WhatsApp Database Encryption
Project Report

D. Cortjens A. Spruyt W.F.C. Wieringa

31th of December, 2011

Abstract

The purpose of this project has been to research if it is possible to decrypt a WhatsApp database and access the information within. To answer this question a few different analysis has been performed on the database. With this analysis in mind, further decompilation and investigation of the database have been performed. This projects main conclusion is that different platforms have different ways of dealing with the WhatsApp database and also use a different way of encrypting the files. The Android implementation is using AES with a 192-bit key and can be decrypted with a Python script as we show in this report. The BlackBerry environment has it's own software encryption build in which is at least partially used for the encryption, but this has not been researched to the full extend. The BlackBerry data can be decrypted (decoded) when extracting the chip of the device. The iPhone has hardware encryption and can be decrypted on-the-fly with existing software. These findings may be useful in any further research on the WhatsApp application.

Contents

Chapter 1

Document Information

1.1 Description

This document is the project report for the WhatsApp Database Encryption project. This project is part of the Security of Systems and Networks subject within the master course System and Network Engineering at the University of Amsterdam.

1.2 Disclaimer

Chapter 2

Project Information

2.1 Introduction

During the last few years the number of mobile devices has grown enormously. Nearly everyone owns a mobile device like a phone or tablet. Mobile devices are now little computers with the power of a normal desktop or notebook computer. People do not need a desktop or notebook computer any more, because they can do everything from their mobile device and *store everything on them*. They are used to browse the web, make appointments and communicate with other people. One way to communicate with a mobile device is WhatsApp. It's a free application that sends messages through the data connection of the mobile device. WhatsApp has become a very popular application for sending messages. It's one of the first applications people install on their mobile device. WhatsApp is cross platform with versions available for the Android, BlackBerry, iPhone and Symbian operating systems.

In the world of Digital Forensics WhatsApp has become an important and useful source of information. This is because WhatsApp stores messages and everything that can be sent in these messages (audio, locations, pictures, video, etc.) in an SQLite database. The database is normally on the memory card in the phone but can be present on the internal memory of the phone when there is no memory card present. Examining the database is a way to extract information from the device without tampering with the device itself.

2.2 Problem

WhatsApp has changed a lot over the last couple of months. They've encrypted the messages sent over the data connection and even encrypted the databases stored on the memory card or internal memory. This has made it very difficult for Computer Crime Experts to search for the, in many cases important, communication between people. The Digital Forensics world is in need of decryption capabilities for these databases. This will allow this source of information to be used in the fight against crime.

2.3 Position

This project was started with a search on the internet to rule out that someone else had already completed the goals set for this project. The search didn't gave the indication that anyone had already solved the problem. The search did resulted in some papers about identifying encryption and decryption schemes. One of the papers that turned up was 'Theory of Cryptography' [1] which describes working with information entropy. This was a good starting point for examining the encrypted WhatsApp databases. The internet search was continued to find papers about the type of algorithm that might have been used. Eventually a selection of possible algorithms was made to rule out some of the papers.

2.4 Questions

2.4.1 Main

Can the WhatsApp database be decrypted and the information within accessed?

2.4.2 Sub

The central question is answered with the help of the following subquestions:

1. *What encryption is used for the WhatsApp databases?*
2. *How strong is this encryption?*
3. *How is the encryption key generated?*
4. *Where is the encryption key stored?*
5. *What are the differences between the Android, BlackBerry and iPhone operating system regarding database encryption?*
6. *Is it possible to decrypt the WhatsApp database?*

2.5 Goal

2.5.1 What encryption is used for the WhatsApp databases?

Determine which encryption cipher is used for the database on the Android, BlackBerry and iPhone operating system.

2.5.2 How strong is this encryption?

Determine how strong the encryption is on the Android, BlackBerry and iPhone operating system.

2.5.3 How is the encryption key generated?

Determine how the encryption key is generated on the Android (and BlackBerry) operating system.

2.5.4 Where is the encryption key stored?

Determine where the encryption key is stored on the Android (and BlackBerry) operating system.

2.5.5 What are the differences between the Android, BlackBerry and iPhone operating systems regarding database encryption?

Summarize the differences between the Android (and BlackBerry) operating system.

2.5.6 Is it possible to decrypt the WhatsApp database?

Determine whether or not the WhatsApp database can be decrypted.

2.6 Scope

The scope of the project is defined as follows:

- The global information for goal one and two in section 2.5 is collected for the Android, BlackBerry and iPhone operating systems;
- The detailed information for goal three and four in section 2.5 is collected for the Android operating system;
- The detailed information for the BlackBerry operating system is collected if this is completed or if it isn't possible for the Android operating system;
- The program for decrypting the database has to work on an unrooted mobile phone from a physical dump;
- The program for decrypting the database may work on a rooted mobile phone when this isn't possble for an unrooted mobile phone.

Chapter 3

Approach

For this project, a forensic approach was chosen. This meant that every step should be documented and that the actual research on files should be done with (physical) dumps and not the actual mobile device. This is done by Digital Forensics departments all over the world to be able to prove certain findings without changing the state of the source device. In this way every device can be contra investigated by other departments or third parties to verify the truth in crime cases.

This project had to come up with a solution that would give Computer Crime Experts a forensicly sound way of decrypting the WhatsApp database, to allow the use of the information within the databases. So this was not a project for analysing the application and coming up with suggestions to enhance the application.

Some scenarios were planned to be able to determine the important aspects of the WhatsApp database encryption. Not all of these scenarios have been executed. However some gave a really good and early insight in how the encryption was handled. In these scenarios the strength of the encryption was studied with an entropy analysis using CrypTool and by compressing the files. The location of the encryption key was studied with software analysis with the use of decompilers.

Chapter 4

Android

4.1 Specifications

The Android phone used in this project is a HTC Hero. The phone was released in 2009.

