Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Image-based steganography using a mobile phone

Bachelor's thesis

2016
Askar Gafurov

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Image-based steganography using a mobile phone

Bachelor's thesis

Študijný program:	Computer Science
Študijný odbor:	2508 Computer Science
Školiace pracovisko:	Department of Computer Science
Školiteľ:	RNDr. Michal Forišek, PhD.

Bratislava, 2016
Askar Gafurov

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Askar Gafurov
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Image-based steganography using a mobile phone
Steganografia v obrázkoch pomocou mobilného zariadenia

Cieľ: Hlavným cieľom tejto práce je implementácia mobilnej aplikácie, ktorá by svojmu používateľovi umožnila skrývať krátke správy do obrázkov publikovaných na bežných webstránkach určených na ich zdieľanie. Dôraz pritom chceme klásť na nenájditeľnosť takýchto správ nepovolanými osobami, dobré kryptografické praktiky, dokumentáciu relevantných častí a prípadne aj na uveriteľné popretie úmyslu. Čiastočné ciele zahŕňajú, ale nie sú obmedzené, nasledovné:
1. Získať a spracovať prehľad vedeckých výsledkov súvisiacich s relevantnými časťami steganografie obrázkov.
2. Získať a spracovať prehľad existujúcich aplikácií príbuzných vyvíjanej, identifikovať ich silné a slabé stránky.
3. Analyzovať požiadavky kladené na aplikáciu a zostrojiť zoznam kľúčových požiadaviek a bezpečnostných rizík, ktoré je potrebné adresovať.
4. Navrhnúť, implementovať a zdokumentovať výslednú aplikáciu.

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 25.10.2015

Dátum schválenia: 27.10.2015 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....
študent vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Askar Gafurov
Study programme: Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study: Computer Science, Informatics
Type of Thesis: Bachelor´s thesis
Language of Thesis: English
Secondary language: Slovak

Title: Image-based steganography using a mobile phone

Aim: The main goal of this thesis is the implementation of an application for a mobile platform that will allow its users to encode short messages into images shared on popular image-sharing websites. Focus should be placed on undetectability of such messages, good cryptographic practices, proper documentation, and possibly plausible deniability. Partial goals include, but are not limited to, the following:
1. Producing a sufficient overview of related topics in image-based steganography.
2. Producing a sufficient overview of applications similar to the intended one, analyzing their advantages and disadvantages.
3. Performing a requirements engineering phase, identifying key requirements and security concerns.
4. Designing, developing, and documenting the resulting application.

Supervisor: RNDr. Michal Forišek, PhD.
Department: FMFI.KI - Department of Computer Science
Head of department: doc. RNDr. Daniel Olejár, PhD.

Assigned: 25.10.2015

Approved: 27.10.2015 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

....
Student Supervisor

Acknowledgment:

I'd like to thank:

my family,
for constantly pushing me out of my comfort zone,

my primary school teachers Nayla Galiyeva and Nina Medvedeva,
for inviting me into a beautiful world of Mathematics,

German Nurlygayanov,
for inviting me into a beautiful world of programming,

Konstantin Ermolin, Alexandr Dulyasov and Emil Vakhitov,
for teaching me the art of friendship,

doc. Dmitriy Kuznetzov,
for teaching me how to lose and how to win,

my secondary school teachers RNDr. Zuzana Frková, PhDr. Daniela Veselovská, PaedDr. Pavol Danko and RNDr. Martin Kollár, PhD.,
for helping me to acclimatize in Slovakia,

Trojsten,
for giving me new friends and supporting my interest in Math and Computer Science,

Michal Smolík, Jaroslav Petrucha, Vladimír Macko, Jakub Šafin and Kamila Součková,
for being my rivals and friends,

doc. RNDr. Radoslav Harman, PhD.,
for inviting me into a beautiful world of Statistics,

my supervisor RNDr. Michal Forišek, PhD.,
for ignoring my e-mails and thus contributing to my independence,

Adam Dej,
for providing his deep knowledge of software development,

Bc. Mária Božová and Nina Hronkovičová,
for supporting me in times of despair,

and many others, without whom this would not have been possible.

