Secure Data Storage for Mobile Data Collection Systems

Samson Gejibo, Federico Mancini, Khalid A. Mughal, Remi A. B. Valvik
Department of Informatics
University of Bergen
Bergen, Norway
{samson.gejibo,federico,khalid}@ii.uib.no, remi@valvik.org

Jørn Klungsøyr
Centre for International Health
University of Bergen, Norway
mihjk@cih.uib.no

ABSTRACT

Wireless network infrastructures, notably cellular networks, are becoming a vital element for exchanging electronic data in low income countries. Several key sectors are already leveraging on cellular networks: mobile financial transactions have already gained an enormous success, and the health care sector is also aiming to tackle outstanding challenges like providing basic health care services to remote communities, by using cheap mobile devices. So far, more than ten mobile based health care services are deployed in low-income countries. Among those, mobile data collection is the one used to replace traditional paper form based data collection with electronic digital forms by the use of Mobile Data Collection Systems (MDCS). However, although such systems are often used to collect sensitive health-related data, critical issues like security and privacy of personal data have not been systematically addressed. Particularly, very little has been done to protect data while stored on the phone. This paper focuses on low budget mobile phones with low hardware and software specification, and proposes adequate secure solutions for data storage protection. Our secure storage scheme is flexible enough to be integrated in existing mobile client applications. The solution has been extensively tested and integrated into a production MDCS. For this work, we collaborated with the open-source mobile data collection project, openXdata.

Keywords

mHealth, Mobile Data Collection Systems, Mobile Security, OWASP, J2ME, RMS, Secure Mobile Data Storage

1. INTRODUCTION

The usage of mobile phones, PDAs and other mobile communication devices in the context of health is an emerging part of eHealth. A report by the United Nations Foundation and Vodafone Foundation [23] suggests that close to half the population in low-income countries own or have access to mobile phones. Health care in these nations can be scarce or difficult to access due to restraints such as limited resources, finances and health care workforce, or parts of the population living in remote locations. High mobile phone penetration makes mHealth a viable option for providing better health care through Mobile Health (a.k.a mHealth) systems.

Leveraging mHealth systems makes it possible to cost-effectively provide services like:

- General education and awareness through, for instance, SMS.
- Communicating with and training of health care workers, even in remote locations.
- Tracking of disease outbreaks or epidemics.
- Diagnosing and assisting in treating patients remotely when traveling to a hospital is not an option.

In this paper [1] we are mainly focused on a specific aspect of mHealth, namely remote data collection. However, the work presented is applicable in any context where secure data storage is required.

1.1 Mobile Data Collection Systems

Mobile Data Collection System (MDCS) allow the collection and transmission of data from remote geographical locations to centrally located data storage repositories through wireless or cellular network. It is a combination of a client application running on the mobile devices, wireless infrastructure and remotely accessible server databases. Most of these systems share identical principles and guidelines to collect data remotely. As shown in the Fig. 1, the process begins by designing a form, which contains a set of questions for collecting the relevant data. The form is stored in an accessible server database. Collectors can then download these forms on their mobile device and use them to collect the actual data on the field. The forms that have been filled are stored on the mobile device until it is possible to upload them to the central server.

[1]This paper is an extended version of a short paper accepted at IEEE HealthCom 2012 conference.

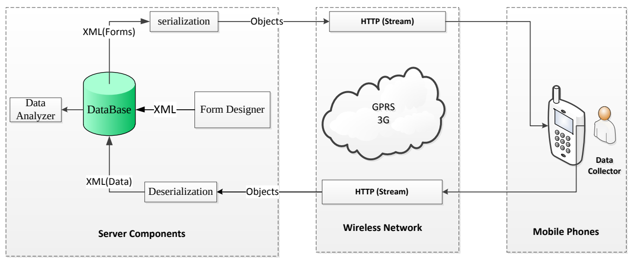

Figure 1: MDCS Data Flow Diagram

These systems clearly handle a lot of sensitive and private information, especially when used for medical purposes, and they should provide an adequate level of protection to the data at all times: both while it is being collected and stored on the mobile device and while being transferred and stored on the server. Insecure data storage is one of the major challenges and ranks at the top of the OWASP Top Ten Mobile Security Risks [18].