Brand:	HTC
Model:	Hero
Serial Number:	HT972L907258
IMEI:	357988020265000
CPU:	Qualcomm MSM 7200A 528MHz
OS:	Android 2.1

Table 4.1: Android phone specifications

4.2 Preparation

The HTC Hero used belongs to the University of Amsterdam. It has been previously used in serveral projects and was therefore equipped with a Cyanogen Android 2.3 ROM[1]. This meant the phone was rooted[2]. To make sure the phone was at factory settings it has been flashed to a stock ROM. This was performed in accordance with the HTC Hero's flashing process[3]. After the flashing process a Vodafone Prepaid SIM card with data connection was inserted and WhatsApp was installed. Using a private phone some messages were sent to the HTC Hero. From the HTC Hero some messages were sent back to the private phone. These messages were predefined and are shown in table 4.2.

HTC Hero	**private phone**
Hi!	
How are you?	
	Hi!
	I'm fine.
Okay!	
Bye!	
	Bye

Table 4.2: WhatsApp messages

[1] http://www.cyanogenmod.com
[2] http://en.wikipedia.org/wiki/Rooting_(Android_OS)/
[3] http://forum.xda-developers.com/wiki/index.php?title=Flashing_Guide_-_Android/

4.3 Physical dump

For this project a Forensic approach has been chosen, therefore it is not possible to directly retrieve the information from the phone. A physical dump of the phone and its memory card has been made with the UFED Physical Analyser[4] software. It is not possible to create a physical dump of an unrooted HTC Hero. However it was possible to create a physical dump of the micro Secure Digital memory card in the phone by using the FTK Imager[5] software, see table 4.3. From this physical dump the WhatsApp database has been extracted with the EnCase[6] software, see table 4.4.

File Name:	android_microsd.E01
File Extension:	E01
File Size:	1.974.146.937 bytes
MD5 Hash:	95A81DC80B962F10C98E2A3E918A8F08
SHA1 Hash:	7D6BD11A386259863972B1CF3308ABE8F36E8196

Table 4.3: Android microSD physical dump specifications

File Name:	msgstore.db.crypt
File Extension:	DB
File Size:	8.208 bytes
MD5 Hash:	F21210FB7F215604B64D84EEB221DB37
SHA1 Hash:	770E20597D9398C75CF50BF1E19ADE30BF4148F9

Table 4.4: Android database file specifications

4.4 Analysis

This section describes the research of the encryption solution employed by WhatsApp. As a primer about encryption techniques and solutions 'Guide to Storage Encryption Technologies for End User Devices' [2] was used.

4.4.1 Entropy

An entropy analysis can be an informative tool when investigating an unknown file. This is especially true for supposedly encrypted files. When a file is correctly encrypted the cipher text should have little to no relation to the plaintext. As a consequence an encrypted file should have little internal patterns. An extremely simple way to express the degree of randomness inside a file is the degree in which the file can be compressed. To quantify the difference between the plain encrypted datafile and a compressed encrypted datafile, the size before and after compression can be compared.

```
ls -lh msgstore-2011-11-01.1.db.crypt*|awk '{print $5 "\t" $9}'

33K    msgstore-2011-11-01.1.db.crypt
20K    msgstore-2011-11-01.1.db.crypt.gz
```

Figure 4.1: Android compressed database file

Figure 4.1 shows that the encrypted file compresses reasonably well. This means that at least one implementation detail has been overlooked. The practical consequences of this flaw cannot yet be stated. Cryptool[7] confirmed on these findings by rating the entropy with a 5.03 as shown in figure 4.2.

[4] http://www.cellebrite.com/forensic-products/forensic-products/ufed-physical-analyzer.html
[5] http://accessdata.com/products/computer-forensics/ftk/
[6] http://www.guidancesoftware.com/forensic.htm
[7] http://www.cryptool.org

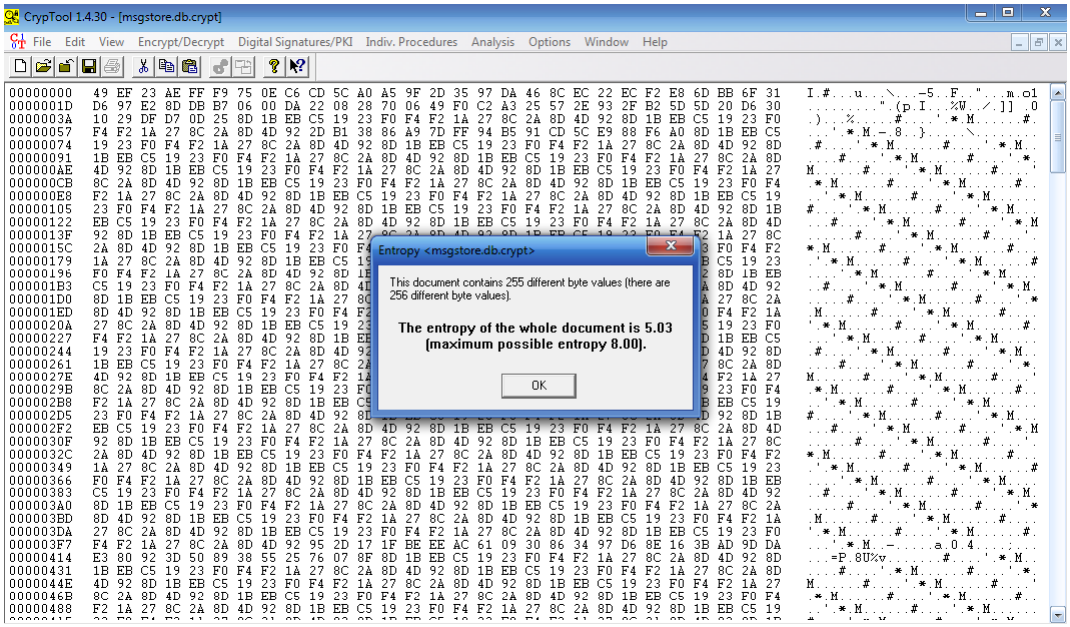

Figure 4.2: Entropy check in CrypTool on Android database file

4.4.2 Database

After having performed an entropy analysis and having come to the conclusion that something is amiss a *check must be* made to verify that the data is actually encrypted. A common Unix program called 'strings'[8] has been used to do so. The program strings takes input and checks if there are sequences of four or more printable bytes and prints these to stdout. This can be used for other things like the identification of binary files. In this case the output did not contain any readable strings. This means the data is indeed obfuscated or encrypted. After concluding that the file is indeed encrypted, but not in an entirely correct manner, a closer look at the data file itself was warranted. Since it is a binary file, a hex editor is the normal way to go about this. The hex dump of the file made a simple fact immediately apparent: a great number of rows are repeated. This information leak can help identification of the encryption cipher and its strength.

```
hd -v  msgstore-2011-11-01.1.db.crypt | sed -r 's/^[0-9a-f]+//'|sed -r 's/\|.+$//' | sed -r 's/ //g'|perl -
e 'my %count;while(<>){chomp($a=$_);$count{$a}++;}foreach $i ( keys %count ){printf ("%4.4d %s
\n",$count{$i},$i)};'|sort -r|head -8
```

