provides mutual authentication, like SRP, and a more advanced access control solution. On the client side, previous experience [46] showed that it is possible to integrate security with minimum changes to the application code, but in the case of smart phones, different technical solutions might have to be considered.

6.8 Conclusions and Future Work

We analysed the problem of secure cloud storage in the context of remote mobile data collection and proposed an approach that could provide a relatively straightforward solution to data protection, sharing and recovery. Standard symmetric and asymmetric cryptographic techniques are employed, which makes it feasible to implement in practice and compatible with most existing systems and technologies. At the same time we tried to minimise heavy resource consumption on the mobile devices and additional storage use on the server. Finally, leveraging on the hierarchical nature of a project organisation we can enable data recovery also in cases where in a user-centric model data would be lost because of the lack of a secure root of trust.

Future work will be directed toward an actual implementation of this system and other problems not discussed in detail in this chapter. The main goal of a prototype implementation will be to test how easy it would be to integrate our approach in existing systems like ODK, and to test the maximum number of public keys that can efficiently be handled in the system, before more advanced solutions like a cryptographic file system as in [53] should be considered.

Research challenges abound. An efficient work-flow to change all master keys of the collectors after a form manager lost the private key should be defined. Bi-directional data flow might also be a requirement in future RMDC systems. Right now they are aimed at simple data collection, where the data is mostly static and flows in one direction: from collectors to data viewers, who download the data on their private machine and analyse it separately. Later there might be the need to analyse and use collected data in real time on the field, so that the data flow will have to be bi-directional, and data collectors might also be data viewers. With our current approach collectors might actually be able to download only their own data, since they store the master keys on their phone. In addition, a more advanced book-keeping of encrypted data will be necessary. Efficient sender authenticity, accountability and data de-duplication mechanisms for low-resources setting projects are also needed in RMDC. These problems still need to be further investigated.

7 Application Provisioning

This Chapter is based on the candidate's previously published work as a book chapter in the "mHealth: A Technology Road Map" book and a paper published at at the International Conference on Availability, Reliability and Security (ARES 2012). The book and the paper are available at [1, 73] respectively.

7.1 Secure Application Distribution

In this section, we present some secure methods for preserving application integrity during distribution and installation. This step is critical in order to guarantee the integrity of the security mechanisms implemented in the application and therefore their correct operation. We discuss first the secure distribution and verification of a Java ME client application and later the secure provisioning of an Android client application.

7.1.1 Secure Application Distribution for Java ME Based MDC Systems

A secure channel for an application delivery is a critical step towards building a secure architecture for MDC Systems. Without this security feature, there is no guarantee that the application being downloaded is genuine. Someone might have tampered with it, turned off its security features, or added a malware in it, and we do not have any way of detecting that. Moreover, since we decided not to use the Public-key infrastructure (PKI) for server public-key distribution and verification (as it is discussed in the Chapter 6 under the section 6.6.1), we need an alternative light-weight secure distribution method. We first briefly review how Java ME applications are organized and installed on a mobile phone.

The Java ME platform provides a digital signature verification feature against with installed certificate authority (CA) root certificates. Typically, Java ME applications consist of two important file: a Java Archive (JAR) file and a Java Application Descriptor (JAD). The JAR file contains the main application, and the JAD file describes JAR file using a set of attributes. The application can be installed by sending the JAR file directly to the mobile phone, through over-the-air (OTA), Bluetooth, WiFi or cable, or it can be installed through the JAD file. In the last case, the JAD file will have to contain some mandatory attributes that must match with those in the manifest file contained in the JAR, plus optional and custom attributes. Among the required attributes is the URL of the JAR, so that the phone can download it automatically by generating a request and sending it to the specified URL in the JAD. The application owner can sign the JAR file with a code signing certificate, and add the signature as an attribute in the JAD file. This allows to verify that the Java Archive (JAR) file downloaded is indeed the same indicated in the JAD file, and that the entity distributing it, is a trusted one. Moreover, code signing protects the application by allowing only signed software to update installed application, so that it cannot be tampered with after installation. Thus, signing the JAR file containing the client application, is a necessary condition to guarantee any kind of security. Therefore, code signing is an important element of the secure software delivery model but it is not a complete solution. The lack of JAD file attributes integrity checking method on the client side gives an attacker an opportunity to manipulate the attributes during installation process. Therefore, in addition to code signing, we need a secure software delivery architecture which also provides a secure way for server public-key distribution. In order to achieve this, we followed a concept similar to that of the software delivery model of Software as a Service (SaaS). Unlike SaaS data architecture, the application vendor does not store tenants specific data except the customized JAD file with tenant public-key and other security attributes. The application vendor does not control any data flow from client to the tenant server after the client application installation is over. As shown the in figure 7.1, the secure software and key delivery process starts with the tenant application server. The tenant creates its own private and public key using simple key generator and add the public key into openXdata client JAD file attributes. Once the tenant finish customizing the JAD file, it submits it to the openXdata server. The openXdata server verify the JAD file attributes and make it ready for installation over OTA. The sequence diagram for software and key distribution is shown in 7.2.

The tenant could register a project specific URL for download on the Application Vendor server and define some custom attributes that the Application Vendor should generate and put in the JAD file for each download. Thus, both JAD and JAR file can be downloaded securely from the Application Vendor server. After a successful installation, the client will have enough information to establish a secure connection with the project server directly. The drawback with this approach is that the Application Vendor must be trusted with potentially sensitive information, and the URL of the JAD must be sent to the client securely, or an attacker could redirect the download to a different JAD file with different settings that could compromise security. One way to mitigate this type of threat can be to add a generic URL like `https://m.openxdata.com/` in the signed manifest file, while the JAD contains only one non-signed attribute, i.e, the

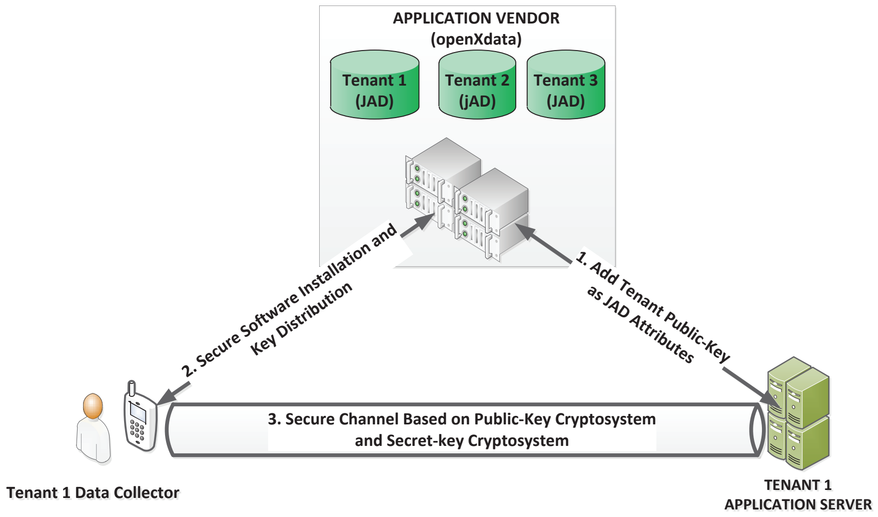

APPLICATION VENDOR
(openXdata)
Tenant 1 (JAD)
Tenant 2 (jAD)
Tenant 3 (JAD)
2. Secure Software Installation and Key Distribution
1. Add Tenant Public-Key as JAD Attributes
3. Secure Channel Based on Public-Key Cryptosystem and Secret-key Cryptosystem
Tenant 1 Data Collector
TENANT 1 APPLICATION SERVER

Figure 7.1: Centralized Secure Software Delivery and Key Distribution Architecture

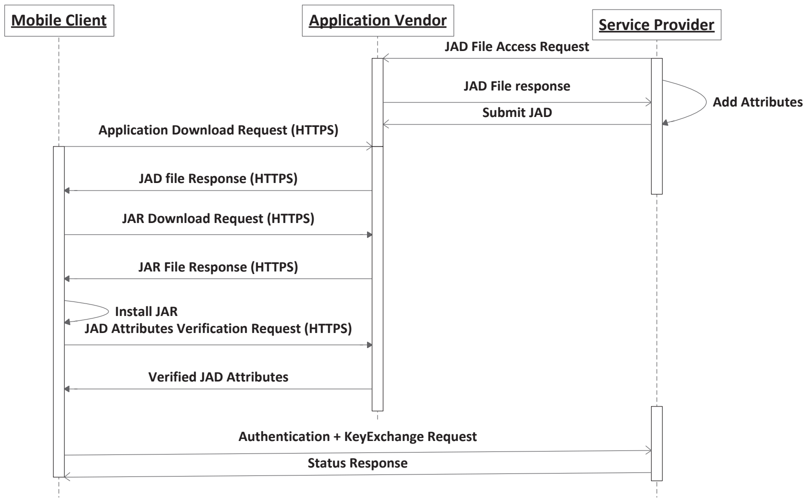

Mobile Client
Application Vendor
Service Provider
JAD File Access Request
JAD File response
Add Attributes
Submit JAD
Application Download Request (HTTPS)
JAD file Response (HTTPS)
JAR Download Request (HTTPS)
JAR File Response (HTTPS)
Install JAR
JAD Attributes Verification Request (HTTPS)
Verified JAD Attributes
Authentication + KeyExchange Request
Status Response

Figure 7.2: Sequence Diagram for Secure Software Delivery and Key Distribution

project name. At this point the settings can be downloaded by the application directly after the installation at the URL `https://m.openxdata.com/project_name`, after the user manually verified the project name.

If the centralized distribution architecture is not available, one might use a challenge-response protocol using pre-shared secret (PSK) that allows to authenticate simultaneously both client and service provider server. A one-time activation code could be distributed to the collectors and used to verify the service provider specific attributes including cryptographic public-key after client installation. In this case the keys and settings could be downloaded directly from the service provider server instead of the application vendor. A drawback of the challenge-response approach is that it requires one time activation code distribution to data collectors in advance especially if there are thousands of collectors, it might be cumbersome. So, a third option, which is also used today, is to deliver the phones to the collectors pre-configured by someone responsible for this specific task, so that installation can be done manually.

7.1.2 Secure Application Distribution Android Based MDCS Systems

Keeping the integrity of an application during installation with its pre-set contents in a distributed environment is imperative to the rest of proposed secure solution. The major mobile OS providers enforce integrity checking through signature based system. However, application provisioning is one of the areas that the two rival platforms Andorid and iOS differ. The iOS built-in security model enabled Apple to distribute applications through a single channel [6]. The integrated security architecture of hardware and software within iOS devices relies on the public key infrastructure (PKI) with Apple as a top Certificate Authority (CA). Figure 7.3 shows the bottom-up signature based chain validation technique from initial boot-up to iOS Kernel loading which takes care of verifying third-party app signature before installing. This makes application distribution safer than most other solutions.

The iOS kernel make sure that apps signed with Apple issued certificate can run on iOS device and prevents unsigned apps or apps signed with self-signed certificate from installing. Apple issues a code signing certificate to an individual or organization after verifying their identity through different ways. Then, application can be signed with issued certificate and distributed through the iTunes app store. There is an additional app checking process that the Apple store enforces before the app is shown to the public. The app passes through an automatic rigorous test following some check list. As of this writing, the check list and how it is enforced is not public, but through cumulative experiences from several developers, there are app performance, memory leak, malware, and signature checking. The overall built-in security and permission enforcement allows to build a trusted relationship between developers, distribution channel, and consumers.

The provisioning of Java ME applications reminds in part of iOS, where code must be signed by a valid CA in order to be installed on the phone. Actually, apps can still be unsigned, but then they would not get access to critical system APIs. In fact, different levels of code signing certificates are needed to allow an application to use different APIs offered by the mobile platform. However, no single distribution channel or appstore exists and applications can be provisioned freely *Over The Air (OTA)*.