The work we present here originated from a collaboration with *openXdata* [15], an open-source MDCS used to run health related projects in low-income countries and has been deployed in different countries such as South Africa, Pakistan, and Uganda [9]. We primarily assessed the system from a security point of view. What we found was that most basic security concerns had not been addressed in a satisfactory manner or had been completely ignored. This led to a security review of similar MDCS, to understand whether they had similar issues or they had found better solutions. We looked especially at those systems that cater for low budget projects, and therefore use low-end mobile devices. In particular, we focused on those systems that provide a Java ME [16] client, like openXdata. Java-enabled phones are, in fact, more likely to be used for low-budget projects than smart-phones or PDAs.

2. SECURITY REVIEWS OF MDCS

We have compared the solutions adopted by some Java ME 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 used offered 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 stores the serialized Java objects, but data elements are still recognizable, while CommCareHQ [2] and EpiSurveyor [5] store all data as XML in clear text. EpiSurveyor, in addition, stores also passwords in clear text if one chooses to have the login form prefilled. Nokia Data Gathering [13] client provides password-protected data encryption. However a quick look at the implementation [14] 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.

MDCS clients like openXdata, Episurveyor and CommCareHQ, have a login procedure in place that authenticates the users remotely on the server when the application is run the first time. Thereafter, openXdata and CommCareHQ store the credentials of the user on the device as a hash, so that off-line authentication is possible by computing the hash of the password entered by the user and comparing it against the stored one. Episurveyor, instead, continues to authenticate the user remotely, although it can save the credentials to pre-fill the login form as we mentioned earlier. The difference is how these credentials are stored. OpenXdata stores a SHA-1 hash of the alphanumeric password, concatenated with a salt received from the server and also stored on the phone, CommCareHQ seems to store SHA-1 hash of a numeric-only password without salt, while EpiSurveyor, as we mentioned stores everything in clear text. CommCareHQ in addition contains a local admin account with a published password, which allows for doing any manipulation on the data, submitting data, creating new users and changing any users password. Nokia Data Gathering, instead, does not seem to have users at all, but some authentication is in place if one chooses to encrypt the data on the phone, since a password must be entered to access and decrypt the data. In this case, the encryption key is the MD5 hash of the password and the authentication mechanism consists in verifying that no decryption error is thrown when some data is decrypted with the key derived form the password entered by the user. One problem with this approach is that it uses error messages from the decryption to detect a wrong key and decide the outcome of authentication, but unless some integrity check is done on the data after decryption, false positives could arise.

Regarding authorization, when multiple users are allowed to share a phone, they should not share storage, or if they do, some mechanism should be in place to prevent them from reading each other's data. In openXdata, a bug has been recently reported that allows a user to do exactly this: if the user has not uploaded all the forms to the server, then other users on the same phone can see those forms. Episurveyor seems to support multiple users, but when testing this feature we experienced that the last user who logged in the application, was automatically logged in again the next time we opened the client, even though the user had been logged out. In any case, it is not acceptable if a mobile phone is shared by many collectors.

2.1 Available Solutions

J2ME does not offer encrypted storage of any kind. 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 [7, 19], and in the MDCS we reviewed at the beginning, are very straightforward and consists in mostly encrypting data with a symmetric key. All the problems related to storing, recovering and managing the key are often ignored. In [20], an interesting overview of the challenges of secure storage in J2ME is presented. Also, many of the challenges are not linked only to security itself, but also software engineering issues. No API for J2ME is in fact available, which provides a complete and flexible system to secure storage for legacy J2ME clients, and the reason is also that J2ME was not intended to create applications that use third party extensions or functionalities, thus the creation of API for this platform is challenging, especially when advanced services must be offered for a generic target system.

3. PROPOSED SECURE DATA STORAGE SOLUTION

3.1 Security Requirements and Practical Constraints

The practical constraints of feature phones such as limited CPU computational power, working memory, persistent storage, battery power, screen size, and input buttons might preclude strong cryptography that is needed to guarantee a high level of security. Most reviewed MDCS projects have been targeted low-income countries where the infrastructure for mobile communication and Internet access are not yet fully developed.