Figure 4.3: One-liner to show block count

```
0328 8d1bebc51923f0f4f21a278c2a8d4d92
0024 6e1b6f61fac8de5cfb515864dc2f9e84
0023 88b314488ab7479cf33a43d8a75eee9e
0023 5e986182b040571cda676af29b4e04b5
0022 cabc02fecb9475d0dfae6032c3ea22a0
0020 54171b5d363ece05068bc075f9df74f8
0020 023f5d05a75db6d02ac4b632eb8ad215
0019 aa2def05503256616b7eb5e9804dc454
```

Figure 4.4: 128-bit pattern count on the Android database file

[8] http://unixhelp.ed.ac.uk/CGI/man-cgi?strings

```
0084 00000000000000000000000000000000
0002 77697468204368726973746 96e656418
0002 6e6368207769746820436872697374 69
0002 6c756e6368207769746820436872697 3
0002 68207769746820436872697374696e65
0002 35016c756e6368207769746820436872
0001 77697468204368726973746 96e65641f
0001 74696e6564180a0335016c756e636820
```

Figure 4.5: 128-bit pattern count on a different unencrypted database file

Figure 4.5 show that an SQLite database contains a lot of data that is all zero's. The idea that the top pattern of the encrypted database consists of encrypted zeros does present itself. If a modern algorithm such as AES[9] or DES[10] is used it should be resistant to plain-text attacks[11]. However if a naive approach is used it might be possible to compute the key. Since it uses a 128 bit block size it rules out DES or 3DES[12]. In fact it severely limits the different algorithms which can be used.

For instance a XOR cipher[13] in which the data is XOR'ed with a pattern would not be resistant to such an attack. A simple strategy for such an attack would be to find the pattern with the highest count and assume that these represent all encrypted zeros. Since the identity of XOR is 0 this pattern would actually be the key with which the file was encrypted. This was not the case for this file.

```
$ ./xorPattern ~/school/ssn/proj/msgstore-2011-11-01.1.db.crypt
'8d1bebc51923f0f4f21a278c2a8d4d92' | hd | less

00000000  c4 f4 c8 6b e6 da 85 fa  34 d7 7b 2c 8f 12 60 a7  |...k....4.{,..`.|
00000010  23 95 4f 04 0b 9f 2e 3e  80 88 ff 61 73 34 b6 a8  |#.O....>...as4..|
00000020  00 c0 5c c3 19 f9 d2 fc  da 6a 21 c5 da 4f ee b7  |..\......j!..O..|
00000030  da 35 78 ea ab 7e ad d4  24 2a 37 a5 f5 5a 40 b7  |.5x..~..$*7..Z@.|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000060  a0 aa d3 43 b0 5e 0f 60  47 8b ea d0 c3 05 bb 32  |...C.^.`G......2|
00000070  a2 77 e3 7d 77 82 d9 4d  44 09 23 b9 32 44 b5 4a  |.w.}w..MD.#.2D.J|
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Figure 4.6: Program output run through 'hd'

Figure 4.6 shows the one-liner and its output. This shows that the recurring line does indeed change to all zeros, however the rest of the file does not become readable ('strings' still doesn't shows the data). At this point it is highly unlikely that a XOR cipher is used.

Comparison of databases

After the mentioned program to count recurring blocks is run on a database from a different Android phone the output is rather surprising. Working with the assumption that the most common block represents a block of zero's in the original it is expected that when we run this one-liner on a different database a different block will be most common. This is expected because the database is presumably encrypted with a different key. This turned out not to be true as the exact same block as in the first case was returned. The chances of this occurring with two different keys are in the order of 2^{128} With the previous assumption that the data file has been encoded with a standard 128-bit block cipher in mind, the conclusion that this database is also encoded with the same key can be made. Since the two phones are unrelated it could be postulated that this key is in fact the same for all Android phones. This would constitute a serious encryption problem.

Search for binary key

A practical consequence of the suspected zero block is that to test a key it is not necessary to decrypt the entire file, but only a single block. For instance if the original file is 500KiB the amount of data that must be decrypted is reduced to only 16 bytes. This severely reduces the amount of work that has to be done and makes the following attack plausible. Instead of trying a full brute force attack on the block, all byte

[9] http://www.ietf.org/rfc/rfc3268.txt
[10] http://tools.ietf.org/html/rfc1829/
[11] http://media.ccc.de/browse/congress/2010/27c3-4203-en-distributed_fpga_number_crunching_for_the_masses.html
[12] http://www.ietf.org/rfc/rfc2420.txt
[13] http://en.wikipedia.org/wiki/XOR_cipher/

sequences (in a number of different representations) in a file are tried. This file could be the database itself (i.e. they store the key in unencrypted form in the file) or a full dump of the entire phone's storage (i.e. they saved they key somewhere on the non-volatile storage of the phone). Note that if the password is derived from something on the data dump this attack will not work. In practical terms this means that if the key is base64 encoded or compressed and stored on the image this will also not work. This program could also be used on an uncompressed version of the installation package to see if the key is hard-coded into the binary. This is mentioned in 'Scenario-based Analysis of Software Architecture' [3].

4.4.3 Software package

Android runtime

The WhatsApp program runs on top of the Dalvik Virtual Machine (VM)[14]. It is based on the Java Virtual Machine (JVM)[15]. However the Dalvik VM is register based where as the JVM is stack based. This difference means the byte code of both are incompatible. Apart from the VM and byte code Android also provides a number of libraries. A number of these are direct ports from the Java world. One of these is Javax.Crypto. This library provides a number of cryptographic classes. This makes encountering crypto constants[16] such as the AES tables when analysing the WhatsApp database unlikely.

APK

Android packages are put inside an Android PacKage[17] (APK)[18] files. These are ZIP files with a special layout and contains the following folders:

- META-INF:
  - MANIFEST.MF
  - CERT.RSA
  - CERT.SF
- res: *the resourses folder of the APK*
- AndroidManifest.xml: *describing the name, version, access rights and the referenced library files for the application*
- classes.dex
- resources.arsc

The classes.dex file in the APK contains the actual compiled program code that will run on the Dalvik VM. For the purpose of tracking down the encryption keys this is the interesting part of the application. This is mentioned in 'The Structure of Android Package files' [4].

The current state of Dalvik byte code analysis is behind the state of affairs of JVM byte code. A possible reason for this is that it's possible to translate Dalvik byte code into JVM byte code and then use all the tools available for JVM byte code. The program used for this translation was 'dex2jar'[19].

Decompilation

After this translation a host of programs become available to do the binary analysis. There are a number of disassemblers and decompilers available. Disassemblers translate the binary into mnemonic opcodes and decompilers are able to translate into higher languages, in this case Java. The choice was made to do an initial analysis on the decompilation from JD-Gui[20] and then to use a disassembler on an as-needed basis. The reason for this choice is that it is much faster to read a high level language than a disassembly.

[14] http://source.android.com/tech/dalvik/dalvik-bytecode.html
[15] http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html
[16] http://crypto.stackexchange.com/questions/1137/how-to-choose-constants-in-a-cryptographic-function/
[17] http://en.wikipedia.org/wiki/APK (file format)/
[18] http://sites.google.com/site/io/inside-the-android-application-framework/
[19] http://code.google.com/p/dex2jar/
[20] http://java.decompiler.free.fr/?q=jdgui

The decompilation of the binary revealed that the Android version of WhatsApp used some form of obfuscation. Two obfuscation methods can be identified: name obfuscation and string obfuscation. The name obfuscation renames all classes, methods and variables. This renaming ensures that most of the classes in the binary have a name which is two lower case letters and/or numbers. An example of such a name is 'a6'. Both the class and object methods are also renamed in this fashion. In this case the renaming is even worse since methods that have a different signature, i.e. take different parameters or return a different type, both get the same name. It is worth noting that if a different binary is decompiled all the class names are different, this suggests that the manner in which the names are assigned is a stochastic process. Apart from this renaming strings in this binary are also obfuscated. The technique used for this is simpler: as the class is loaded all strings are rewritten to their actual values. This is mentioned in 'An evaluation of current java bytecode decompilers' [5]. The reason for this obfuscation is probably to make it harder for people to do binary analysis on the application. Another reason is to counter the attack in section **??** where an unobfuscated, unencoded or otherwise transformed key is extracted from the binary. It should still be possible to run this program on a memory dump of the application.

Obfuscated code

Even with the presented obstacles it should still be possible to identify where the key is generated or possibly even stored. A cursory glance at the decompilation presents us with class 'a6'. This class uses a 'SecretKeySpec' and 'Cipher.getInstance' both with obfuscated parameters. This constitutes a good place to start a static analysis. At this point it can't be ruled out that these classes are not responsible for the encryption of the database. However they are not responsible for the encryption of traffic as that is handled by the javax.net.ssl package used in a number of other classes. Figure 4.7 shows the use of the mentioned methods. The trick will be to de-obfuscate 'k8.k' and possibly 'z[5]' where 'k8.k' is possibly the used key and 'z[5]' is the used encryption algorithm.