Abstrakt

V tejto práci sme sa venovali digitálnej steganografii (ukrývaniu správ) pomocou obrázkov a konkrétne steganografickým aplikáciám pre smartfóny s operačným systémom Android. Popísali sme momentálny stav poznania v tejto oblasti. Potom sme rozanalyzovali súčasné aplikácie a popísali sme ich chyby. Na základe týchto výsledkov sme navrhli požiadavky, ktoré by mali takéto aplikácie spĺňať na to, aby boli efektívne. Následne sme vyvinuli aplikáciu, ktorá danú špecifikáciu spĺňa. Pri tom sme vytvorili implementáciu nového steganografického algoritmu a kostru pre použitie nových steganografických metód.

Kľúčové slová: steganografia, Android, open-source, JPEG, komplementárne vkládanie

Abstract

In this paper we've addressed digital image-based steganography (message hiding) and specifically steganographic applications for Android smartphones. We've given an overview of modern steganographic and steganalytic methods and mistakes of currently available steganographic applications. With that in mind, we've deduced requirements to an application, that would help it to avoid these mistakes. Then we developed such application. As byproduct, we've brought free open-source Java implementation of complementary embedding algorithm, which is superior to currently widely used F5 algorithm.

Keywords: steganography, Android, open-source, JPEG, complementary embedding

Contents

Introduction **1**

1 Image-based steganography **2**

1.1 What steganography is? 2

1.2 Image-based steganography 3

1.2.1 JPEG compression algorithm 3

1.2.2 How to hide information in JPEG? 5

1.3 Present state of free and/or open-source programs 10

1.3.1 Common mistakes 10

2 Goal of this work **12**

2.1 Motivation 12

2.2 Functional requirements 12

2.3 Technical requirements 13

3 Theoretical aspects of the application **15**

3.1 Main processes 15

3.2 Used algorithms 15

3.2.1 Steganographic algorithm: Complementary embedding (CE) . . 16

3.2.2 Cryptographic algorithm: AES 20

3.2.3 Compression algorithm: GZIP 20

3.3 Resistance against additional compression 20

3.4 How to use the application correctly 21

4 Implementation **23**

4.1 How to build the application 23

4.2 Overview of the design 23

4.2.1 Embedding process 23

4.2.2 Extracting process 24

4.3 Algorithms' settings 24

4.4 Testing the steganographic algorithm 25

4.5 How to replace used algorithms 25

Summary **28**

List of Figures

1.1 Example of YCbCr color space (Public domain) (source: https://en.wikipedia.org/wiki/File:Barns_grand_tetons_YCbCr_separation.jpg) 4
1.2 DCT basis functions (Public domain) (source: https://en.wikipedia.org/wiki/File:Dctjpeg.png) 5

3.1 Main pipelines 16

Introduction

Steganography is "an art of secret writing". Its goal is to conceal the fact of communication. It is often compared to cryptography, which purpose is to conceal the information, carried by the message. Steganography is probably as old as written communication itself. In ancient times, people wrote their messages under layer of wax of wax tablets to sneak their messages into another city, or used invisible inks (e.g. milk) to write their notes.

In modern days, digital steganography has arisen. Messages are embedded into digital texts or multimedia (like images or music).

Digital steganography and steganalysis (art of revealing hidden messages) are young branches of modern science and new approaches are developed each year.

Despite its growing popularity, many applications are still using old methods, which have been proven to be unreliable. Applications for mobile phones, specifically, are very vulnerable because of usage of old algorithms and limited knowledge of their developers about modern state of the area.

In this paper, we will discuss modern methods of steganography and steganalysis, analyze the common mistakes of currently available applications (chapter 1). Then we will develop our own free open-source application for steganography. Firstly, we will specify functional and technical requirements on our application. Application would be created in a way that would encourage users to avoid mistakes and achieve maximum security and inconspicuousness (chapter 2). Secondly, we will discuss theoretical aspects of our application: describe used algorithms and enlist "good practices" that could help a user (chapter 3). Lastly, we will describe technical details of our application (chapter 4).

Chapter 1

Image-based steganography

In this chapter we will cover the term steganography and explain how can we measure its strength. Then we will proceed to image-based (specifically to JPEG-based) steganography. We will end this chapter with an overview of present state of free and/or open-source software for steganography on Android platform.

1.1 What steganography is?

Steganography is an art of hiding information. The difference between steganography and cryptography is that the goal of cryptography is to hide the content of message, whereas the goal of steganography is to conceal the very fact that the message exists.

A bit of terminology

Information, which we want to hide, is often called *secret message*. The object, in which we want to embed our secret message, is often called *cover message* (or, in our case, *cover image*). Additional information, required to detect and extract the secret message from cover object, is called *stegokey* (as opposed to cryptokey, which is used to unveil the content of the message). If the stegokey, needed to embed a message is different from the key, needed to extract it, we call this steganographic method *asymmetric*. In other case, we call it *symmetric*.