The storage has been designed to account for some typical scenarios in mobile data collection. In particular that multiple users should be allowed to use the same mobile device and the same user can use multiple mobile devices and that Internet access might not be always available. This means that mobile devices can no longer be considered private or personal to an user and that most of the data collection might have to be done off-line. From a security perspective this translates in 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

3.2 Proposed Secure Storage Solution

In section, we explains why the security properties identified in section 3.1 are satisfied. Since standard algorithms are used, such properties are quite straightforward to derive at high level. Besides the robustness of the proposed solution is directly dependent on the strength and correctness of such algorithms, and is therefore dependent on the correctness of the implementation used (in our case the Bouncy Castle API).

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 to a successful authentication grants direct access to this key and the data encrypted with it, but nothing else. Below we discuss some possible ways to implement this approach and discuss whether they satisfy the requirements we identified.

- 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 authorization mechanism prevents unauthorized access only as 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 it if the password is lost (requirement 4). Also, the password can be easily brute forced if the hash is not strong enough and used to impersonate the user to the server.

- 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. This approach would satisfy also requirement 6 if the key is generated with a reasonable iterations, 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.

- 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 to re-encrypt 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-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 synchronization 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).

Fig. 2 shows a solution that satisfies all given requirements based on the previous discussion.

The user will have to register on the phone the first time, and this requires remote authentication on the server with

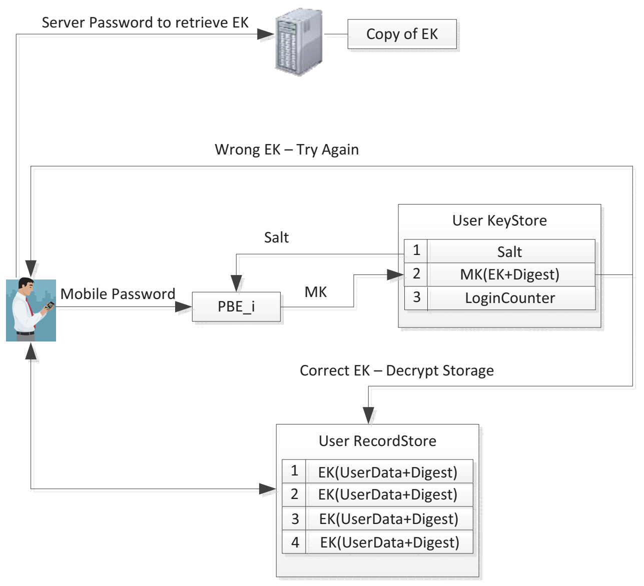

Figure 2: Storage encryption scheme and authentication. MK=Master Key, EK=Encryption Key, PBE_i= PBE algorithm with i iterations. The notation Key(data) stands for data encrypted with Key.

the server password. If the authentication is successful, then an encryption key can be created for the specific user, either by the server or the mobile phone itself, and the user will be asked to choose a new mobile password to access the encrypted storage. In either cases, a copy of the encryption key will be stored on the server. A master key is then created from the mobile password and used to encrypt the encryption key together with its digest. Authentication can now be performed by decrypting the encryption key with the master key derived by the mobile password, and checking that it matches both the decrypted key and the digest appended to it.

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 pass phrase, 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 to use a PBE (password based encryption) algorithm as described in [8], with as many iterations as possible, based on the computation power of the phone.

4. IMPLEMENTING SECURE STORAGE SOLUTION

4.1 Implementation Criteria and Challenges

The ultimate goal of this work is to create a working API implementation of the secure storage solution discussed in section 3, this will be referred to as secureXdata or simply as the API. We identified some important API design criteria, which are discussed in this section.

4.1.1 *Ease of Use / Transparent Design*

This is the most important of the criteria for implementing secure data storage. There are multiple MDCS in use today, many of which lack proper security. If we are to be successful in getting any of these to adopt the API into their applications, integrating the API into an existing system needs to be as hassle free as possible.

Trying to address this issue, we have focused on making the API design as transparent as possible. This means that a programmer using the API to the least possible degree needs to know anything about the implementation of the API. Ideally any programmer with Java ME experience should be able to use the API without having security background.

In Java ME, it is not possible to extend the existing standard classes we're interested in. Instead we mimic the way they work down to the method signatures. This makes integration into existing systems very easy, since the only changes needed to be done is one or two lines of initialization and changing the object type.