Android completely depart from the idea of bottom-up built-in security model, controlling, and validating app distribution channel. The main principle for departing from this idea is that with less restriction developers can build apps faster and put them into market in a shorter period of time. On similar history, code signing is also often been reported as the main reason for the the limited success of Java ME applications. Prac-

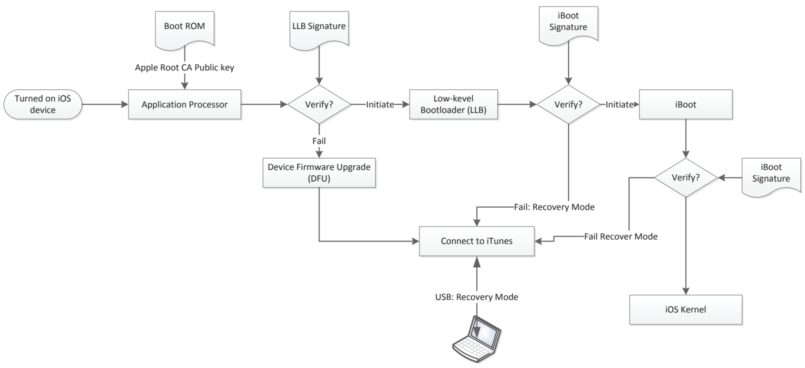


Figure 7.3: iOS Security Model.

tically, this principle does not seem to be working as expected. If we look at some statistics, by March 2009, Android Market started with hosting 2300 Android apps and on May 2013, the number of apps on Google Play was estimated to 1,000,000 with 48 billion downloaded to Android devices [1]. On the other hand, Apple started with hosting 500 apps in July 2008. As of July 2013, it was hosting 900,000+ apps with 50 billion download which is close to the number in Android [2]. In conclusion, Apple submission criteria and checking procedures did not affect the number of apps. Therefore, the premise from Android team's security trade off for faster app development principle, does not seem to be justified.

The Android security model spurred other app stores to host Android apps. At the beginning, Android departed from PKI based code signing architecture with a top CA and introduces a self-signed code signing mechanisms for application testing and release. The departure from PKI based architecture helped others to host Android apps in their store. As of this writing, there are more than thirty Android app stores. Some of them are Google play, Amazon App Store [3], Mikandi Adult App store[4], Samsung App Store [5], Android App Online [6], Aptoide [7], and Opera App Store [8].

Android app distribution starts with by digitally signing the application with a self-signed certificate (no CA required). The only requirement is that the code signing certificate should be valid for more than 25 years. The Android security model uses the application signature to ensure the authenticity of the app author at installation time and during app update, and to establish a trust between different apps that are signed with the same key. Applications signed with the same key share the same Linux kernel User

[1]Google IO, [Last Accessed: May 2013],`http://www.youtube.com/watch?v=9pmPa_KxsAM`

[2]Apple's App Store Reaches 50 Billion Downloads, [Last Accessed: May 2013], `http://www.apple.com/itunes/50-billion-app-countdown/`

[3]Amazon App Store, [Last Accessed: April 2014], `https://developer.amazon.com/welcome.html`

[4]Mikandi Adult App store, [Last Accessed: April 2015], `https://mikandi.com/`

[5]Samsung App Store, [Last Accessed: May 2013], `http://apps.samsung.com/`

[6] Android App Online, [Last Accessed: May 2013],`http://www.appsgeyser.com/`

[7]Aptoide, [Last Accessed: April 2013],`http://www.aptoide.com/`

[8]Opera App Store,[Last Accessed: April 2014], `http://apps.opera.com/`

Id, which helps to share resource, permissions, and processes. David et al. [10] perform thorough analysis on collective Android apps from different app stores and have shown Android's current signing architecture does not encourage best security practices and propose some platform level mitigation methods. The Android Development Tool (ADT) is shipped with a default debug code signing key to facilitate code development and testing. Once the code is ready for production, the code must be signed with a release key which needs to be kept in a secure place. The class files, native libraries, resources, and AndroidManifest file pass through a hashing algorithm for instance SHA-1. The output from the hash algorithm is signed with signing algorithm such as RSA or DSA. The signature is packaged together with the class files and resources and an APK file ready for deployment is generated and sent to the Android app store. In order to upload the app to Google Play, Google requires an initial registration through a Google account and a payment of $25 paid through an individual credit card. Some of the third-party app stores are free or require an annual developer membership fee for uploading apps.

Reverse engineering has never been easier when we come to Android apps. There are a number of tools available that make it very easy for an attacker to get into the source code and resource files of the application. Once the installed app is extracted from the mobile device we can unzip the application file and all class files. Generally, it is recommended to obfuscate apps, but even then it is not so hard to get the source code. Traditionally, an attacker can manipulate the source files, add malware into it, repackage it, and sign it with his/her own key and finally publish to the app store. Rahul et al. [104] demonstrated this type of attack and proposed a three detection system level scheme that relies on syntactic fingerprinting to detect plagiarised applications under different levels of obfuscation used by the attacker. This kind of practice has been overwhelmingly high in Google Play before Google introduced the Google Bouncer [9] in 2011. The purpose of the bouncer is to scan apps for known malware, spyware, Trojans, and hidden and malicious behaviour (comparing it with previously analysed apps, and checking new developer accounts against previously known offenders) by using a simulated area inside Google's cloud. Google also introduced a remote kill-switch mechanisms to remove already installed apps remotely. As a result of the bouncer, the number of potential malicious apps has been reduced by 40% [10]. To further strengthen the application validation, Google introduced a client verification mechanism in Android 4.2 (Jelly bean), but later incorporated it into Google play service app (which already exists from Android 2.3, a.k.a Gingerbread). In the end, there is no guarantee that the application being downloaded is genuine. Someone might have tempered with it, turned off its security features, or added a malware in it, and we do not have any way of detecting that for sure. A more recent attack of this type has also been demonstrated, where a legitimately signed file could be arbitrarily altered without invalidating the original signature, and could therefore be distributed as the original genuine one [60].

We propose four possible ways to alleviate this problem without having to change the whole system:

[9]Google Bouncer, [Last Accessed: May 2014], `http://jon.oberheide.org/files/summercon12-bouncer.pdf`

[10]Potential Malicious Apps at Google Play Store, [Last Accessed April 2013], `http://googlemobile.blogspot.no/2012/02/android-and-security.html`

1. As a first option, we can adopt the standard approach and distribute the application through the platform vendor infrastructure, for instance, *Google Play Store*. Even if this approach comes with security challenges related to storing apps in the play store, it provides an excellent opportunity to distribute apps at scale, and it does guarantee the application integrity to a reasonable degree. The advantage for the users is particularly significant as standard channels and procedures are used. Besides it does not require to enable the "Unknown Sources" option for installing apps from third-party application distributors, which could constitute a potential entry point for other malwares. The inherent security challenges that come with the *Play Store* can be minimized by uploading a generic mobile data collection app on the Play Store and once the application is installed on the device, we can establish a secure channel with the application server and sync all important credentials and keys onto the device. The security framework fits well for establishing a secure tunnel and sync the credentials. The problem with this approach is clear when we consider that most projects require a customized MDCS. In this case, each customization should have to be packaged, resigned and uploaded on the Play Store, with all the problems we have already discussed. In addition, having many customized versions of the same MDCS on the Play Store would actually make it easier for a purposely compromised version to be confused with the genuine one and be downloaded by mistake. In this case, other approaches may be more appropriate.

2. Manual installation is the ideal choice of customized secure app distribution. Apps delta update (aka a patch) can be pushed to client by the project server in coordination with Google Cloud Messaging system (GCM) [11]. The Android platform enforces the proper signature checking procedure during update. The main drawback of this approach is that it will be cumbersome installing the app at the initial stage of a large scale project. An alternative solution might be necessary for this type of scenario.

3. Bypassing the app stores altogether and distributing apps through the vendor websites can also be an option. Application vendors such as ODK, mUzima, or openXdata would act like a trusted central CA. The application is going to be signed, even self-signed, for integrity checking, but we are not relying on this signature to trust its authenticity. Rather we are relying on the fact that we download the app from the developer site in a secure manner, and that we are actually connecting to the correct download site. The download link can be sent through a secure side channel as SMS or mail, and now the root of trust becomes the combination of this secure link and the SSL certificate of the application vendor signed by a trusted CA pre-installed on the device. The downside is that the application cannot be customized, and no project specific configuration added to it, because then it would have to be signed again and downloaded possibly from another site we might not trust. The same problem as the appstore would arise again. However, the advantage is that the application is not compromised and it will behave as expected, which is the fundamental requirement to be able to enforce any other

[11]Google Cloud Messaging, [Last Accessed: May 2015], `https://developers.google.com/cloud-messaging/`

security mechanism later on.

4. A fourth option might be to have some MDCS System vendors become specialized app stores, where they guarantee for all customized versions of their systems. This requires to set up a model similar to that of the iTunes, but the chances of achieving this within the foreseeable future are slim. Tenants who are customers to the application vendor customize the app based on their needs and sign it with their own public/private key pairs. The tenant submits the custom app to application vendor hosting server and give the download link to the collectors/users through a trusted side-channel (SMS or mail).The app can then be downloaded through a secure connection from the Application Vendor server and contain for instance the project server certificate. After a successful installation, the client will have enough information to establish a secure connection to the actual project server directly. The application vendor does not store tenants specific data except the customized app and also does not involve in client-tenant interaction after the client application installation is over.If the centralized distribution architecture is not available, one might setup their own secure channel.

7.2 Conclusion

Our experience with partners projects including ODK and openXdata, is that an MDC system is designed to be generic on purpose, so that any organization interested in using the system is required to customize the application, particularly the client application, in order to meet their requirements. This model helps the application vendors such as ODK and openXdata to focus on the common functionalities of the system that is needed by all organizations. As a result, ODK and openXdata can continue the development with low resources and funding. From this, we learn that each organization who customizes the main application like ODK is expected to package, sign, and distribute their application to the remote clients. In this scenario, manual installation fits well if it is a small scale project. However, for rolling out at scale for instance nationwide, a comprehensive distribution channel is needed. The first approach in the list 7.1.2 may fit with this scenario which is using the normal distribution channel such as *Play Store*. At the same time, the organization may need to make the application in the *Play Store* uniquely identifiable so that the users can download the correct app easily. The organization also must pay and acquire Google developer account for uploading the customized application. If no customization is involved, the organization can use an existing MDC app provided the app vendor. This may cause problems especially when the organization needs an immediate fix because of a bug or a security flaws. In that case, since the app vendor is the app signer which means they are the only one who can push an update, they must respond in timely manner. However, since the app vendor usually has few resources, the response time may be long or even they may even not respond at all. The pros of this approach are that the user is not required to enable settings such as download apps from "Unknown Source" for installing applications. Once the app is installed, all important credentials and keys are securely downloaded through our security framework. As an alternative approach, the main application vendor can host the customized applications and provide a distribution channel as described in third place of section 7.1.2, but this still requires user involvement on enabling the settings. Concluding, there is no best alternative, but a set of viable solutions that fulfill different needs and project types.

8 Discussions and Future Works

8.1 Discussions

In this chapter, we try to summarize the experiences we had while working with MDC Systems, and discuss the most critical security concerns around these systems. We have followed their development from the first Java ME based systems until the latest cloud-based solutions. Unfortunately, security was and still is a major issue in many of these systems. We have designed our solutions in close collaboration with the main players in this field including the openXdata and ODK, and have tried to develop a framework that was both secure and user-friendly, and that could easily be integrated in a transparent way in MDC systems. The secure client storage has been the area where we were able to make most progress, and now openXdata has a secure client version in production and secure version of ODK is under testing.

Most of the problems we discussed in this dissertation are general security issues that can easily be applied to most mobile client-server applications. However, we had to look at them from a different perspective, due to the specific constraints and requirements mobile data collection is subject to. This meant that standard solutions not always were applicable and we had to find an acceptable alternative. Here we discuss more in detail some of the problems we faced and how we tackled them.