How to measure strength of steganographic methods?

Basic idea is simple: if a message concealed by method X was not discovered, then it's a good method. But the concealing power depends on the message length (analogy: it is relatively easy to write one date on your school desk without having your professor seeing it, but it's a lot harder to write a whole textbook on it).

Keeping that in mind, one can never tell with 100% guarantee that this object doesn't carry any message, as sometimes it could carry only a few bits.

As for embedded messages are often encrypted and the order of bits is permuted, it is hard to actually extract the message. So it is reasonable only talk about the possibility to distinguish between "clean" images and images with hidden messages.

1.2 Image-based steganography

In this section we will talk about how to hide information in digital images.

Back in younger days of steganography, lossless images were actively used as cover images. Algorithms were based on embedding the secret bits into least significant bits (LSB) of pixels with minor alternations. This approach was heavily criticized by Westfeld and Pfitzmann in [1].

In modern era, when vast majority of images are encoded in JPEG format, usage of lossless images is at least questionable. So, in order to avoid suspicion, one has to use JPEG files.

1.2.1 JPEG compression algorithm

JPEG format provides many options for storing images. We will talk about one of the most common versions. For further details we recommend to read [2].

Input of compression algorithm is a raw image with pixels encoded in RGB model (each pixel is encoded as three 8-bit numbers, representing red, green and blue component of resulting colour, resulting in 24 bits of information per pixel).

Compression consists of several steps:

1. Transformation from RGB to YCbCr color space
2. Reduction of Cb and Cr channels definition (lossy operation)
3. Splitting each channel into 8×8 bytes long blocks
4. Discrete cosine transformation (DCT) of numbers in each block
5. Rounding of DCT coefficients (main lossy operation)
6. (Lossless) compression of coefficients (main compression operation)

Color space transformation

Pixels are transformed from RGB to YCbCr color space. Y channel represents brightness of a pixel (which can be viewed as grayscale copy of the original image), Cb and Cr components represent so called *chrominance* – color without brightness. You can see example in figure 1.1.

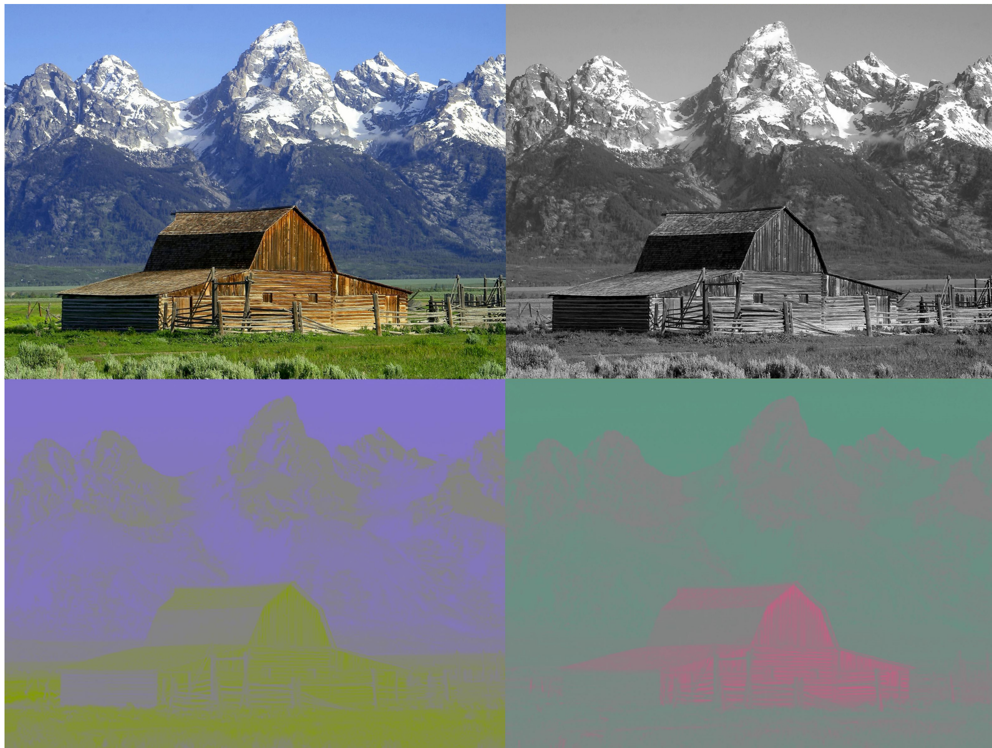

Figure 1.1: Example of YCbCr color space. Left upper corner is an original image, right upper corner — Y channel (grayscale version of image), left lower — Cb channel, right lower — Cr channel.