```
\$ java read | hd
00000000  34 6a 23 65 2a 46 39 2b  4d 73 25 7c 67 31 7e 35  |4j#e*F9+Ms%|g1~5|
00000010  2e 33 72 48 21 77 65 2c                           |.3rH!we,|
00000018
```

Figure 4.7: Obfuscated class a6

The decompilation of 'a6' was still reasonably coherent and readable. This does not hold for the decompilation of k8. Figure 4.8 shows a schematic view of k8. It is suspected that the earlier mentioned string obfuscation is the reason for this. The string that eventually will become k8.k is in the class stored as a translated string. When the class is loaded this and other strings will be translated back into their original and usable representation. This translation back can be seperated into two stages: a number of transposition rounds and a XOR operation with 0x12. The XOR operation is one which is trivial to undo correctly. The transposition part of the alogrithm will take significantly longer to reverse.

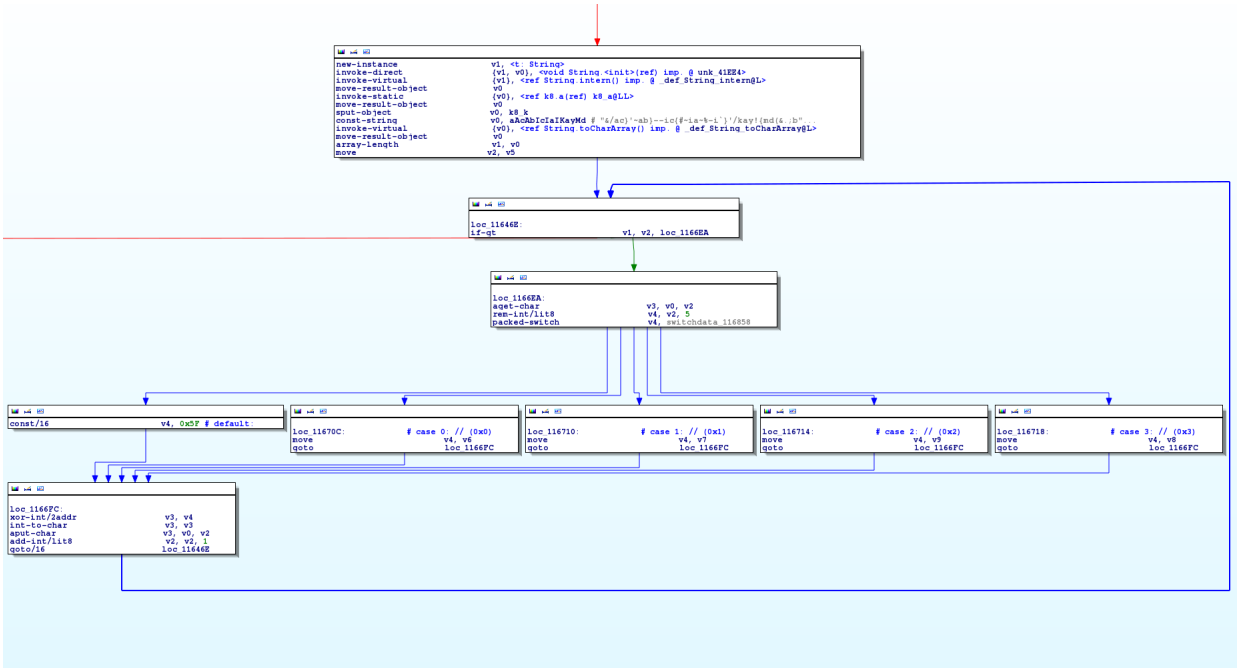

Figure 4.8: Schematic view of k8 in IDA

Running the code

Another possibility for static analysis does present itself: active analysis. Since the binary was previously translated into JVM byte code, it should be possible to actually run parts of the code on a computer based JVM and simply outputting the de-obfuscated key. This process would amount to simply asking nicely for the encryption keys.
The Java class file for k8 can simply be extracted from the translated JAR[21] archive. All that is needed is a small wrapper which reads k8.k after it has been automatically initialized and de-obfuscated and then outputs the result. Figure 4.9 shows a possible implementation. Please note that this outputs the key in binary.