Most of the challenges we faced during the implementation required finding the right balance between flexibility, efficiency and usability, while not compromising security. In general, we decided to give more emphasis to flexibility, in order to create an API that is easy to use and integrate with different clients, at the cost of some efficiency. In the following sections, we discuss some of issues we consider to be highly relevant.

8.1.1 Cryptography API Providers

8.1.1.1 Java ME

Early versions of Java ME did not support a cryptography API. However, since the introduction of MIDP 2.0, the Security and Trust Services API (SATSA) has been developed and added to the Java ME platform as an *optional* package that provides some basic cryptographic primitives. Besides, since it is implemented as part of the phone libraries, its use does not affect the memory footprint of the application. Unfortunately, very few low-end mobile phones actually support it. On the other hand, Bouncy Castle (BC)[70] provides a flexible lightweight cryptography API which is extensively used in Java ME applications. Since it is an external API, it allows us to develop device independent solutions, but its libraries can add a significant memory overhead. In order to allow for future compatibility, we opted for an hybrid solution. The security framework provides an interface that defines the required cryptographic operations, but leave the actual implementation open, with BC as default provider. However, if the phone supports the SATSA package, the security framework can automatically switch to that implementation. So, even though memory footprint is not reduced (BC is always loaded anyway), one can gain in performance by using the phone built-in libraries. Using two different implementations, means also that we are forced to use only algorithms supported by both libraries. In particular: RSA for public key encryption, AES in padded CBC mode with initializing vector (IV) for symmetric cryptography, SHA1 digest, Hash-based Message Authentication Code (HMAC) based on SHA1 digest. Only BC provides an adequate Pseudo Random Number Generator (PNRG) and Password Based Encryption based on PKCS#5.

8.1.1.2 Android

Android platform includes two major open source crypto library providers, OpenSSL (C and Assembly implementation) and Bouncy Castle (Java implementation). Bouncy Castle has been shipped with Android platform in a cut-down and crippled state due to device memory constraints. As a result of this, the external Bouncy Castle API for Android has been re-packaged and re-named as Spongy Castle to avoid class loader conflicts with the old built-in Bouncycastle API. The Spongy Castle must be included with Android project as external library, while OpenSSL is supported and maintained as a native library. OpenSSL is also available in iOS which make solution porting easier. Developers must however know how to use these libraries properly, or security might be compromised[1]. As it is noted in [33], there was a plan to remove Bouncy Castle and replace it with the OpenSSL, but after Android 4.0, i.e. API level 15, Android has updated the Bouncy Castle library from version 1.34 to version 1.46 and it is accessible through java.security.* APIs. In Android 4.2 and onwards, the default Bouncy Castle-based SecureRandom implementation is changed to OpenSSL-based SecureRandom generation due to deterministic behavior of Bouncy Castle implementation. Therefore, we strongly recommended OpenSSL as a crypto provider to build a secure solution in Android platform.

[1]Java Cryptography Architecture (JCA), [Last Accessed: May 2015], `http://android-developers.blogspot.com.au/2013/08/some-securerandom-thoughts.html`

8.1.2 Key Generation

8.1.2.1 Java ME

A critical issue when using cryptography on a mobile phone is the generation of good random keys, since mobile phones do not have good sources of entropy [24], and even if they have, Java ME might lack the necessary libraries to access them. In the security framework, we generate a strong seed on the server and send it securely to the client whenever possible, so that strong cryptographic keys can be generated. Every user will have their personal set of seeds stored encrypted in their key store, so that the PNRG can be seeded also at boot time, and in a different way for each user. This solution avoids putting the burden of generating the seed on the user by pressing random keys or playing a game, or turning on the camera or the microphone to collect entropy, as it has been suggested in the literature.

8.1.2.2 Android

As of writing, vulnerability on Android SecureRandom generator has been identified and Google officially acknowledged it and posted a short term fix [2]. According Google's report, due to improper initialization of the underlying PRNG, both system-provided OpenSSL PRNG and Java Cryptography Architecture (JCA) based key generation, signing, and random number generation may not receive cryptographically strong values. The remedy is to explicitly initialize the PRNG with entropy from /dev/urandom or /dev/random and re-evaluate if a user needs to use a new key or random values when it uses JCA APIs including SecureRandom, KeyGenerator, KeyPairGenerator, KeyAgreement, and Singature. It is also recommended to generate a random AES key upon first application launch and store it somewhere secure such as internal memory. The security framework further enhance the randomness of the key by using a seed generated on the server side.

8.1.3 Secure Communication

The security framework is designed to be flexible and support both HTTPS and the SRP protocol for authentication and key exchange proposed in chapter 4.4.2.2. If the application is setup to use HTTPS, the security framework will behave as normal and let the HTTPS connection handle the transaction. If however it is not HTTPS, any request headers or data written to the connections output stream will be encrypted prior to being sent to the server by using the secure communication module presented in chapter 6. The secure communication module is designed to hide the underlying complexity from the programmer and allow to communicate with the module transparently without requiring security knowledge and security API experience. This makes for easy and transparent integration into existing systems. We are able to create a secure tunnel by changing only two lines of code in the existing openXdata and ODK clients.

Initially we had thought of exploiting the fact that data is encrypted on the phone, and send it as it was, to avoid further encryption and decryption operations. However,

[2]Java Cryptography Architecture (JCA), [Last Accessed: April 2015], `http://android-developers.blogspot.com.au/2013/08/some-securerandom-thoughts.html`

that could not be done without significant changes to the existing client code, and without exposing many cryptographic operations to the developers. Not to mention that the same key used for the storage would be re-used for transmission, raising security concerns and key management issues. Hence, even if this could give better performance, it could also affect the security of the API and its usability. We chose, therefore, to simply wrap the data sent from the client in a secure connection, which, despite some extra traffic, allows also for a complete decoupling between the secure layer and the client requests.

The secure communication module of the security framework uses TLS for data protection and SRP for authentication and key exchange in combination as standardized in [25]. One of the drawback with this approach the two round trips for user authentication and key exchange required for SRP protocol and even more, the SRP needs user credentials every time when the user authenticate to the server. This means either we always ask the user to put in user credentials or cache for later use on the device. The former one has significant impact on usability and the later may compromise security. We have work in progress for solving this challenge using *Certificate Pinning* and it further discussed under future work in Section 8.2.1.

8.1.4 Secure Storage

The storage has been designed to accommodate typical scenarios in mobile data collection. In particular that multiple users should be allowed to use the same mobile device, that the same user can use multiple mobile devices and that Internet access might always not be available. This means that mobile devices can no longer be considered private or personal to a user and that most of the data collection might have to be done off-line. These requirements are very specific to mobile data collection systems and the secure solution is devised to address these requirements in satisfactory manner. From a security perspective this translates into the following concerns:

1. A mobile device must store some identification token to authenticate users off-line.
2. If a user loses the password, other users on the same device and their data should not be affected.
3. If users change their password on the server, possibly from a web application, the access to the mobile device should not be compromised.
4. Even if the password is lost, it should always be possible to recover the encrypted data stored on the mobile phone by some authorized entity.

A secure scheme that satisfies all the storage requirements is described in 5 and implemented in the secure framework as secure storage module, which offers tools to secure individual users data in personalized secure storage. register a new user so that a new personal secure storage is initialized according to such scheme. The data is encrypted with symmetric encryption, and the encryption key is protected by a password-based key. This means that losing the password does not prevent access to the data if the data encryption key has been saved, for example, on the server. Notice, however, that the overall security of the data still depends on the strength of the password, and as long as off-line local authentication is required on the mobile phone, and smart cards are not supported by the phone, this is a problem that cannot be solved. But, with Android, the risks can be minimized by storing the keys in the keystore of the device which some devices provides hardware support and some implement the keystore using software, but the keys are stored at the system level storage which means any third-party applications do not have access to the storage except the one who store it. We can't depend on this feature for rooted device, but the device can be protected from rooting using appropriate measures.

The keystore is one of an interesting area to work on and we further discussed it under the future work in section 8.2.3

Every write/read operation will, respectively, encrypt the input data before writing it in the actual storage, and decrypt it before returning it to the . The secure storage module also takes care of dynamic checking whether the current user has permissions to write/read in that storage and handles the corresponding keys. All of this happens completely transparent way for the client.

The drawback of this approach is that the user has no control over the data encryption, so, every time something is read or written from the secure storage, a cryptographic operation is performed. This can be a computational overhead if a search must be done across the stored data, since several decryption operations are required. This happens, for instance, when a menu must be generated to show the users which form values have been saved in the record store. To mitigate this problem, we offer to store the data with a label that describes it. All the labels are stored as meta-data in a separate storage or database table, so that only this list needs to be decrypted to generate a menu, rather than all the records. One advantage, instead, is that the client is not forced to pre-process the data and store it in a specific format or in a dedicated record store. The only assumption we make, is that each user has a dedicated record store, so that a unique key can be assigned to it. This makes the key and permission management much easier for the secure framework.

An alternative solution we tested was to offer methods that took a byte stream and returned an object containing the encrypted stream plus a set of fields to manipulate it, so that the developer could have direct control over the encrypted data. However this idea was discarded because it would have required substantial refactoring in the existing client, and it could have potentially introduced security issues if the data were manipulated incorrectly.

There is a lot more efficient way of making the encrypted data *search query* friendly, increase performance, and visualize encrypted form data in a cloud through field-level encryption, discussed in the future works.

8.1.5 Modularity of the Secure Framework

When designing the framework, we followed a modular and centralized design approach to fulfill the design criteria that are listed in the section 3.1. We tried to make the modules that constitutes the secure framework as independent as possible, so that a client using only the secure communication, would not import the secure storage module and vice-versa or the authentication module can be used alone without having the storage or communication module, thus minimizing the final size of the application. The messages between MDC client and server are securely wrapped. The encapsulation or wrapping process help us to make a transparent integration and no prior knowledge is required to interact with the security framework.

8.1.6 User Authentication Protocol

We adopted the latest version of the Secure Remote Password protocol (SRP-6a) [132] which allows user authentication and key exchange. The SRP protocol allows client and server to mutually authenticate each other without requiring a trusted third party. The protocol is resistant to several attacks as it is discussed in 4.4.2.2. We have a working prototype for remote authentication based on the SRP protocol and the preliminary performance test looks very promising. We leverage the SRP protocol to provide an acceptable guarantee even when a user chooses weak password. This would not have been possible on low-end phone running Java ME applications, with limited or non-existing crypto APIs and low CPU power.

The Authenticator module of the secure framework provides local authentication. We use a password or PIN code for local authentication but we have seen interests from the MDC communities such as mUzima to use FingerPrint technology as local authentication. As of this writing, Fingerprint is at infant stage on mobile devices, not all platform providers or device manufacturer ships the fingerprint technology with the device. There are encouraging signs from Android that the new release Android M may incorporate the Fingerprint sensors in Android device by default. Even we assume that all mobile devices embedded the fingerprint sensors, it actually requires in depth research on whether the technology fits the MDC systems need. Currently, the server authenticate the user using normal username and password, replacing this with fingerprint requires an investigation. This is because the user should use the same credentials for local and remote authentication. Furthermore, is it possible to revoke users when they are not allowed to the access the system, or how can we deal with a compromised fingerprint, is it possible to change it? Fingerprint is one of a unique component that characterize the user and any activity can be uniquely traced back to user. So user privacy is a major issue. Also, how can multiple users register multiple fingerprints on the same device to access their own application data? These and other issues must be addressed before adopting fingerprint technology.

8.1.7 Phone Sharing

Phone sharing is among the functional requirements list mentioned in section 2.3.1 and the security framework has provided a secure solution to share single device among many users. Mobile devices are evolving continuously and we have already mentioned that Android has recently introduced a multi-user environment. So, many challenges we have today might be solved by the natural evolution of mobile devices. But as of this writing, Android multi-user support is a manual process and the setup process requires user involvement. Furthermore, user management is performed on the device and not remotely. There is no public APIs available to access this feature, but if it will be made available in the future, our framework will probably naturally leverage it.

8.2 Future Works

In this section we discuss the work in progress and the future work that we envision will be needed to further develop our framework and meet the future security requirements that naturally will arise from the adoption of new emerging technologies in mobile data collection.