Downsampling

Human eye is more sensitive to small changes of brightness rather than small changes in the hue and color saturation. Using that fact, we can use less space to represent Cr and Cb channels with almost no visible change to an image.

There are many options, but one of the most used is to use 4 bits instead of 8 to represent Cr and Cb and 8 bits to represent Y channel (so Y channel is left unchanged).

Splitting into blocks

Each channel is divided into 8×8 bytes long blocks and then processed separately. If channel is not exact multiple of block size, then incomplete blocks are filled with fake values (there are several methods to construct them, but details are irrelevant for our purposes).

Discrete cosine transformation

Discrete cosine transformation is a way of representing block of pixels as a linear combination of so called *basis functions*, showed on figure 1.2, so we can store coefficients of linear combination instead of the original values.

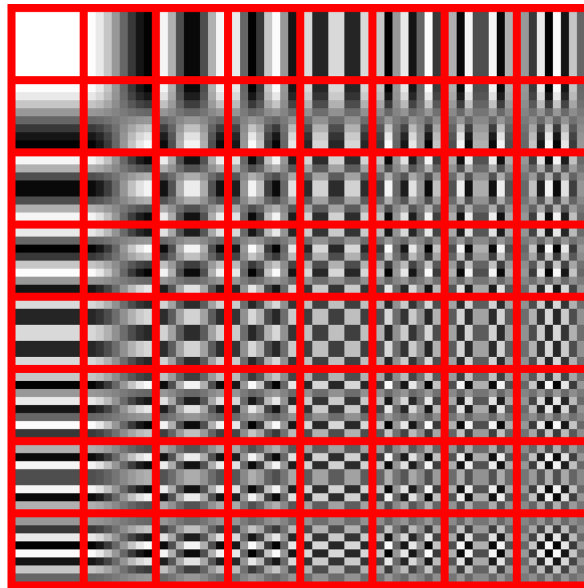

Figure 1.2: DCT basis functions.

Rounding of DCT coefficients

Here we use two facts. First, DCT basis functions varies from simple to very complex. Second, human eye is less capable to distinguish such fine local changes. Therefore, we can represent coefficients of complex basis functions with less definition (in other words, quantize them).

The level of quantization is defined by demanded quality ratio.

The most fine quality $Q = 100\%$ let all coefficients be rounded to whole numbers, so it still would be lossy operation.

Lossless compression of DCT coefficients

Another interesting property of DCT is that the coefficients of last basis functions are often very small (because DCT "tries" to express image with the first sinusoids). Therefore, after quantization process many of DCT coefficients could be zero.

As could be expected, long sequences of zeros could be efficiently compressed, and therefore significant compress ratio could be achieved.

1.2.2 How to hide information in JPEG?

An image encoded in JPEG format could be seen as a bunch of (quantized) DCT coefficients, which are just regular integer numbers. One of the most natural ideas is to change least significant bits (LSB) of these numbers to store our message, as these changes are mostly invisible for human eye and the message can be extracted without knowledge of the original image.

Problems with direct application of this method is that coefficients with values 0 or 1 would be changed drastically and it would create great disproportion in amount of zeros and ones.

Modern algorithms try to fight with statistical approach by devising ways to keep intact various statistical properties of resulting numbers.

JSteg library (direct LSB)

This library embed secret message in least significant bits of DCT coefficients sequentially.

This approach was quite common for a period of time (there were even algorithms that performed such embedding in lossless images), but in year 1999 Westfeld and Pfitzmann came up with a so called *chi-square attack*.

Chi-square attack

As you can see in the original paper [1], this attack was initially pointed against algorithms with lossless cover images, but this approach can be applied for JPEG images too.

Westfeld and Pfitzmann pointed out, that relative amount of values of the DCT coefficients (or pixels in the original paper), which differ only in their LSB (so called *paired values*, or PoV), is mostly very far from equal (i.e. there it is more probable that there would be 1000 coefficients with value 30 and 4000 coefficients with value 31 than that there would 2500 of each).