```
hd -v msgstore-2011-11-01.1.db.crypt | sed -r 's/^[0-9a-f]+//'|sed -r 's/\|.+$//' | sed -r 's/ //g'|perl -
e 'my %count;while(<>){chomp($a=$_);$count{$a}++};foreach $i ( keys %count ){printf ("%4.4d %s
\n",$count{$i},$i)};'|sort -r|head -8
```

Figure 4.9: Wrapper code

Figure 4.10 shows the output of the wrapper as run through 'hd'. This is the AES 192-bit key that is used to encrypt the database.

```
\$ java read | hd
00000000  34 6a 23 65 2a 46 39 2b  4d 73 25 7c 67 31 7e 35  |4j#e*F9+Ms%|g1~5|
00000010  2e 33 72 48 21 77 65 2c                           |.3rH!we,|
00000018
```

Figure 4.10: Wrapper output run through 'hd'

4.5 Script

For decrypting a WhatsApp database file on an Android phone a script or program should be created. So a Python script was created to perform the decryption process. This script is build to take an encrypted database file as the first argument and an output file name as the second argument. The script contains the encryption key as an hex value. Then the script uses the Crypto.Cipher.AES package to perform the

[21] http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/jar.html

decryption of the given database file. Eventually the decrypted data is written to a new file with the given output file name. This is a forensicly sound process that will result in a source file (encrypted database file) and an output file (decrypted database file). The Python code without the key is shown in figure 4.11.

```
from Crypto.Cipher import AES
import sys

fileIn = open(sys.argv[1],"r").read()

fileOut = sys.argv[2]

PADDING = '{'

secret = ""

DecodeAES = lambda c, e: c.decrypt(e).rstrip(PADDING)

# create a cipher object using the random secret
cipher = AES.new(secret,1)# 1

output = open(fileOut,"w")

# decode the encoded string
output.write(DecodeAES(cipher, fileIn))
```

Figure 4.11: the Python code

Chapter 5

BlackBerry

5.1 Specifications

The BlackBerry phone used in this project is a Research In Motion BlackBerry Curve 9300. The phone was released in 2010.

Brand:	Research In Motion
Model:	BlackBerry Curve 9300
PIN:	287FCD93
IMEI:	358966043972108
CPU:	Marvell Tavor PXA930 624 MHz
OS:	BlackBerry OS 5.0

Table 5.1: BlackBerry phone specifications

5.2 Preparation

The Research In Motion BlackBerry Curve 9300 used belongs to the University of Amsterdam. It has been ordered for this project and was brand new with stock settings. On first boot a Vodafone Prepaid SIM card with a data connection was inserted and WhatsApp was installed. With a private phone some messages were sent to the BlackBerry Curve 9300. From the BlackBerry Curve 9300 some messages were sent back to the private phone. These messages were predefined and are shown in table 4.2 of chapter 4.

5.3 Physical dump

This project employed a Forensic approach and as a consequence it is not possible to retrieve the information directly from the phone. So an attempt has been made to retrieve a physical dump of the device with the software UFED Physical Analyser. At this time it isn't possible to create a physical dump without soldering the chip from the device. However a physical dump was created of the micro Secure Digital memory card in the device by using the software FTK Imager, see table 5.2. From this physical dump the WhatsApp database was extracted using the software EnCase, see table 5.3.

File Name:	blackberry_microsd.E01
File Extension:	E01
File Size:	1.978.342.775 bytes
MD5 Hash:	81237AD307AE79E43B1FC33BED171B01
SHA1 Hash:	A5252EEBC063B60A16D42196BEEE4B8AC5C51103

Table 5.2: BlackBerry microSD physical dump specifications

File Name:	messageStore.db
File Extension:	DB
File Size:	5.212 bytes
MD5 Hash:	FC3DE3EB5EF14DC1B9E60A774CCE073C
SHA1 Hash:	E36D39ABFDCB174504D401D055BE3A56972C0163

Table 5.3: BlackBerry database file specifications

5.4 Analysis

5.4.1 Entropy

As with the Android WhatsApp database analysis this was started by performing an entropy test on the database. This was done by compressing the file and observing how well it compresses. As figure 5.1 demonstrates the compressed file is even larger than the uncompressed file. It can be concluded that this file has an extremely high entropy. This suggests that the encryption was not done in an ECB[1] mode, but a mode such as CBC[2], that the file has been compressed before possibly being encrypted[3].