8.2.1 Certificate Pinning

To reduce the overhead of performing the two SRP round trips every time, and assuming that we can store securely certificates on a mobile client, we started looking at another strategy called certificate pinning. With the help of strict-host-key-checking feature, an OpenSSH client can be configured to grant or reject a connection request by verifying incoming request key against known keys list stored in the key store. This idea has been adopted in a certificate or public key pinning. Pinning is an emerging concept of associating a given client with a list of known public keys or certificates. The client then verifies an incoming connection request against the list and grant or reject accordingly. After Man-in-the-Middle attack on Gmail's SSL through a compromised DigiNotar certificate (DigiNotar was one of well know Certificate Authority (CAs) before bankrupting) [87], Google and other organisations have been actively engaged on making pinning as a part of their products. As a result, there are public APIs to implement pinning solution on mobile apps and web browsers. There is one fundamental problem with pinning though. There is no secure way of distributing the certificates or public keys to a client at the beginning of a project in a distributed environment. Basically, there are two ways to distribute the certificate or public key to the client. It can be incorporated into the application before application distribution or it can be fetched after the application is installed. OWASP recommends the former i.e. before the application is installed. However, if we choose the second distribution scheme proposed in the previous section, where no project specific configuration can be added to the app, we have to go for the later approach. That is, to fetch the keys and certificates after the application is installed. We found SRP protocol to be a potential candidate to accomplish this task.

The idea of pinning is a good starting point for future works, especially in the case of mobile applications which are supposed to interact with a certain number of servers.

8.2.2 Secure Cloud Storage

Cloud is becoming popular for systems deployment and among all systems, MDC Systems are one of them. MDC Systems can benefit from fast deployment, minimized cost, and less maintenance, etc. Public cloud is an ideal solution for MDC Systems from a cost perspective, but it comes with trade-offs such as data security. User management and data protection are some of the challenges. Compliance with regulatory requirements such as HIPAA in a cloud-based mobile health solution is another interesting area to look into. As of today, there are some cloud providers such as Amazon that claims the Amazon cloud provide a HIPAA compliant deployment and running environment, but there is very little information publicly available to validate the claim or how Amazon meets the requirements. Furthermore, there is no information on the list of responsibilities that MDC Systems need to address at when they signed up for HIPAA compliant infrastructure.

In the future, we see more and more online collaboration between collectors and data viewers, and scenarios where health personnel will need to access collected data on a real-time basis. Cloud resources will be used to analyse the huge amounts of data on a centrally accessible server, so that anyone who needs it will be able to simply query the server without having to download all the data on some local machine and perform the analysis locally. This implies that our current solution for encryption of the data is not portable for big collaborative projects, and new solutions must be devised. A possible compromise we are thinking about, is to leverage the form definition structure to define which information is sensitive and which can be processed without privacy concerns, so that a selective encryption can be performed and only privacy-critical fields will be protected with strong cryptography. Authentication mechanisms to access the on-line data without having to maintain huge user databases with passwords and access control lists are starting to get mature and widely used. OAuth authorization framework [3] is an example which is already being adopted by ODK. Usernames and passwords might soon become obsolete, and one single account might be enough for a variety of services. This entails some challenges when mobile devices are involved, and even shared as in Mobile Data Collection. Having the device, might be enough to access a service without further authentication.

8.2.3 Secure Key Store

The SIM card (Subscriber Identity Module) is a type of smart card and can be considered as a secure key store. Android provides APIs to access the SIM element but, as of this writing, there is no Android Crypro API exposed to perform cryptographic operations on the SIM card. The network operator owns the SIM element and they have all privileges and control over the SIM. An application signed with the operator signing key may have access to the SIM card and use it as secure element. Another way to store keys and credentials in Android is through its built-in Keystore daemon. Since Android 1.6 (Donut), the Keystore daemon was introduced for maintaining cryptographic keys for system-level apps such as VPN and Wifi connection. There was no public API for accessing the Keystore from third-party apps until Android 4.0 (a.k.a Ice Cream Sandwich (ICS)). A KeyChain public API is added in ICS for accessing the Keystore from third-party apps and store keys and certificates. All Keys and certificates added into the Keystore are associated with their owners through the app user ID and stored in the /data/misc/keystore partition which is outside the application user space. From the security perspective, this a good place to store credentials. However, there is only one Keystore instance exist per mobile phone and all applications must share this instance. On ICS and later, the Keystore is unlocked when the device is unlocked through a pattern, a Pincode or username and password, and it is accessible to all apps. However, on pre-ICS devices, unlock screen was separated from Keystore unlocking and Keystore unlocking is performed through an intent [23, 33]. Despite the fact that Keystore deamon provides a secure store, it does not maintain multi-user credentials. As a result, we leverage the keystore by enforcing access control at the application layer, particularly

[3]OpenID Connect Framework: `http://openid.net/`

when a project is looking for multi-user support per device. It is worth mentioning that third-party application can leverage system level protection for its credentials through keystore and enhance application usability by introducing a single login interface for accessing the device.

With the introduction of Android Pay [50], credentials storage solutions such as secure elements may be easily accessible for third-party applications as well. Even smart card elements such as the SIM card might be accessible for storage purposes. Since secure key storage on the device is a critical component of the security architecture, the future work needs to investigate closely the progress and come up with a better solution. Furthermore, hardware backed or software based Android Keystore in combination with trusted execution environment technology (TEE) [48] such as ARM TrustZone [7] [48] may provide data protection guarantee as it is discussed in section 5.4.2. However, as of this writing, there are few OEMs that incorporates TEE technology on newly released devices and older devices do not have it. Therefore, for MDCS, in particular, the TEE may not be a feasible solution if most devices support it particularly cheap Android devices.

8.2.4 Native vs Hybrid vs HTML5/JavaScript

On the client side, we are observing a specific trend, where many MDC Systems are being implemented as a Hybrid application that is built from the best of both HTML5/CSS/JavaScript and native API. Native apps are still proven to be the best in terms of performance and security, but they are specific to a given platform and requires multiple code bases to run on multiple platforms. HTML5/CSS/JavaScript is promising to bridge the gap and run on cross-platform with a better user interface, but still has limitations on accessing device hardware such as camera and geo-location. Therefore, the Hybrid solution leverages native functionality by embedding HTML5/JavaScript within the web-browser container such as Webkit. This architecture is known to be vulnerable to a range attacks such as Cross-Site-Scripting and needs in-depth security analysis before adopting it as a solution.

Hybrid MDC applications (best of Native & best of HTML5/Javascript/CSS) are getting the momentum and are expected to continue at the same pace in the future. Addressing standard security properties are still challenging particularly in fast changing, dynamic, and fragmented web browser technologies. JavaScript is becoming more popular than ever been before, and most MDCS have started leaning towards heavily use of JavaScript. However, JavaScript by nature is thought to be an insecure programming language, and it has proven weakness. Finding a balanced, secure solution is still a challenging task. We have seen an outstanding works from W3C Web Crypto Group[4], Crypto-JS[5], Node.js Crypto[6], and End-to-End JS Security Library from Google security team[7]. However, it still requires tremendous effort and an insightful thinking from the research community to investigate a given MDC System from performance, security and usability perspectives. It requires a depth in research on how data is stored, accessed and disseminated in browser based MDC.

[4]Web Crypto Group, [Last Accessed: May 2015], http://www.w3.org/TR/WebCryptoAPI/

[5]Crypto-JS, [Last Accessed: May 2015], https://code.google.com/p/crypto-js/

[6]Node.js Crypto, [Last Accessed: May 2015], https://nodejs.org/api/crypto.html

[7]End-to-End from Google, [Last Accessed: May 2015], https://github.com/google/end-to-end

8.2.5 Field Level Encryption

An answer to each question in the form definition has a different level of importance and sensitivity than the others. For instance, as HIPAA specifies a list of attributes called protected health identifiable information (PHI) and requires all electronic transaction and storage to guarantee that those attributes are protected using all security measures. With field level encryption, it will possible only to secure those attributes and leave the rest unprotected. In return, we can easily make a search on encrypted data, visualize the data on the cloud without exposing sensitive data, and gain performance through partial encryption. This is work in progress and requires further investigation on security issues. For instance, how do we label fields protected or non-protected during form design? How do we develop a generic solution that identifies fields in the form and apply field level encryption without prior knowledge of how the form is organized? These issues require a depth in research and we highly recommended as future work.

8.2.6 Privacy in MDC Systems

In general, mobile data collection might evolve in order to make the collected data available automatically available to other services in real time, so that decision might be taken based on the latest information collected also in remote locations, which would therefore be as precise and updated as possible. Privacy issues might then become even more pressing, and privacy preserving and secure data retrieval techniques developed for other areas of eHealth and mHealth might become relevant for data collection as well.

9 Related Work, Conclusions, and Contributions

9.1 Related Work

When it comes to related work, with the exception of one work that resembles our research very closely, we are not aware of any security initiative specifically aimed at developing a comprehensive secure solution for remote mobile data collection systems. There is, however, a vast literature about the many individual problems we have discussed throughout the thesis. The most relevant ones have already been discussed in the chapters dedicated to the framework modules, which consider individual security aspects separately. We will therefore not repeat them here. Let us also point out again that we considered mHealth systems where data is entered by human collectors, and we are not considering the situation where a sensor is feeding data to a remote server or device, or a monitoring program is installed on the mobile device and their correct operation must be guaranteed at all times as in [124]. The collected data is also not thought to be readily accessible to health-care providers, or in general to be offered as a service. These are different areas of mHealth where much research is ongoing, but not directly applicable to our case. For instance, in [136] a solution for secure data storage is suggested, and they also propose to encrypt the data when it is not in use. However, this solution is designed to protect patient records that are retrieved to be read and should not be edited or distributed. Clearly this cannot be adopted in a data collection scenario. The same applies to solutions like those proposed in [84, 123] or in [135] for secure patient's record retrieval.

9.1.1 Comparison with the SecourHealth Framework: a Similar Work from Brazil

On the other hand, we have the SecourHealth Framework[76] which is a security framework proposed by a group of security researchers from the University of São Paulo, Brazil. The SecourHealth framework main objective was to secure MDC project that was deployed in a national family health program in Brazil from 2011-2013. Hence, the scope of the security research focused only on mHealth solutions in Brazil. We found their first publication (which is published on April 2014) completely by chance and found that the underlying approach, the motivations and the requirements (both functional and non-functional) coincided with our first published work [74], which they also refer to as a term of comparison. Unfortunately they did not seem to be aware of our more recent work and did not take that into consideration when writing their paper. This led to some inaccurate statements on their part and results that resembles ours very closely. Since then, we reached out the group and had a constructive conversation on our concerns about the overlapping work which led to the following, more accurate, comparison between the two works. First, we see how the client applications with security configuration are distributed and installed. Second, we compare user authentication methods in these systems. Next, we present how the systems manage data uploading and session handling. Finally, we discuss the SecourHealth framework novel contribution that is perfect forward secrecy using symmetric encryption and compared it with our approach.

9.1.1.1 Secure Application Provisioning

Secure application provisioning is about the method used to guarantee the integrity of the application and its data by making sure that the correct application with the right configuration is downloaded and installed. Having a mechanism to securely distribute the application is an important component of the entire security architecture of MDC systems. This security aspect is not addressed or discussed at all in the SecourHealth Framework. We, on the other hand, have spent considerable time and effort to look for viable solutions both for J2ME and Android based MDCS, as presented in Chapter 7.