4.1.2 *Functionality Decoupling and Flexibility*

The full API provides more than one functionality. Since each of these services cover different areas, they should be as decoupled as possible, preferably completely independent of each other. This makes the API very flexible since a programmer can use only the functionality that is needed, and not being forced to add unwanted functionality.

4.1.3 *Low-end Device Requirements*

As discussed in section 1, the primary target for the API is low end (in terms of memory and CPU).

4.2 Integration Approaches

There are three main approaches for integration, these provide different degrees of control and responsibility for the programmer. That is, as with most other things, with more control comes greater responsibility. If the programmer is knowledgeable when it comes to security, he or she can control how keys are stored and managed. If the programmer is not familiar with security design then this task can be taken care of by the API. The approaches will cover the continuum from programmer control to API control.

4.2.1 *Fully Programmer Control (Manually Manage Keys)*

The `SecureRecordStore` class of the API which has the responsibility of handling user specific data can be initialized by the programmer directly, this means that the keys used will be managed by the program using the API, and the API only provides secure storage, meaning the data stored on the device is secure. This is the most flexible way of using the API. The programmer has full control of how user credentials and data is stored and handled. However, this means that the programmer has to take care of keeping any keys used safe both from a potential attacker, and from being lost. This could, for instance, be handled by the `SecureRecordStore` class provided by the API.

4.2.2 *Partial Programmer Control*

: The API provides a user key store which has the responsibility of handling user specific data and keys. The key store is protected by a password and identified by a user name, if the correct log in information is provided, the key store will be unlocked and the data made available to the program. This means that the programmer does not need to manage user keys and data, they will be stored in the `UserKeyStore` class of the API. Furthermore, once a `UserKeyStore` is unlocked, the `SecureUserRecordStore` is initialized with data from the key store. A user password (the programmer needs to retrieve it from the user), and as a contingency should the users lose their passwords, backing up the key used by the `SecureUserRecordStore` (called the storage key, discussed in greater detail later) is advised. Failing to do so could result in not being able to decrypt data should a user forget his/her `UserKeyStore` password.

This approach takes care of most of the key management required, but still leaves the programmer with a decent amount of control. This approach, as well as the first one, requires no server interaction whatsoever and could, for example, be used to secure local data in an off-line application. The last approach, however, relies on server connectivity.

4.2.3 Secure API control

By extending the `AbsSecureClient` class of the API, an application can delegate the task of handling users entirely over to the API. Once the application starts, the screen control would be given to secureXdata. Depending on the configuration, the controller may prompt users to authenticate the server on the first run, ask them to register or just log in. Once the authentication is successful, control is given back to the application. At this point the `UserKeyStore`,`SecureUserRecordStore` have been initialized and are ready for use. By using this approach, the programmer is completely free from any security concerns. The API will take care of account recovery, logging users in, and initializing the different components with the correct keys. This however comes at the cost of the programmer being confined to the solution provided by the API, and having little or no control over how parts of the application works.

4.2.4 The CryptoTools Interface

Because of this lack or difference in functionality, anything related to cryptography needs to be very flexible to be able to accommodate any requirements or shortcomings a device might have. Through the `CryptoTools` interface the API tries to address this by enabling the developer to make their own implementation using whatever algorithms or libraries they have available. In other words, if there exists some other implementation of the cryptographic primitives that work on the device, the API will be able to use these. Some devices support native Java ME libraries based on the JSR177 [17] specifications, using these might be beneficial with regards to memory and CPU/battery usage compared to using third party libraries such as BouncyCastle [10]. Using native code will also help keep the binary size of the resulting application down. All of the cryptographic operations done by the API use an implementation of this interface.

4.2.5 Bouncy Castle and the Default Implementation

The API comes with a default implementation of `CryptoTools`, this implementation uses BouncyCastle[10]. Since BouncyCastle is a third party API it should support any Java ME enabled phone with the capacity to load it. So the default implementation should work on any device. It also provides a large arsenal of cryptographic algorithms and primitives, making different implementations possible.