```
-rwxr-xr-x 1 root  root     5212 Dec  6 12:46 messageStore.db
-rwxr-xr-x 1 root  root     5709 Dec  6 12:46 messageStore.db.bz2
```

Figure 5.1: BlackBerry compressed database file

Cryptool confirmed on these findings by rating the entropy with a 7.99 on the physical dump of the BlackBerry as shown in figure 5.2.

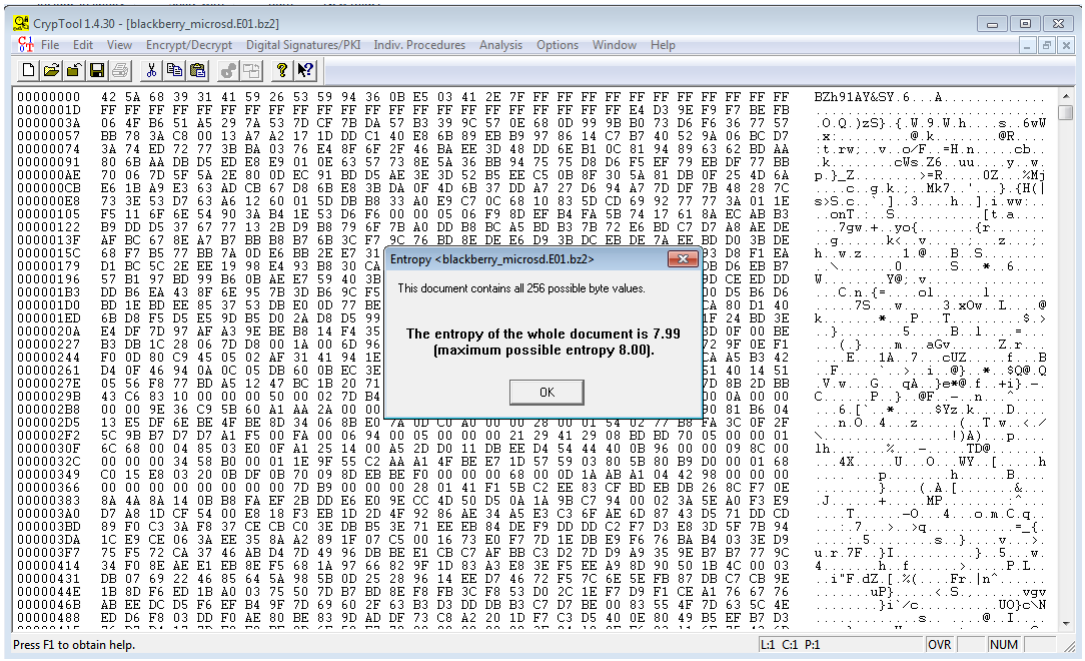

Figure 5.2: Entropy check on BlackBerry MicroSD physical dump

5.4.2 Database

It is a good idea to take a longer look at the actual file. What peculiarities can be discovered? The file header consists of 'REMF' and possibly some following octets. This header might consist of a simple header

[1] http://www.itl.nist.gov/fipspubs/fip81.htm
[2] http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation#Cipher-block_chaining_.28CBC.29
[3] http://www.blackberry.com/developers/docs/7.1.0api/net/rim/device/api/crypto/DecryptorFactory.html

or might have meta-data interspersed with the data. This points to a special file format (REM) developed by Research In Motion (RIM). According to the BlackBerry cryptographic API it turned out that REM files are files that have been encrypted using the BlackBerry device's encryption option. For these files an AES 256-bit encryption is used.