9.1.1.2 User Authentication

The SecourHealth framework assumes that the project using their solution can leverage an authentication protocol called Generic Bootstrapping Architecture (GBA). The GBA technology is a 3GPP standard, its specification defined in [127]. This authentication protocol is designed for cellular network and it requires users to register at the network provider and get a valid account. Then, the network provider issues a SIM card which holds user identity information. Therefore, the network provider uses the SIM card as user token and user authentication is performed between the SIM card and the network. This approach has the following drawbacks:

- GBA is designed to work on cellular networks, which means that the user authentication requires strict collaboration with mobile operator for pre-user registration and the issuing of SIM cards specific to the users. Based on our experiences, this approach is virtually impossible for several projects deployed in the field, and therefore only applicable to the specific MDCS they were targeting.
- The user identity and credentials are stored in the HLR (Home Location Register) or on HSS (Home Subscriber Server) of the operator network. These storage is also a place for storing millions of subscribers and customers data. The operator might not allow to access the service remotely and they may get involved in daily

maintenance or update of the system. Furthermore, in normal circumstance operators are profit oriented and they might not be willing to support all GBA related activity for free for longer period.

- Each SIM card supports only one user, hence there is no possibility for device sharing which is one of our most critical functional requirements.

With our security framework, we offer one of the most reliable authentication methods, namely *Secure Remote Password Protocol (SRP)*, a Password Authenticated Key Exchange (PAKE) protocol based on a pre-shared password for mutual authentication and key exchange, standardised in RFC 2945 [139]. It allows mutual authentication and secure key exchange, while being resistant to on-line brute-force and Man-in-the-Middle attacks (MITM). In addition, it can be used in conjunction with TLS [25] to create a secure transport layer without the need of certificates or new protocols [44, 52]. IT can be implemented independently from both the network operator can be used to support multiple users per application instance.

The user registration procedure they use to establish a shared secret for a newly installed application is also very similar to the one used by SRP. The difference is that while they designed their own protocol, with all the risks that entails, SRP is a recognized and reviewed standard which gives much better security guarantees. One novelty they introduce, however, is that the secret material negotiated during this registration process is used to create keys that can guarantee strong forward secrecy to the data stored locally on the device. We will discuss this in detail in Section 5.6.

9.1.1.3 Secure Data Upload and Session Handling

Our earlier work designed for Java ME based MDC systems and the SecourHealth framework share also a common way of uploading encrypted data directly to the server, without establishing an extra layer of security with TLS. However, we differ in how sessions are handled. Our approach used a public key encrypted request to establish a session, and obtain a HTTP session-Id from the server. The server maps the HTTP session object to a user session key sent by the client and sends an HTTP session-id to the client. The client uses this session-id for further interaction with the server. However, unlike HTTPS, we encrypt the traffic at the application layer, and the session-id will have to travel in clear. Still, hijacking the session would require the attacker to know both the encryption-key and the authentication-key currently used by both client and server, and replay attacks can be prevented by adding a unique sequential number to each request. Furthermore, in order to minimize the need for establishing new sessions, we renegotiate the session key periodically, without changing the session-Id. This should be done as a separate operation, since if either the server does not receive the request or the client the response, the two parties might end up using different session-keys.
SecourHealth session handling depends on the default TCP protocol. If some data is lost during transmission, the TCP may handle it transparently. Despite the fact that TCP has delivery guarantee, we still think that a solution at application layer is needed to handle the case of double or partial submissions at least on the server side, as TCP can tell us that something went wrong, but not exactly how many and which forms were

delivered and which not (well, according to how we organize the upload data structure we can make some guesses).

9.1.1.4 Perfect Forward Secrecy

The SecourHealth framework provides perfect forward secrecy using symmetric encryption as novel feature. As we briefly discussed in Section 5.6, the idea behind perfect forward secrecy is that a compromised encryption key does not compromise data encrypted in the past. In other word, it is virtually impossible to reveal keys used to encrypt previous data based on a currently compromised key.

Their approach works as follows. When the user registers on the device for the first time, a hash function is used to derive the first form encryption key. The hash function takes a master key derived from the user password and a random seed sent from the server. After the form encryption is derived, the application deletes the seed from the memory, so that only the server keeps the seed associated with the user. The first form data is encrypted using the derived encryption key. Afterwards, the form encryption serves to derive the next form key by passing through a hash function. Each subsequent forms are encrypted using a form key derived from the previous key. When the user logs out, the next key is derived and stored in persistent storage ready to be used when the user logs in once again. This means that the current key found on the device has not been used to encrypt anything yet, and previous keys cannot be derived directly from this one. The only way to do that would be to generate the whole sequence of keys starting from the seed, but that is only available on the server.

Next, we discuss the pros and cons of this approach:

Pros:

- Performance gain since every encryption/decryption is performed using symmetric algorithm both on client and server
- Even the device is physically lost, the attacker cannot reveal encrypted data. Furthermore, a compromised user's password does not help the attacker to access the encrypted data unless (s)he intercept the seed sent from the server during user registration.

Cons:

- Once the form is encrypted, there is no way to decrypt it and make a change or update. Many of the use cases we reviewed and worked with do not like the idea of making this approach. Instead, they need some flexibility of making changes on the form. For instance, when the field of a household form, members of the the house may not be present at the same time. So, the data collector stores partial information on the device and wait until all questions on the form answered.
- The server must persist and track of each seed associated with the user, requiring additional book keeping on the server. This may also require changes on server code base and database scheme, since the addition of new field introduce changes in the server database table which is already deployed.

- Since the solution requires storing encryption key on the server database, it may not fit for cloud based MDC systems. We discuss why in the next paragraphs.

It is worth mentioning that the team behind SecourHealth framework created the solution based on the assumption that their server is running on privately own server and the organization have a complete control over the physical network, application or any other level access. However, cloud computing is changing the demography of application deployment. When it comes to the first drawback listed above, we should also mention that SecourHealth does offer alternative solutions to allow editing, but of course at the cost of security since if the user must be able to decrypt the forms locally, an attacker may also be able to do so.

Although the solution for strong forward secrecy presented in [76] is very elegant and definitely something that also our framework should include, it is not a completely new concept. We also proposed a similar approach in our first paper [74] to distinguish between partially filled forms that could be edited locally at a later moment, and completed forms that could be encrypted so that only the server could read them, hence guaranteeing also a form for forward secrecy. The difference is that we used asymmetric cryptography to achieve this, since we were concerned also with secure provisioning, while in [76] this is done by leveraging only symmetric cryptography, in a very elegant and lightweight way. Currently, ODK offers also perfect forward secrecy by downloading a public key attached to the form definition, and then upload each single form to the cloud encrypted with a random symmetric key, which is in turn encrypted with the aforementioned public key. This assure very strong data security both at rest on the client and on cloud based server since the private key is stored on a third party machine and neither on the device nor the cloud server. The downside is the overhead generated by all the encryption required for each form and the little flexibility the collector has, since forms cannot be edited after being encrypted. Besides only the owner of the public key can decrypt and possibly redistribute the collected data. It is clear that the safest solution when it comes to confidentiality is to encrypt the data locally before upload with a key which is not shared with the cloud storage provider.

satisfying all requirements including the perfect forward secrecy, secure cloud storage, data recovery, and data sharing among many actors is not a trivial task. At the end, we made some compromise on the perfect forward secrecy and came up with a decent solution that satisfies all the above requirements. For further details on the solution, we refer the readers to our publications [44, 45]. A downside of this approach is that the heavy cryptography limits the possibilities to collaborate on-line and the number of possible data viewers, but it might be an appropriate solution for small research projects that place a premium on privacy. When we say "heavy cryptography", we are referring to the computation needed for single search query in entirely encrypted cloud storage. The public/private key operation is performed only once on the device and does not have any performance impact on the device. In fact, even if symmetric encryption outperform the heavy public/private key encryption, it had an acceptable performance even on old Java ME devices. Therefore, this is not an issue on advanced device like Android.

9.2 Conclusions

In this dissertation, the candidate have presented an assessment of the security of systems for remote data collection on Java enabled low-end mobile phones and Android enabled smartphones, and have identified the specific challenges for their deployment and how these affect their security needs. Based on these observations we have outlined and discussed some possible solutions that are particularly appropriate to secure these systems. Our primary aim in designing these solutions has been that it should be easy to integrate with existing systems, without requiring substantial retrofitting. The secure authentication, storage, and communication schemes are quite modular, they are independent and single requirements might be dropped to achieve solutions better tailored for a specific client, while our analysis can be used as starting point to assess the security risks involved, and weigh them against the benefits.

The SecureMDC framework confirmed also that the schemes we proposed are feasible in practice and quite flexible, and they can be tailored for clients with specific additional constraints by possibly dropping some of the requirements we identified. For instance, if one already uses HTTPS and it works well for the purpose, then only the storage or the authentication scheme can be adopted. If a risk analysis, instead, shows that it is an acceptable risk to use the same password both for the server and the mobile phone, or we can assume that each mobile phone is personal, then it is possible to adapt the protocol to use only one password. The candidate does not claim that these are the best or the only possible solutions, but the candidate think that they are general and flexible enough to derive implementations that can be adapted to MDC Systems with a wide range of requirements.

We have not mentioned any particular cryptographic algorithms to use either, leaving each implementation to choose what is most appropriate. For public key cryptography one could use either RSA or EC (Elliptic Curve), but the point is that it is much easier to generate a pair of public and private key pairs and use them, rather than having to go through the expensive and complicated process of obtaining a signed certificate.

In our implementation we used RSA with a 2048 bits key, and for symmetric cryptography we chose AES in CBC mode with a unique initialization vector per encryption, associated with a HMAC based on the SHA-1 hash scheme. We also offer an alternative that could use an authentication mode like CCM [41] or GCM [78], which takes care of both encryption and authentication at the same time, thus not having to generate a separate HMAC.

Secure provisioning and cloud storage remain the most challenging parts, and the candidate still does not have a working solution in these areas. The idea the candidate proposed here for the secure cloud storage is probably good enough for small projects, but would probably prevent good usability in large scale projects.

The solution we implemented was based on both a custom and standard protocol developed by considering the specific constraints of MDC Systems, but it makes almost no assumptions about how or where data is stored, or how the communication layer of an existing application is implemented. This guarantees wide compatibility. Besides, the different secure solutions that it offers are very modular, and can be used independently to fit the needs of MDCS Systems with different security requirements. Most of the secure framework interfaces that are exposed to the programmer are very similar to the Java ME and Android counterparts in terms of methods and behavior. This means that any programmer who knows how to use the normal Java ME or Android APIs will also know how to use the secure ones.

We analyzed the problem of secure cloud storage in the context of remote mobile data collection and proposed an approach that could provide a relatively straightforward solution to data protection, sharing and recovery. Standard symmetric and asymmetric cryptographic techniques are employed, which makes it feasible to implement in practice and compatible with most existing systems and technologies. At the same time, we tried to minimize heavy resource consumption on the mobile devices and additional storage use on the server. Finally, leveraging on the hierarchical nature of a project organization we can enable data recovery also in cases wherein a user-centric model data would be lost because of the lack of a secure root of trust.

9.2.1 Contributions

In this dissertation, the candidate has looked at security challenges in resources-constrained, low-budget mobile data collection systems. We systematically identified and addressed most of the functional and security requirements in MDC systems. In collaboration with partners project, the candidate designed a secure framework that can be integrated in existing systems and provides:

- an Authenticator module that handles user authentication on a mobile device and a remotely located server. The Authententicator module is presented in the chapter 4.
- a Secure Storage module that provides data protection while data is at rest on the mobile device. The Secure Storage module is described in the chapter 5
- a Secure Communication module that handles establishing a secure tunnel between client and server without relying on third-party certificates. The Secure Communication module is presented in the chapter 6
- secure cloud storage solution particularly tailored for mobile data collection systems in healthcare domain. The secure communication storage is discussed in the chapter 6
- account and data recovery mechanism as it is presented in the chapters 6, 4, and 6.

Furthermore, the candidate together with the MDCS security team at the University of Bergen promoted the research in order to gain data protection awareness within system developers, system users, organizations, and academia.

A
List of Publications

These list of publications are the result of candidate's work during the doctoral program period.

1. **Biblography**: Gejibo, Samson and Mancini, Federico and Mughal, Khalid, *Mobile Data Collection: a Security Perspective*, A Book Chapter in Mobile Health (mHealth): A Technology Road Map, pp. 1015-1042, Springer Series in Bio-/Neuroinformatics, Springer International Publishing, March 2015.