On the other hand, since it is not a native implementation there are down sides as well. Computational time and memory usage might be higher than what would be the case with a native implementation, this indirectly effects battery usage. This is because native code can be in a more powerful language or even hardware. Furthermore, the BouncyCastle library is huge (the version used in this project is roughly 1mb), and as such the memory footprint of the API will take a hit, obfuscation helps keeping the size down, but it will still be larger than when using native libraries. In order to use some of the features of Bouncy Castle obfuscation is required, which can add complexity to the testing and debugging aspect of the code.

The default implementation uses RSA for public key encryption. AES in padded with CBC mode and initializing vector (IV) for symmetric cryptography, SHA1 digest, HMAC based on SHA1 digest, Password Based Encryption based on PKCS#12. A random generator based on the `SecureRandomGenerator` class provided by Bouncy Castle. During the next phase of the development of the secureXdata system, these and other algorithms will be tested extensively on a number of different devices to get an overview of what works better on which devices and so on.

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 the attacker manage to crack the local storage, the password the attacker would gain is used only locally and thus the server is safe.

Secondly, by saving the storage key elsewhere, such as on a remote location during registration, we make it possible to recover any encrypted data on the device even if the local password should be lost.

A third consequence of this separation is that it is possible to change the local password independently from any data that might be stored on the device. In other words, if a user changes his or her password, the data on the device does not have to be re-encrypted, we only need to encrypt the `UserKeystore` with the new password.

5. INTEGRATION WITH OPENXDATA

We transformed the insecure standard storage to secure storage by introducing a secure layer on top of the Java virtual machine as shown in figure 3. Introducing the secure layer and making any Java ME application store secure requires nothing but only basic understanding of standard Java ME APIs. All the cryptographic operations and secure logics are completely abstract to the application developers. Unlike the centralized data stores for all user, the secure layer protect user data in individual record store which provides both data confidentiality and access control.

During the integration phase, there are two things that we wish to accomplish: first and foremost we want to keep the data safe on the device, and secondly we wish to separate data from different users. OpenXdata has an issue with their storage system, if a user logs out, the next user logging in has access to the other users data. By using the `SecureUserKeyStore` class of the API both these issues can be solved. Since openXdata uses a centralized class for all storage and stores no data that are shared between users,

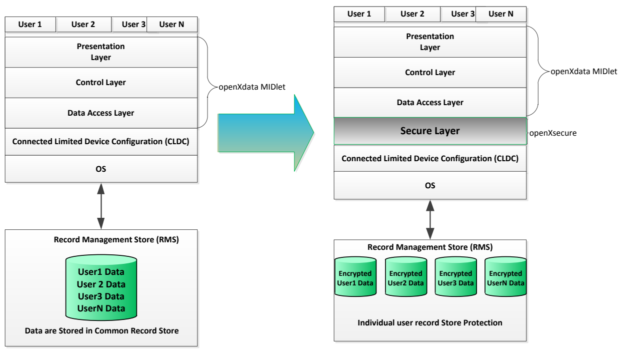

Figure 3: Transformation of Insecure Standard Java ME store to a Secure Store

very few changes were needed to secure storage and provide users with their separate storage areas.

Since we used the openXdata client as our reference system, we have been in collaboration with the openXdata group throughout the development of the API. At the current time it is their wish to try and actually incorporate the API into their system. Together we decided to give priority to the integration of the secure storage part, as this requires only local changes to the client (and possibly some small changes to the server side to allow a recovery procedure for user passwords).

In the current version of openXdata, they no longer store the entire user database on the device, only the registered users. Leveraging on this, we can use the `UserKeyStore` for storing the user credentials instead of storing them in a normal record store. When a user tries to authenticate, if a user's `UserKeyStore` exists, we try to unlock it using the supplied password and stored salt. If the password is entered three times incorrectly, the user is locked out and needs to recover the account by contacting the server. If a user's `UserKeyStore` does not exist, the user name and password are sent to the server for authentication, and, if successful, the salt is returned and the `UserKeyStore` is created.

The integration of the `SecureUserRecordStore` would be done in the same way as described in the in the prototype integration section 5. Server communication would still be insecure, however providing confidentiality locally on the device is a step in the right direction. The client can make a secure connection to the server through HTTPS or some secure protocol.

6. EVALUATION OF PERFORMANCE

While designing the API, the main focus has been to make it easy to integrate with existing systems and easy to use in general. Furthermore, decoupling between the storage and communication parts has also been important. In this section we will evaluate the cryptographic performance of the current design. That means using the default `CryptoTools` implementation that is implemented using BouncyCastle.