```
00000000  52 45 4d 46 01 00 54 80  c8 fe 20 3a 10 10 90 6b  |REMF..T... :....k|
00000010  e5 86 cc c8 f3 89 d8 00  5d da 1f 7f c2 9e 1b 27  |........].......'|
00000020  d2 a7 96 e2 0e 05 37 9b  6a 1b bd a2 75 06 cf a7  |......7.j...u...|
```

Figure 5.3: BlackBerry encrypted file header

When a file is encrypted it appends a REM[4] suffix to the file name. The header tells us that we are dealing with a file that took advantage of the encryption possibilities of the BlackBerry encryption system. The phones private key is stored in a specialy protected area. This area is protected with a device password[5]. There are two basic ways of encrypting the content.

When the device encrypts files, it uses a randomly generated device key which is stored in its NVRAM. If the device is erased or re-installed the NVRAM is obviously cleared and the key will no longer be available. Therefor the file can't be opened any more.

If files are encrypted using a device password it obviously requires this password to open a BlackBerry encrypted REM file. The proper way to open the encrypted file on the microSD memory card is to use the application it is made with. When this is done file decryption occurs and the file moves to the main memory to be read. It uses a master key that is stored on the microSD media card to encrypt files. This prevents having to decrypt or re-encrypt all files when encryption is disabled or the password has been changed. If the microSD card is moved to another device that does not use a device password or isn't able to decrypt the microSD card master key, the device prompts the user to enter the microSD card password manually. So when this encryption is in place, it is depending on the knowledge of the master key whether or not will be able to open the encrypted files.

5.4.3 Environment

It is outside the scope of this project to completely explore the operating system that is used by the BlackBerry phones. To get an impression of the system, research has been conducted about the storage and encryption used by the BlackBerry phones. There are three storage types available:

- the BlackBerry Application storage: *This contains the operating system, JVM and an internal proprietary file system. It is refered as flash memory or onboard memory. It is the only place from which applications can be run.*
- built-in media storage: *This is an embedded Multi Media Card (eMMC) which contains a File Allocation Table (FAT) file system. It is also refered as internal media memory or onboard memory.*
- external memory card storage: *This is used to extend the the storage of the BlackBerry device. It consist of a removable microSD card which also contains the FAT file system.*

From this it is known that WhatsApp itself runs from the flash memory and that the database for the application is stored in the eMMC or the removable microSD card. In our case the latter was found to be true.

A more in depth look at the encryption was found on the BlackBerry Documentation[6] site. It has been studied, especially the area for the developers. For application developers there is a Crypto API available that can be used. It contains the packages you can use to perform tasks that involve encryption. Figure shows the content and packages within the API. With this API it is possible to use symmetric (private key) as well assymetric (public key) encryption. The API gives the means to:

- encrypt and decrypt data
- work with secure connections

[4] http://docs.blackberry.com/en/developers/deliverables/4526/BlackBerry_Java_Development_Environment-4.7.0-US.pdf
[5] http://docs.blackberry.com/en/admin/deliverables/25763/Two-factor_content_protection_865776_11.jsp
[6] http://docs.blackberry.com

- manage cryptographic keys
- digitally sign and verify data

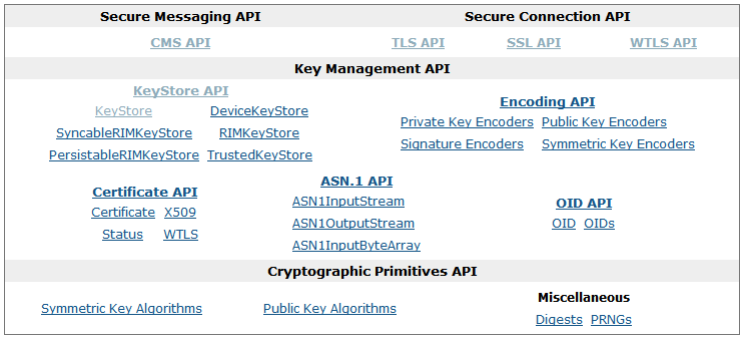

Figure 5.4: BlackBery Crypto API

The Crypto API consist of a ammount of packages which all have their own specifications. The most important ones for this project will be explained to a certain level.

The BlackBerry enviroment plays a large part in the encryption of the database. The BlackBerry operating system (OS) transparently encrypts files used by programs. This can include taken pictures and downloaded programs.

The BlackBerry has a feature called content protection. According to BlackBerry this works as follows[7]:

> *"Content protection can be used to encrypt data in string objects or byte arrays. Content protection can apply to data that is not persisted, but the Content Protection API contains specific functionality for the persistent store. Whenever an application attempts to encrypt an object, the unencrypted version of the object is marked with a special bit called a plaintext bit. Any object marked with a plaintext bit is presumed to contain unencrypted, sensitive data. An application specifies which data it considers to be sensitive by encrypting and decrypting objects, marking these objects with plaintext bits. Until an attempt has been made to encrypt data, the device assumes that that data is not sensitive. When the device is locked, the content protection framework uses these plaintext bits to ensure that as many plaintext objects as possible are erased from device memory."*

The documentation states that the phone employs Elliptic Curve Cryptography[8] (ECC) to encrypt its data.

5.4.4 BlackBerry Enterprise Server

There is a way to bypass the content protection on a BlackBerry device that is running BlackBerry Device Software v4.3 or later. This can be done by using a BlackBerry Enterprise Server v4.1 SP5 or later. The software uses a remote password reset cryptographic protocol to reset the device password when content protection is turned on. In this case the BlackBerry device will not prompt the user for the old device password[9]. The protocol is supposedly built in such a way as to make recovery of the key pair impossible. However this does provide a path to decrypt the data. However this would constitute an active attack.

However the device would have to have been previously known to the server. This means that if the BlackBerry device is a company provided phone, it could be unlocked by the company. In a criminal investigation this could verywell work, it would however need cooperation from a third party. It would certainly be preferable to be able to recover the data from a forensic data dump.

[7] http://docs.blackberry.com/en/developers/deliverables/21091/Content_protection_1554382_11.jsp
[8] http://en.wikipedia.org/wiki/Elliptic_curve_cryptography/
[9] http://docs.blackberry.com/en/admin/deliverables/16648/Resettng_a_BB_pswrd_when_contnt_prtctn_on_843329_11.jsp

Chapter 6

iPhone

6.1 Specifications

The iPhone phone used in this project is an Apple iPhone 3GS. The phone was released in 2009.

Brand:	Apple
Model:	iPhone 3GS
Serial Number:	88014UH2Y7H
IMEI:	012024006048363
CPU:	Samsung APL0298C05 600 MHz
OS:	iOS 4.1

Table 6.1: iPhone phone specifications

6.2 Preparation

The Apple iPhone 3GS belongs to from the University of Amsterdam. It has been used in serveral projects and was therefore equipped with a jailbroken[1] iOS 4.2. To revert this phone to it's factory settings the phone needed to be recovered to a stock iOS. This was done trough the Apple iPhone 3GS's recovery process. After the recovery process a Vodafone Prepaid SIM card with a data connection was inserted in the phone and WhatsApp was installed. With a private phone some messages has been sent to the iPhone 3GS. From the iPhone 3GS some messages has been sent back to the private phone. These messages were predefined and are shown in table 4.2 of chapter 4.

6.3 Physical dump

For this project a forensic approach was employed, this meant the information couldn't be retrieved directly from the phone. Therefor a physical dump of the phone has been made with the UFED Physical Analyser software, see table 6.2. From this physical dump we extracted the WhatsApp database file from the phone with the same software, see table 6.3.

File Name:	iPhone_3G_4.1_Physical_Extraction_24-11-11_07.28.14.img
File Extension:	IMG
File Size:	8.120.172.544 bytes
MD5 Hash:	091FD9EAF950597DB96ED0074A210033
SHA1 Hash:	41D66D961BA07177ABDEE31E88E3ED54C3CF5952

Table 6.2: iPhone physical dump specifications

[1] http://en.wikipedia.org/wiki/IOS_jailbreaking/

File Name:	ChatStorage.sqlite
File Extension:	SQLITE
File Size:	114.688 bytes
MD5 Hash:	6CAD93D35B0D9E920EDB2730F5F13A96
SHA1 Hash:	7C3050C4BEEC044DDD8992395428F05EFED7AD52