   Synopsis: This is a book chapter that focuses on the latest technologies and the security challenges faced by MDCS specifically when Android-based smart phones and cloud based servers are employed. The chapter also discusses the solutions designed for older Java ME based devices.

2. **Bibliography**: S. Gejibo, D. Grasso, F. Mancini, and K. A. Mughal, *Secure cloud storage for remote mobile data collection*, In Proceedings of the Second Nordic Symposium on Cloud Computing and Internet Technologies, NordiCloud '13, pages 8–14. ACM, 2013.

   Synopsis: This paper analyzed the problem of secure cloud storage in the context of remote mobile data collection and proposed an approach that could provide a relatively straightforward solution to data protection, sharing and recovery.

3. **Bibliography**: S. H. Gejibo, F. Mancini, K. A. Mughal, R. A. B. Valvik, J. I. Klungsyr, *Secure Data Storage for Mobile Data Collection Systems*, Proceedings of International ACM Conference on Management of Emergent Digital EcoSystems (MEDES '12), October, 2012, Addis Ababa, Ethiopia, p. 131–144, ACM.

   Synopsis: This paper focuses on low budget mobile phones with low hardware and software specification, and proposes adequate secure solutions for data storage protection. The solution has been extensively tested and integrated into a production MDCS.

4. **Bibliography**: S. H. Gejibo, F. Mancini, K. A. Mughal, R. A. B. Valvik, J. I. Klungsøyr, *Secure Data Storage for Java ME-Based Mobile Data Collection Systems*, Proceedings of 14th IEEE International Conference on e-Health Networking Applications and Services, Healthcom2012, October 2012, Beijing, China.

   Synopsis: This paper builds on our previous work and present a solution to storage security issues for data collection systems running on the Java ME platform.

5. **Bibliography**: F. Mancini, S. H. Gejibo, K. A. Mughal, R. A. B. Valvik, and J. I. Klungsøyr, *Secure Mobile Data Collection Systems for Low-Budget Settings*, Proceedings of the 7th International Conference on Availability, Reliability and Security (ARES 2012), August 2012 , Prague, Czech Republic, p. 196-205.

   Synopsis: This paper analyzed the challenges in securing Mobile Data Collection Systems deployed in remote areas and in low-budget settings, and discuss how one can provide an adequate security solution for such systems.

6. **Bibliography**: S. H. Gejibo, F. Mancini, K. A. Mughal, J. I. Klungsøyr, R. A. B. Valvik, *Challenges in Implementing End-to-End Secure Protocol for Java ME-Based Mobile Data Collection in Low-Budget Settings*, Proceedings of International Symposium on Engineering Secure Software and Systems, February, 2012, Eindhoven, The Netherlands, Springer, p. 38-45.

   Synopsis: In this paper we analyze implementation challenges of a proposed security protocol based on the Java ME platform. The protocol presents a flexible secure solution that encapsulates data for storage and transmission without requiring significant changes in the existing mobile client application.

7. **Bibliography**: Samson H Gejibo , Mzomuhle T. Nkosi, Fisseha Mekuria, *Challenges in Mobile Bio-Sensor Based mHealth Development*, 2011 IEEE 13th International Conference on e-Health Networking, Applications and Services, June 2011, Columbia, MO USA, p. 21-27.

   Synopsis: The paper addresses bio-sensor signal processing and secure communication of sensor signals based on next generation mobile technology and bio sensors, with the aim to facilitate the development of secure and innovative mobile health services.

8. **Bibliography**: F. Mancini, S. H. Gejibo, K. A. Mughal, J. I. Klungsøyr, *Adding Security to Mobile Data Collection*, Proceedings of 13th IEEE International Conference on e-Health Networking Applications and Services, Healthcom2011, June 2011 , Columbia, MO, USA, p. 86-89.

Synopsis: In this paper we propose a protocol that provides end-to-end security, encrypted data storage and recovery mechanisms on mobile devices. We use openXdata as our reference mHealth system.

Bibliography

[1] ADIBI, S., Ed. *Mobile Health (mHealth): A Technology Road Map*. Springer Series in Bio-/Neuroinformatics. Springer International Publishing, March 2015. 7

[2] AMAZON ELASTIC COMPUTE CLOUD (AMAZON EC2). http://aws.amazon.com/ec2/. Online, Accessed March 2015. 6.1

[3] ANDRIOTIS, P., TRYFONAS, T., OIKONOMOU, G., AND YILDIZ, C. A pilot study on the security of pattern screen-lock methods and soft side channel attacks. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks* (New York, NY, USA, 2013), WiSec '13, ACM, pp. 1–6. 4.4.1.2

[4] ANOKWA, Y., RIBEKA, N., PARIKH, T., BORRIELLO, G., AND WERE, M. C. Design of a phone-based clinical decision support system for resource-limited settings. In *Proceedings of the Fifth International Conference on Information and Communication Technologies and Development* (2012), ICTD '12, ACM, pp. 13–24. 1.5

[5] ANOOJ, P. Implementing decision tree fuzzy rules in clinical decision support system after comparing with fuzzy based and neural network based systems. In *IT Convergence and Security (ICITCS), 2013 International Conference on* (Dec 2013), pp. 1–6. 1.5

[6] APPLE INC. ios security as of may 2015. Tech. rep., Cupertino, CA, United States, April 2015. Last Accessed: May 2015. 7.1.2

[7] ARM LIMITED. ARM Security Technology: Building a secure system using Trustzone Technology. Technical report, Internet Engineering Task Force, 2009. 5.4.2.2, 8.2.3

[8] AVANCHA, S., BAXI, A., AND KOTZ, D. Privacy in mobile technology for personal healthcare. *ACM Computing Surveys 45*, 1 (Dec. 2012), 3:1–3:54. 1.4

[9] AVIV, A. J., GIBSON, K., MOSSOP, E., BLAZE, M., AND SMITH, J. M. Smudge attacks on smartphone touch screens. In *Proceedings of the 4th USENIX Conference on Offensive Technologies* (Berkeley, CA, USA, 2010), WOOT'10, USENIX Association, pp. 1–7. 4.4.1.2

[10] BARRERA, D., CLARK, J., MCCARNEY, D., AND VAN OORSCHOT, P. Understanding and improving app installation security mechanisms through empirical

analysis of android. In *2nd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)* (2012), pp. 81–92. 3.3.2, 7.1.2

[11] BECHER, M., FREILING, F., HOFFMANN, J., HOLZ, T., UELLENBECK, S., AND WOLF, C. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Security and Privacy (SP), 2011 IEEE Symposium on* (May 2011), pp. 96–111. 1.4.1, 1.4.1

[12] BELLARE, M., AND MINER, S. A forward-secure digital signature scheme. In *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999, pp. 431–448. 5.6

[13] BLANZ, V., AND VETTER, T. Face recognition based on fitting a 3d morphable model. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 25*, 9 (Sept 2003), 1063–1074. 4.4.1.3

[14] BORGMANN, M., HAHN, T., HERFERT, M., KUNZ, T., RICHTER, M., VIEBEG, U., AND VOWE, S. On the security of cloud storage services. Tech. rep., Fraunhofer Institute for Secure Information Technology SIT, Darmstadt, Germany, 2012. 2.5, 6.5.1

[15] BRUNETTE, W., SODT, R., CHAUDHRI, R., GOEL, M., FALCONE, M., VAN ORDEN, J., AND BORRIELLO, G. Open data kit sensors: A sensor integration framework for android at the application-level. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2012), MobiSys '12, ACM, pp. 351–364. 1.2, 2.2.1

[16] BRUNETTE, W., SUNDT, M., DELL, N., CHAUDHRI, R., BREIT, N., AND BORRIELLO, G. Open data kit 2.0: expanding and refining information services for developing regions. In *HotMobile - 14th Workshop on Mobile Computing Systems and Applications* (2013), p. 10. 2.2.1, 3.2, 6.5.1

[17] CELL-LIFE. Emit: mhealth solutions. `http://www.emitmobile.co.za/`. Online, Accessed March 2011. 5.2

[18] CHADWICK, P. Regulations and standards for wireless applications in ehealth. In *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE* (Aug 2007), pp. 6170–6173. 1.1

[19] CHAKRABARTI, S., AND SINGHAL, M. Password-based authentication: Preventing dictionary attacks. *Computer 40*, 6 (June 2007), 68–74. 4.4.2.2

[20] CHEN, S., PANDE, A., AND MOHAPATRA, P. Sensor-assisted facial recognition: An enhanced biometric authentication system for smartphones. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2014), MobiSys '14, ACM, pp. 109–122. 4.4.1.3

[21] CHRIS JUNG AND CARTONG. Mobile data collection systems: A review of the current state of the field. Tech. rep., NOMAD: HumanitariaN Operations Mobile Acquisition of Data, January 2012. 2.1

[22] COMMCAREHQ. Mobile data collection tools. http://www.commcarehq.org. Online, Accessed November 2011. 2.1.1.3, 5.2

[23] COOIJMANS, T., DE RUITER, J., AND POLL, E. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices* (New York, NY, USA, 2014), SPSM '14, ACM, pp. 11–20. 5.4.2, 5.4.2.2, 5.4.2.3, 5.4.2.3, 8.2.3

[24] CROCKER, S., AND SCHILLER, J. RFC 4086 - randomness requirements for security. http://www.ietf.org/rfc/rfc4086.txt, 2005. 8.1.2.1

[25] D. TAYLOR, T. WU, N. M., AND PERRIN, T. RFC 5054: Using the secure remote password (srp) protocol for tls authentication. http://www.ietf.org/rfc/rfc5054, 2007. 3, 4.4.2.2, 4.4.2.2, 8.1.3, 9.1.1.2

[26] DATADYNE. Mobile data collection system (mdcs). http://www.datadyne.org. Online, Accessed November 2011. 5.2

[27] DAVID SCHUETZ. The iOS MDM Protocol. Tech. rep., BlackHat, Intrepidus Group, Inc., 2011. 5.4.4

[28] DEBBABI, M., SALEH, M., TALHI, C., AND ZHIOUA, S. Security evaluation of j2me cldc embedded java platform. *Journal of Object Technology 5* (2006). 2.4.1

[29] DELL, N., CRAWFORD, J., BREIT, N., CHALUCO, T., COELHO, A., MCCORD, J., AND BORRIELLO, G. Integrating odk scan into the community health worker supply chain in mozambique. In *Proceedings of the Sixth International Conference on Information and Communication Technologies and Development: Full Papers - Volume 1* (New York, NY, USA, 2013), ICTD '13, ACM, pp. 228–237. 2.2.1

[30] DHIS 2. The District Health Information Software (DHIS). https://www.dhis2.org/. Online, Accessed March 2015. 2.1.1.3

[31] DIFFIE, W., VAN OORSCHOT, P. C., AND WIENER, M. J. Authentication and authenticated key exchanges. *Des. Codes Cryptography 2*, 2 (June 1992), 107–125. 5.6

[32] EGEBERG, T. Storage of sensitive data in a Java enabled cell phone. Master's thesis, Høgskolen i Gjøvik, 2006. 5.2, 5.4.1

[33] ELENKOV, N. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*, 1st ed. No Starch Press, San Francisco, CA, USA, 2014. 4.4.1.1, 4.4.1.2, 4.4.1.3, 5.4.3, 5.4.3, 5.7.2, 8.1.1.2, 8.2.3

[34] ENG, T. The e-health landscape – a terrain map of emerging information and communication technologies in health and health care. princeton nj:. *The Robert Wood Johnson Foundation* (2001). 1.1

[35] ENGLER, J., KARLOF, C., SHI, E., AND SONG, D. Is it too late for PAKE? In *Web 2.0 Security and Privacy Workshop* (2009), W2SP 2009. 4.4.2.2

[36] ERIKA MCCALLISTER, TIM GRANCE, K. S. Guide to protecting the confidentiality of personally identifiable information (PII), 2010. NIST Special Publication 800-122. 1.5