6.1 Cryptographic Benchmarking

We primarily used the openXdata model phone, Nokia 2330c-2 [12] for benchmarks and testing. Based on preliminary performance testing that can be found in [6], this phone has fair performance with processor speed of 4.7MHz while still being inexpensive (less than 50$). We will be benchmarking the encryption and decryption speed of the same setup as used in the default implementation of `CryptoTools`. Once we have the encryption and decryption times, we can calculate the throughput speed of both operations, and the combined speed for decrypting followed by encrypting. Decryption followed by encryption is what happens if data is read from a secure record store and then sent to the server, this is a result of the strong decoupling between storage and communication.

If the speeds for the cryptographic operations are much lower than the speed at which data can be transferred to or from the server, then this means that the cryptography is a bottleneck in the system, which we want to avoid.

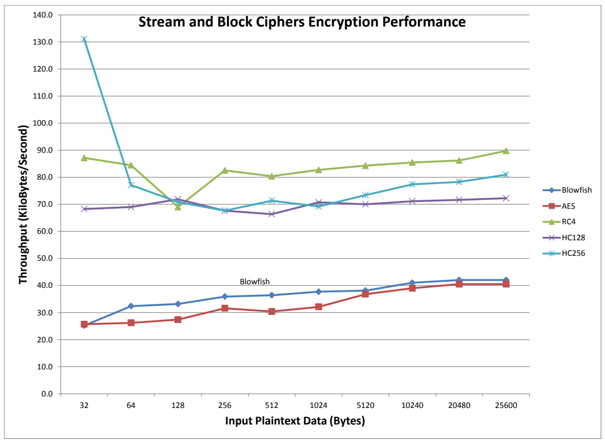

Figure 4: A chart displaying the cryptographic speed results.

As we can see from the Fig. 4, 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 data size of 25KB. Fig. 4 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 throughput speeds on the used device (Nokia 2330c) are reasonably high. Under optimal conditions a GPRS/EDGE [4] as of 2007 speeds up to 22 kb/s can be reached for upload. If 3G network [1] then higher speeds could be expected. It is however likely that the areas where the API and device would be used (low-income countries) would not have optimal conditions.

Since reliable data regarding what connectivity conditions to expect are scarce, we try to put the speeds into perspective using a different factor: the maximum size of a given record store on the device. This also provides some insight in how the throughput is when looking at the other specifications of the phone. The size can be found easily by using the `getSizeAvailable()` method on an empty record store. Doing so on Nokia 2330c shows that the record store can contain a maximum of 131072 bytes, or 128 kilobytes. These bytes would include any bookkeeping or structure overhead from the RMS system itself. Comparing this to the speeds of the larges benchmarked data set, we get the following times: 3,3s encryption, 3,2s decryption, or 6,5s for both. Waiting 3,3s to store some data is a fairly acceptable, but then we are talking about a full record store. The average English word length is somewhere around 8 characters [21]. Lets assume that due to spaces or other formatting characters the average length is 10. This means that a full record store corresponds to roughly 1300 words of text, that is a lot of text for a low-end mobile device. OpenXdata estimates a form being between 1 and 50 kb in size, usually a couple of kilobytes.

7. CONCLUSIONS

In general, all modern smart phones equipped with operative systems like Blackberry, Android and iOS provide a crypto API to develop secure applications. However, we are developing a secure solution for the Java ME platform, which lacks support for any kind of data security [3, 24], and we target low-end phones, so that solutions that might be adequate for high-end phone like smart phones, are not an option in our context. The solution we implemented is based on a custom protocol developed by considering the specific constraints of MDCS [11], 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 with different security requirements. Most of the secure API objects that are exposed to the programmer are very similar to the Java ME counterparts in terms of methods and behavior. This means that any programmer who knows how to use the normal Java ME classes will also know how to use the secure ones.

We have developed our own prototype MDCS using the API, and tested it on various phones with different settings in order to collect experimental data on the performance of the API. The results are encouraging, since the performance with the default security settings was acceptable also on very low-end phones, and the openXdata integration is proceeding smoothly. The secure solution has been integrated into a production openXdata code base and ready to be tested in the field.