Table 6.3: iPhone database file specifications

6.4 Analysis

6.4.1 Database

The WhatsApp database file could be directly opened by SQLite Database Browser[2] and had no kind of encryption on it. This has to do with the fact that the UFED Physical Analyser software has a real-time decryption feature. This feature interprets encrypted data from various layers of the phone. The decryption is done on-the-file, obtaining access to files including application content such as databases[3]. So data encryption is available (see subsection 6.4.2 for more information), but the UFED Physical Analyser software is able to decrypt the data. This makes it unnecessary for this project to conduct further research on the encryption, because there is already software available that can extract the needed data in a forensicly correct manner.

6.4.2 Environment

The encryption on an iPhone device is performed by hardware encryption and is available on the iPhone devices since the iPhone 3GS. The hardware encryption is using AES 256-bit encoding[4]. The encryption can be enhanced by enabling data protection. This feature enhances the encryption by protecting the hardware encryption keys with your passcode[5]. Because of this hardware based encryption it can be stated that the WhatsApp database is kept reasonably safe for a typical user. For law enforcement agencies (and others) it is however possible to bypass the encryption with the use of the right forensic tools.

[2] http://sqlitebrowser.sourceforge.net
[3] http://www.cellebrite.com/forensic-products/forensic-products/ufed-physical-analyzer/iphone.html
[4] http://www.apple.com/iphone/business/docs/iPhone_Security.pdf
[5] http://support.apple.com/kb/HT4175/

Chapter 7

Differences

The previous chapters showed that there are a lot of differences between the Android, BlackBerry and iPhone operating systems in general and according to the WhatsApp database encryption. In table 7.1 these difference are shown in schematic way. Each subject in this table describes a subject from one of the research questions within this project.

	Android	**BlackBerry**	**iPhone**
General			
OS	Android	BlackBerry	iOS
Encryption			
Entropy	low	high	high
Algorith	AES	AES	AES
Key Size	192-bit	265-bit	256-bit
Key Storage	software package	OS	OS
Decryption			
Possible?	yes, by a created Python script	yes, by chip-off (very complex)	yes, by UFED Physical Analyser

Table 7.1: Comparison between the Android, BlackBerry and iPhone WhatsApp implementations

The comparison table shows that there is relation between these differences. The algorithm used on the different operating systems is especially AES. It's considered a secure algorithm for secret data[1] with the right implementation. The implementation of the algorithm determines the secureness of the system. That's where the Android phone fails. The key is stored within one of the classes in the software package and is generic for every Android. The BlackBerry and iPhone implementations are done through the operating system and store the key in a protected area. This is a much better implementation, because on a (default) BlackBerry or iPhone device you don't have access to this area. The Android operating system offers a similar way of storing keys, but it's not used. This makes the Android implementation a possible risk. This risk should be considered a medium level risk, because although the key has been comprimised you still need physical access to the device or its memory card. This shows that one aspect in the table can weaken the implementation of a very secure algorithm.

Eventually all the data is decryptable with the right knowledge of the application and operating system. The guys at Cellebrite did a good job with there latest version of UFED Physical Analyser, which is capable of creating a physical dump of an iPhone phone and is able the decrypt (decode) this information to readable data. The UFED Physical Analyser software is also capable of the decrypting (decoding) the chip-off (desoldered[2]) data from a BlackBerry phone, but for this process the chip has to be extracted from the device. This research project resulted in a way of decrypting a WhatsApp database on and Android phone with the created Python script. So with this result it is possible to use WhatsApp data for law enforcement agencies from phones with one of the main platforms.

[1] http://csrc.nist.gov/groups/ST/toolkit/documents/aes/CNSS15FS.pdf
[2] http://en.wikipedia.org/wiki/Desoldering/

Chapter 8

Conclusion

8.1 Introduction

The goal of this project was to research the possibility of reading the content of an encrypted WhatsApp database. Some of the goals were achieved while some were not, but stayed within the scope of the project. This chapter concludes the findings from this research project.

The main question was:
How can the WhatsApp databases be decrypted and the information within it be accessed?

From our main question, several sub-questions were derived. These were:

- How is the database encrypted and how strong is this encryption?
- How is the encryption key generated and where is it stored?
- What are the differences in encryption between the three platforms regarding the database?

The research has been performed on the following three platforms: Android, iPhone and Blackbery. Included in this approach are the different ways platforms take care of WhatsApp database encryption implementation. Because there is a distinctive difference in the way the platforms store the database the conclusions from this project are summarized accordingly.

8.2 Android

The WhatsApp implementation on the Android platform uses an AES 192-bit encryption for the database file. However the implementation isn't done in an error free manner. Storing the key in the software package isn't a secure solution. It has been shown that it is possible to extract the encryption key from the software package. A Python script has made it possible to decrypt a database file and access the data within.

8.3 BlackBerry

The BlackBerry environment plays a large part in the encryption of the database. The analysis of the physical dump of the microSD card showed a extremely high entropy, meaning strong encryption. The REM file uses an AES 256-bit encryption, but without the right key (device or password) the file can't be decrypted. In certain cases it is possible to export these encrypted files in the original unencrypted format, but only if the BlackBerry device is part of a BlackBerry Enterprise Server network. At this point in time it is only possible to decrypt the data of a BlackBerry device by soldering the chip from the device (chip-off).

8.4 iPhone

The iPhone platform uses hardware encryption. The hardware encryption is using AES 256-bit encryption and can be enhanced by enabling data protection. There is already software that can physically dump a full

iPhone and decrypt the content. This includes the WhatsAppp database which can be easily extracted from the physical dump. No further research was conducted.

8.5 Completion

As a final conclusion it can be said that the main question has been answered for all of the platforms. This project has made it possible to decrypt a WhatsApp database on an Android phone in a forensically sound manner. The decryption of BlackBerry data is only available through a chip-off process which is complex. So further research can be done to simplify this process. The iPhone didn't need any further research, because there is allready software that does decryption of an iPhone device.

This paper will hopefully be a good starting point for further research on the WhatsApp application and its forensic value.

Bibliography

[1] Yevgeniy Dodis and Adam Smith, *Theory of Cryptography*, vol. 3378 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, Reading, Massachusetts, 2005.

[2] Matt Sexton Karen Scarfone, Murugiah Souppaya, *Guide to Storage Encryption Technologies for End User Devices*, Number SP800-111. NIST National Institute of Standards and Technology, Gaithersburg, MD 20899-8930, nov 2007.

[3] Len Bass Paul Clements Rick Kazman, Gregory Abowd, "Scenario-based analysis of software architecture".

[4] Ophone, "The structure of android package (apk) files", Tech. Rep., 11 2010.

[5] James Hamilton and Sebastian Danicic, "An evaluation of current java bytecode decompilers", in *Ninth IEEE International Workshop on Source Code Analysis and Manipulation*, Edmonton, Alberta, Canada, 2009, vol. 0, pp. 129–136, IEEE Computer Society.