[37] EUROPEAN COMMISSION. GREEN PAPER on mobile Health ("mHealth") - SWD(2014) 135 final. Tech. rep., Brussels, EU, April 2014. 1.2

[38] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE (ETSI. General packet radio service (gprs) standard. `http://www.etsi.org`. 5.8.2

[39] EYSENBACH, G. What is e-health? *J Med Internet Res 3*, 2 (Jun 2001), e20. 1.1

[40] FEDERAL INFORMATION PROCESSING STANDARDS (FIBS) PUBLICATION 199. Standards for Security Categorization of Federal Information and Information Systems. Tech. rep., National Institute of Standards and Technology (NIST), February 2004. 1.4

[41] FERGUSON, N., HOUSLEY, R., AND WHITING, D. RFC 3610 - Counter with CBC-MAC (CCM), 2003. 9.2

[42] FRANKS, J., HALLAM-BAKER, P., HOSTETLER, J., LAWRENCE, S., LEACH, P., LUOTONEN, A., AND STEWART, L. HTTP authentication: Basic and digest access authentication. RFC 2617, Internet Engineering Task Force, June 1999. 2.2.1

[43] FRANKS, J., HALLAM-BAKER, P., HOSTETLER, J., LAWRENCE, S., LEACH, P., LUOTONEN, A., AND STEWART, L. RFC 2617 - HTTP authentication: Basic and digest access authentication, 1999. 4.4.2, 4.4.2.1, 4.4.2.1

[44] GEJIBO, S., GRASSO, D., MANCINI, F., AND MUGHAL, K. A. Secure cloud storage for remote mobile data collection. In *Proceedings of the Second Nordic Symposium on Cloud Computing and Internet Technologies* (2013), NordiCloud '13, ACM, pp. 8–14. 3, 4.5.3, 6, 6.5.2, 9.1.1.2, 9.1.1.4

[45] GEJIBO, S., MANCINI, F., AND MUGHAL, K. *Mobile Data Collection: a Security Perspective*. Springer Series in Bio-/Neuroinformatics. Springer International Publishing, March 2015, pp. 1015–1042. 9.1.1.4

[46] GEJIBO, S., MANCINI, F., MUGHAL, K. A., VALVIK, R., AND KLUNGSØYR, J. Secure data storage for mobile data collection systems. In *MEDES'12 - International Conference on Management of Emergent Digital EcoSystems* (2012), J. Kacprzyk, D. Laurent, and R. Chbeir, Eds., ACM, pp. 131–144. 5, 5.7.1, 6.6.4, 6.6.5.1, 6.7.4

[47] GEJIBO, S., NKOSI, M. T., AND MEKURIA, F. Challenges in mobile bio-sensor based mhealth development. In *Healthcom 2011 - 13th IEEE International Conference on e-Health Networking Applications and Services* (june 2011), pp. 21 –27. 1.2, 4.4.1.4

[48] GLOBAL PLATFORM. Trusted User Interface API Specification v1.0. `https://www.globalplatform.org/specificationsdevice.asp`, 2013. 5.4.2.2, 8.2.3

[49] GOODIN, D. Hacked blizzard passwords are not hard to crack. `http://arstechnica.com/security/2012/08/hacked-blizzard-passwords-not-hard-to-crack/`. [Last Accessed : January 2015]. 4.4.2.2

[50] GOOGLE ANDROID. Android Payment System. `https://www.android.com/pay/`. Online, Accessed June 2015. 8.2.3

[51] GOOGLE APP ENGINE. https://developers.google.com/appengine/. Online, Accessed March 2013. 6.1

[52] GRASSO, D. Authentication and secure data storage in cloud based mobile data collection. Master's thesis, UNIVERSITA DEGLI STUDI ROMA TRE, 2014. 1.6.2, 2.1, 2.1.2, 2.1.3, 2.3, 4.3, 4.4.2.2, 4.4.2.2, 9.1.1.2

[53] GROLIMUND, D., MEISSER, L., SCHMID, S., AND WATTENHOFER, R. Cryptree: A folder tree structure for cryptographic file systems. In *Reliable Distributed Systems, 2006. SRDS '06. 25th IEEE Symposium on* (2006), pp. 189–198. 6.8

[54] GÜNTHER, C. An identity-based key-exchange protocol. In *Advances in Cryptology — EUROCRYPT '89*, J.-J. Quisquater and J. Vandewalle, Eds., vol. 434 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1990, pp. 29–37. 5.6

[55] HARTUNG, C., LERER, A., ANOKWA, Y., TSENG, C., BRUNETTE, W., AND BORRIELLO, G. Open data kit: Tools to build information services for developing regions. In *Proceedings of the 4th ACM/IEEE International Conference on Information and Communication Technologies and Development* (New York, NY, USA, 2010), ICTD '10, ACM, pp. 18:1–18:12. 2.2.1

[56] HARVEST YOUR DATA. Mobile data collection system (mdcs). 2

[57] HEALTH INSURANCE PORTABILITY AND ACCOUNTABILITY ACT (HIPAA), 1996. Pub. L. No. 104-191, as implemented by 45 C.F.R. § 160, 162, 164. 1.4

[58] INTERNET SOCIETY. RFC 2818 - http over tls. `http://www.ietf.org/rfc/rfc2818.txt`, 2000. 4.4.2.1, 6.5.2

[59] ITANI, W., AND KAYSSI, A. J2ME application-layer end-to-end security for m-commerce. *Journal of Network and Computer Applications 27*, 1 (January 2004), 13–32. 5.4.1, 6.5.2

[60] JEFF FORRISTAL. Android: One Root to Own Them All. Tech. rep., BlackHat, United States, 2013. 7.1.2

[61] JOINT UNITED NATIONS PROGRAMME ON HIV/AIDS. Guidelines on protecting the confidentiality and security of hiv information. Tech. rep., UNAIDS, May 2007. 1.5

[62] JONES, M., BRADLEY, J., AND SAKIMURA, N. RFC 7519: Json web token (jwt). `http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html`, 2015. 4.5

[63] KLUNGSØYR, J. Handheld computers for data collection in field research in Uganda. Development of EpiHandy and field tests. Master's thesis, Centre for International Health, University of Bergen, 2004. 1.6.1

[64] KLUNGSØYR, J., TYLLESKAR, T., MACLEOD, B., BAGYENDA, P., CHEN, W., AND WAKHOLI, P. OMEVAC - open mobile electronic vaccine trials, an interdisciplinary project to improve quality of vaccine trials in low resource settings. In *Proceedings of M4D '08 - The 1st International Conference on Mobile Communication Technology for Development* (2008), Karlstad University Studies, pp. 36–44. 1.6.1

[65] KLUNGSØYR, J. I., SKJØRESTAD, S., CHEN, W., AND TYLLESKAR, T. Using mobile technology for data collection. In *Proceedings of NOKOBIT '05- Norsk Konferanse for Organisasjoners Bruk av IT, University of Bergen, Bergen.* (2005), Tapir, pp. 181–194. 1.6.1

[66] KOMANDURI, S., SHAY, R., KELLEY, P. G., MAZUREK, M. L., BAUER, L., CHRISTIN, N., CRANOR, L. F., AND EGELMAN, S. Of passwords and people: Measuring the effect of password-composition policies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2011), CHI '11, ACM, pp. 2595–2604. 4.4.1.1

[67] KWON, T., PARK, Y.-H., AND LEE, H. J. Security analysis and improvement of the efficient password-based authentication protocol. *Communications Letters, IEEE 9*, 1 (Jan 2005), 93–95. 4.4.2.2

[68] LANE, N., MILUZZO, E., LU, H., PEEBLES, D., CHOUDHURY, T., AND CAMPBELL, A. A survey of mobile phone sensing. *IEEE Communication Megazine 48*, 9 (2010), 140 – 150. 1.2

[69] LANE, N. D., MILUZZO, E., LU, H., PEEBLES, D., CHOUDHURY, T., AND CAMPBELL, A. T. A survey of mobile phone sensing. *Comm. Mag. 48*, 9 (Sept. 2010), 140–150. 4.4.1.4

[70] LEGION OF THE BOUNCY CASTLE, T. `http://www.bouncycastle.org/`. Online, Accessed March 2011. 8.1.1.1

[71] LIU, F., TONG, J., MAO, J., BOHN, R., MESSINA, J., BADGER, L., AND LEAF, D. *NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology (Special Publication 500-292).* CreateSpace Independent Publishing Platform, USA, 2012. 4

[72] MAGPI. Mobile Data Collection System (MDCS). `https://www.magpi.com/`. Online, Accessed September 2013. 2.1.1.3, 5.2

[73] MANCINI, F., GEJIBO, S., MUGHAL, K. A., VALVIK, R., AND KLUNGSØYR, J. Secure mobile data collection systems for low-budget settings. In *ARES 2011 - The Seventh International Conference on Availability, Reliability* (2012), IEEE Computer Society, pp. 196–205. 3, 4, 4.5.3, 6.1, 6.3, 6.5.2, 6.6, 6.6.3, 7

[74] MANCINI, F., MUGHAL, K., GEJIBO, S., AND KLUNGSØYR, J. Adding security to mobile data collection. In *Healthcom 2011 - 13th IEEE International Conference on e-Health Networking Applications and Services* (june 2011), pp. 86 –89. 4.5.3, 6.3, 9.1.1, 9.1.1.4

[75] MANGANARO, A., KOBLENSKY, M., AND LORETI, M. Design of a password-based authentication method for wireless networks. In *WINSYS 2007 - Proceedings of the International Conference on Wireless Information Networks and Systems, Barcelona, Spain, July 28-31, 2007, WINSYS is part of ICETE - The International Joint Conference on e-Business and Telecommunications* (2007), pp. 9–16. 4.4.2.2

[76] MARCOS A. SIMPLÍCIO JR. AND LEONARDO H. IWAYA AND BRUNO M. BARROS AND TEREZA CRISTINA M. B. CARVALHO AND MATS NÄSLUND. Secourhealth: A delay-tolerant security framework for mobile health data collection. *IEEE J. Biomedical and Health Informatics 19*, 2 (2015), 761–772. 1.2, 3.1.6, 4.2.1, 5.5.1, 5.6, 9.1.1, 9.1.1.4

[77] MAYRON, L. M., HAUSAWI, Y., AND BAHR, G. S. Secure, usable biometric authentication systems. In *Proceedings of the 7th International Conference on Universal Access in Human-Computer Interaction: Design Methods, Tools, and Interaction Techniques for eInclusion - Volume Part I* (Berlin, Heidelberg, 2013), UAHCI'13, Springer-Verlag, pp. 195–204. 4.4.1.4

[78] MCGREW, D. A., AND VIEGA, J. The galois/counter mode of operation (gcm). *NIST Modes Operation Symmetric Key Block Ciphers* (2005). 9.2

[79] MECHAEL, P., BATAVIA, H., KAONGA, N., SEARLE, S., KWAN, A., GOLDBERGER, A., FU, L., AND OSSMAN, J. Barriers and gaps affecting mhealth in low and middle income countries:policy white paper. Tech. rep., Center for Global Health and Economic Development Earth Institute, Columbia University, May 2010. 4.6

[80] MEGA. Secure Cloud Storage Provider. `https://mega.nz`. Online, Accessed March 2013. 6.5.1

[81] MEKURIA, F., NKOSI, M., SEOTLO, V., AND TWALA, B. Intelligent mobile sensing analysis systems. In *Proceedings of 3rd CSIR Biennial conference* (2010). 1.1, 1.2

[82] MICROFOCT. Microsoft Data Gathering: a Mobile Data Collection System (MDCS). `https://github.com/nokiadatagathering/ndg-mobile-client`. Online, Accessed September 2011. 5.2

[83] MICROSOFT. Microsoft security intelligence report. Tech. rep., June 2014. 4.4.2

[84] MIRKOVIC, J., BRYHNI, H., AND RULAND, C. Secure solution for mobile access to patient's health care record. In *e-Health Networking Applications and Services (Healthcom), 2011 13th IEEE International Conference on* (2011), pp. 296–303. 9.1