We have focused our research on feature phones instead of smart phones because the market share of feature phones is about 80countries [22], so it will take some time to migrate to smart-phones, although prices are falling. Besides, there are still many MDCS in use which leverage on a J2ME client, and using our API is a simple way for them to quickly add security to their existing system while they migrate to other platforms.

Move to smart phones is inevitable, and openXdata community has started work to implement an Android client, and we are collaborating to port our secure API to Android.

8. REFERENCES

[1] 3rd generation mobile telecommunications(3G). `http://en.wikipedia.org/wiki/3G`. Online, Accessed December 2011.

[2] CommCareHQ. `http://www.commcarehq.org`. Online, Accessed November 2011.

[3] T. Egeberg. Storage of sensitive data in a Java enabled cell phone. Master's thesis, Høgskolen i Gjøvik, 2006.

[4] Enhanced Data Rates for GSM Evolution(EDGE). `http://en.wikipedia.org/wiki/Enhanced_Data_Rates_for_GSM_Evolution`. Online, Accessed December 2011.

[5] Episurveyor. `http://www.episurveyor.org/`. Online, Accessed March 2011.

[6] S. Gejibo, K. A. Mughal, F. Mancini, J. Klungsøyrg, and R. B. Valvik. Challenges in implementing end-to-end secure protocol for java ME-based mobile data collection in low-budget settings. In *ESSoS*, Lecture Notes in Computer Science, pages 38–45. Springer, 2012.

[7] W. Itani and A. Kayssi. J2ME application-layer end-to-end security for m-commerce. *Journal of Network and Computer Applications*, 27(1):13–32, January 2004.

[8] B. Kaliski. RFC 2898 - PKCS #5: Password-based cryptography specification. `http://www.ietf.org/rfc/rfc2898.txt`, 2000. Online, Accessed April 2011.

[9] J. Klungsøyr, T. Tylleskar, B. MacLeod, P. Bagyenda, W. Chen, and P. Wakholi. 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*, pages 36–44. Karlstad University Studies, 2008.

[10] T. Legion Of the Bouncy Castle. `http://www.bouncycastle.org/`. Online, Accessed March 2011.

[11] F. Mancini, K. Mughal, S. Gejibo, and J. Klungsoyr. Adding security to mobile data collection. In *Healthcom 2011 - 13th IEEE International Conference on e-Health Networking Applications and Services*, pages 86 –89, june 2011.

[12] Nokia 2330c classic. `http://www.developer.nokia.com/Devices/Device_specifications/2330_classic`. Online, Accessed September 2011.

[13] Nokia Data Gathering. `http://projects.developer.nokia.com/ndg`. Online, Accessed November 2011.

[14] Nokia, Nokia Data Gatherings(NDG). `https://github.com/nokiadatagathering/ndg-mobile-client`. Online, Accessed September 2011.

[15] openXdata. http://www.openxdata.org. Online, Accessed March 2011.

[16] Oracle. Java ME reference. `http://www.oracle.com/technetwork/java/javame/index.html`. Online, Accessed March 2011.

[17] Oracle Inc. Security and Trust Services API for J2ME(SATSA).

http://java.sun.com/products/satsa/. Online, Accessed March 2011.

[18] OWASP. Mobile Security Project. https://www.owasp.org/index.php/OWASP_Mobile_Security_Project. Online, Accessed March 2012.

[19] S. M. A. Shah, N. Gul, H. F. Ahmad, and R. Bahsoon. 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*, pages 887–896.

[20] T. Egeberg. Storage of sensitive data in a Java enabled cell phone. http://egebergweb.com/tommy/masterfiler/masteroppgave2.pdf. Master Thesis, Accessed on March 2012.

[21] C. Z. G. N. W. unit based multilingual comparative analysis of text corpora. http://speechlab.tmit.bme.hu/publikaciok/. Online, Accessed January 2012.

[22] Vision Mobile. Global Smartphone Penetration . http://techcrunch.com/2011/11/28/its-still-a-feature-phone-world-global-smartphone-penetration-at-27/. Online, Accessed August 2012.

[23] 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.

[24] B. Whitaker. Problems with mobile security #1. http://www.masabi.com/2007/07/13/problems-with-mobile-security-1/, July 2007. Online, Accessed March 2011.