[85] MOBENZI RESEARCHER. Mobile data collection systems (mdcs). http://www.mobenzi.com/researcher/. Online, Accessed September 2013. 5.2

[86] MODI RESEARCH GROUP. Mobile Data Collection System (MDCS). `http://www.formhub.org/`. Online, Accessed September 2014. 2.1.1.3

[87] MOXIE MARLINSPIKE. Your app shouldn't suffer ssl's problems. `http://www.thoughtcrime.org/blog/authenticity-is-broken-in-ssl-but-your-app-ha/`. Online, Accessed November 2014. 8.2.1

[88] NCRYPTED. Secure Cloud Storage Provider. https://www.ncryptedcloud.com/. Online, Accessed March 2013. 6.5.1

[89] NIELSEN, P. M. The secure remote password protocol isn't bad. `https://patrickmn.com/security/secure-remote-password-isnt-bad/`. [Last Accessed : November 2014]. 4.4.2.2

[90] NOKIA 2330C-2 CLASSIC. Device specification. `http://www.forum.nokia.com/Devices/Device_specifications`. Online, Accessed Mars 2011. 5.8.2

[91] OPEN CLINICA. Clinical Trial Software for Electronic Data Capature. `http://www.openclinica.org/`. Online, Accessed March 2011. 2.1.1.3

[92] OPEN DATA KIT. Encrypted Forms. `https://opendatakit.org/help/encrypted-forms/`. Online, Accessed March 2015. 5.2

[93] OPEN DATA KIT (ODK). Help For Hire (ODK Business Model). http://opendatakit.org/help/help-for-hire/. Online, Accessed March 2015. 6.5.1

[94] OPEN WEB APPLICATION SECURITY PROJECT (OWASP). OWASP Mobile Security Project - Top Ten Mobile Risks. `https://www.owasp.org/index.php/OWASP_Mobile_Security_Project`. Online, Accessed May 2015. 5.1

[95] OPENMRS. OpenMRS Medical Record System. http://openmrs.org/. Online, Accessed March 2015. 2.1.1.3

[96] OPENROSA CONSORTIUM. OpenRosa Authentication API. https://bitbucket.org/javarosa/javarosa/wiki/AuthenticationAPI. Online, Accessed May 2015. 4.4.2.1

[97] OPENXDATA. Mobile data collection systems (mdcs). http://www.openxdata.org. Online, Accessed March 2013. 2.1.1.3, 5.2

[98] ORACLE. Security and trust services API for J2ME (SATSA). `http://java.sun.com/products/satsa/`, 2006. Online, Accessed March 2011. 5.4.1

[99] OWASP. Certificate and Public Key Pinning. `//www.owasp.org/index.php/Certificate_and_Public_Key_Pinning`. Online, Accessed November 2014. 4.5.3

[100] PAN, G., SUN, L., WU, Z., AND LAO, S. Eyeblink-based anti-spoofing in face recognition from a generic webcamera. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on* (Oct 2007), pp. 1–8. 4.4.1.3

[101] PERCIVAL, C., AND JOSEFSSON, S. The scrypt Password-Based Key Derivation Function. `https://tools.ietf.org/html/draft-josefsson-scrypt-kdf-03`, 2015. 5.4.3

[102] PETER KHISA WAKHOLI. *Process Aware Mobile Systems: Applied to mobile-phone based data collection*. PhD thesis, University of Bergen. 1.6.1, 5.5.1

[103] PIGADAS, V., DOUKAS, C., PLAGIANAKOS, V. P., AND MAGLOGIANNIS, I. Enabling constant monitoring of chronic patient using android smart phones. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments* (New York, NY, USA, 2011), PETRA '11, ACM, pp. 63:1–63:2. 1.2, 4.4.1.4

[104] POTHARAJO, R., NEWELL, A., NITA-ROTARU, C., AND ZHANG, X. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *ES-SoS'12 Proceedings of the 4th international conference on Engineering Secure Software and Systems* (2012), Springer-Verlag Berlin, Heidelberg, pp. 106–120. 7.1.2

[105] QUEIROLO, C., SILVA, L., BELLON, O., AND PAMPLONA SEGUNDO, M. 3d face recognition using simulated annealing and the surface interpenetration measure. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 32*, 2 (Feb 2010), 206–219. 4.4.1.3

[106] RAHBAR, A. An e-ambulatory healthcare system using mobile network. In *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations* (Washington, DC, USA, 2010), ITNG '10, IEEE Computer Society, pp. 1269–1273. 1.2, 4.4.1.4

[107] RAJPUT ZA, MBUGUA S, A. D. E. A. Evaluation of an android-based mhealth system for population surveillance in developing countries. *Journal of the American Medical Informatics Association : JAMIA 19*, 4 (Dec. 2012), 655–659. 1.5

[108] RESCORLA, E. Http over tls (rfc 2818), 2000. 3

[109] RESCORLA, E. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Professional, 2000. 2.5

[110] RESCORLA, E., AND SCHIFFMAN, A. RFC 2660 - The Secure HyperText Transfer Protocol, 1999. 6.6.1.1

[111] RON GUTIERREZ AND STEPHEN KOMAL. Unwrapping the Truth: Analysis of Mobile Application Wrapping Solutions. `https://www.youtube.com/watch?v=TfjnkOI5B18`. Online, Accessed May 2015. 5.4.5

[112] Roosa, S. B., and Schultze, S. The "Certificate Authority" Trust Model for SSL: A Defective Foundation for Encrypted Web Traffic and a Legal Quagmire. *Intellectual Property & Technology Law Journal 22*, 11 (2010), 3–7. 3, 4.5.3, 6.5.2, 6.6.3

[113] Samsung. Samsung KNOX: Mobile Enterprise Security NSA Approval. `https://www.nsa.gov/ia/programs/csfc_program/component_list.shtml`. Online, Accessed May 2015. 5.4.4

[114] Samsung Electronics. Samsung KNOX: Mobile Enterprise Security. `https://www.samsungknox.com/`. Online, Accessed March 2015. 5.4.4

[115] Sapsford, R., and Jupp, V., Eds. *Data Collection and Analysis*, second edition ed. SAGE Publications Ltd, March 2006. 2.1

[116] Schaub, F., Deyhle, R., and Weber, M. Password entry usability and shoulder surfing susceptibility on different smartphone platforms. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia* (New York, NY, USA, 2012), MUM '12, ACM, pp. 13:1–13:10. 4.4.1.1

[117] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., and Glezer, C. Google android: A comprehensive security assessment. *Security Privacy, IEEE 8*, 2 (March 2010), 35–44. 2.4.2

[118] Shah, S. M. A., Gul, N., Ahmad, H. F., and Bahsoon, R. Secure storage and communication in J2ME based lightweight multi-agent systems. *Proceedings of KES-AMSTA'08 - the 2nd KES International conference on Agent and multi-agent systems: technologies and applications, Incheon, Korea*, 887–896. 5.4.1, 6.5.2

[119] Shahzad, M., Liu, A. X., and Samuel, A. Secure unlocking of mobile touch screen devices by simple gestures: You can see it but you can not do it. In *Proceedings of the 19th Annual International Conference on Mobile Computing &#38; Networking* (New York, NY, USA, 2013), MobiCom '13, ACM, pp. 39–50. 4.4.1.2

[120] Shamir, A. How to share a secret. *Commun. ACM 22*, 11 (1979), 612–613. 6.6.3

[121] Shao, D. A proposal of a mobile health data collection and reporting system for the developing world. Master's thesis, Malmö University, 2012. 4.2.1

[122] Simonsen, K. I. F., Moen, V., and Hole, K. J. Attack on sun's MIDP reference implementation of SSL. *Proceedings of NORDSEC 2005 - 10th Nordic Workshop on Secure IT systems, Tartu, Estonia.* (2005). 2.5, 6.5.2

[123] Singh, K., Zhong, J., Mirchandani, V., Batten, L. M., and Bertók, P. Securing data privacy on mobile devices in emergency health situations. In *MobiSec* (2012), A. U. Schmidt, G. Russello, I. Krontiris, and S. Lian, Eds., vol. 107 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer, pp. 119–130. 9.1

[124] SORBER, J., SHIN, M., PETERSON, R. A., AND KOTZ, D. Plug-n-trust: practical trusted sensing for mhealth. In *MobiSys* (2012), N. Davies, S. Seshan, and L. Zhong, Eds., ACM, pp. 309–322. 1.3, 9.1

[125] SPILMAN, J. Srp won't protect blizzards stolen passwords. `http://opine.me/blizzards-battle-net-hack/`. [Last Accessed : February 2015]. 4.4.2.2

[126] SUFATRIO, TAN, D. J. J., CHUA, T.-W., AND THING, V. L. L. Securing android: A survey, taxonomy, and challenges. *ACM Comput. Surv. 47*, 4 (May 2015), 58:1–58:45. 2.4.2

[127] THE 3RD GENERATION PARTNERSHIP PROJECT (3GPP). Generic authentication architecture (gaa) & generic bootstrapping architecture (gba). Online, Accessed March 2015. 9.1.1.2

[128] THE CA/BROWSER FORUM. Baseline requirements for the issuance and management of publicly-trusted certificates, v.1.0, 2011. 6.7.3

[129] THE INTERNATIONAL FEDERATION OF RED CROSS AND RED CRESCENT (IFRC) DISASTER RELIEF EMERGENCY FUND (DREF). DREF Final Report: Argentina floods. Tech. rep., The International Federation of Red Cross and Red Crescent, May 2014. 1.5

[130] THE STANFORD SRP HOMEPAGE. The secure remote password (srp) protocol. `http://srp.stanford.edu/`. [Last Accessed : January 2014]. 4.4.2.2

[131] THOMAS WU. The secure remote password protocol. In *Network and Distributed System Security (NDSS'98) Symposium* (1998), vol. 98, p. 97–111. 4.4.2.2, 4.4.2.2

[132] THOMAS WU ET AL. Srp-6: Improvements and refinements to the secure remote password protocol. Submission to IEEE P, 1363, 2002. 4.4.2.2, 4.4.2.2, 4.4.2.2, 4.4.2.2, 4.4.2.2, 8.1.6

[133] THOMSON REUTERS FOUNDATION. Patient privacy in a mobile world a framework to address privacy law issues in mobile health. Tech. rep., Thomson Reuters Foundation and mHealth Alliance and United Nations Foundation and Baker McKenzie and Merck, June 2013. 1.1, 1.4, 1.5

[134] VITAL WAVE CONSULTING. *mHealth for Development: The Opportunity of Mobile Technology for Healthcare in the Developing World*. Washington, D.C. and Berkshire, UK: UN Foundation-Vodafone Foundation Partnership, February 2009. 1.1, 1.3

[135] WEERASINGHE, D., AND MUTTUKRISHNAN, R. Secure trust delegation for sharing patient medical records in a mobile environment. In *Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference on* (2011), pp. 1–4. 9.1

[136] WEERASINGHE, D., RAJARAJAN, M., AND RAKOCEVIC, V. Device data protection in mobile healthcare applications. In *eHealth* (2008), pp. 82–89. 9.1

[137] WHITAKER, B. Problems with mobile security #1. http://www.masabi.com/2007/07/13/problems-with-mobile-security-1/, July 2007. Online, Accessed March 2011. 6.5.2

[138] (WHO), W. H. O., Ed. *PLANNING, IMPLEMENTING, AND MONITORING HOME-BASED HIV TESTING AND COUNSELLING : A PRACTICAL HANDBOOK FOR SUB-SAHARAN AFRICA*. WHO Document Production Services, Geneva, Switzerland, 2012. 1.5

[139] WU, T. The srp authentication and key exchange system - rfc 2945, 2000. 3, 4.4.2.1, 4.4.2.2, 6.6.3, 9.1.1.2

[140] ZHAO, Z., DONG, Z., AND WANG, Y. Security analysis of a password-based authentication protocol proposed to {IEEE} 1363. *Theoretical Computer Science 352*, 1–3 (2006), 280 – 287. 4.4.2.2