Assuming that 0 and 1 bits are distributed uniformly in the secret message (which is usually the case for compressed and/or encrypted message), relative amount of PoVs becomes equal. And assuming sequential writing in DCT coefficients, one can test whether paired values show such behavior. The corresponding statistics has chi-square distribution, from which this attack got its name.

Let there be M different PoV in the image (e.g. one pair could be 20-21). Let $N_{k,0}$ ($k \in \{1, \dots, M\}$) denote the amount of coefficients of k-th PoV with 0 in the least significant bit. Analogically, $N_{k,1}$ would denote the amount of coefficients with 1 in the least significant bit. Let denote $N_k := N_{k,0} + N_{k,1}$. It is important that the LSB steganography does not change any of N_k.

Let our null hypothesis be that an encrypted message (with uniform distribution of 0 and 1) was embedded into these coefficients (and alternative hypothesis would be the opposite sentence). Assuming the null hypothesis, we can see $N_{k,0}$ as a random variable following binomial distribution with $n = N_k$ and $p = \frac{1}{2}$, i.e.

$$N_{k,0} \sim Bin\left(N_k, \frac{1}{2}\right)$$

This distribution can be approximated with normal distribution, i.e.

$$N_{k,0} \dot{\sim} N\left(\frac{N_k}{2}, \frac{N_k}{4}\right)$$

We can then normalize that variable to $N(0,1)$ distribution, i.e.

$$\tilde{N}_{k,0} := \frac{N_{k,0} - \frac{N_k}{2}}{\sqrt{\frac{N_k}{4}}} \dot{\sim} N(0,1)$$

Then the sum of squares of such random variables $\tilde{N}_{k,0}$ would approximately follow the χ^2 distribution with $M-1$ degrees of freedom (we subtract one because the sum of all variables is fixed), i.e.

$$\sum_{k=1}^{M} \tilde{N}_{k,0}^2 \dot{\sim} \chi^2_{M-1}$$

We can simplify the expression a bit to get the final statistics:

$$\sum_{k=1}^{M} \tilde{N}_{k,0}^2 = \sum_{k=1}^{M} \left(\frac{N_{k,0} - \frac{N_k}{2}}{\sqrt{\frac{N_k}{4}}}\right)^2 = \sum_{k=1}^{M} \frac{\left(N_{k,0} - \frac{N_k}{2}\right)^2}{\frac{N_k}{4}} = \sum_{k=1}^{M} \frac{\left(2N_{k,0} - N_k\right)^2}{N_k} =: T$$

Now we can construct a test for our hypothesis. We can do it, for example, by evaluating the $(1-\alpha)\%$ confidence interval (α is usually chosen as 5%) for χ^2 distribution, which is equal to $\left(\chi^2_{M-1}\left(\frac{\alpha}{2}\right), \chi^2_{M-1}\left(1-\frac{\alpha}{2}\right)\right)$, where $\chi^2_{M-1}(x)$ is a x-th quantize of this distribution. (e.g. for $\alpha = 5\%$ and $M = 100$ this interval is equal to $(73.36108, 128.422)$).

The test then work in following way: if evaluated T statistics is out of the confidence interval, then the null hypothesis is rejected, i.e. it is improbable that there is a hidden message.

This test is used for estimating the length of a hidden message in following way: the null hypothesis is tested for every prefix of DCT coefficients array. Assuming sequential writing, the test should justify the null hypothesis as long as the message is longer than the prefix (i.e. the secret message has smoothed the differences between PoVs), and reject it as long as the message is very short compared to the prefix length (i.e. the secret message hasn't changes a lot of PoVs).

This attack could be extended to random writing (i.e. that DCT coefficients to be used are chosen randomly) by using shifting windows instead of prefixes (so called *extended chi-square attack*). This approach is described in the paper [3].

F5 algorithm

In order to defend against chi-square attack, Westfeld and Pfitzmann proposed to use subtraction operation to match the secret bit instead of the replacing the last bit. In

this way, the PoVs (described in previous subsection) won't be so drastically changed. This approach was applied in Westfeld's algorithm F5 (described in paper [4]).

The algorithm is a bit complex, and so it was originally described as a series of algorithms (thus the number 5) with gradual removal of the flaws.

F3 algorithm decreases the absolute value of the coefficients to match the secret bits instead of their replacement, thus being resistant against chi-square attack. Because of this approach, it skips zero coefficients. The problem is that if the original coefficient has the absolute value 1 and the secret bit is 0, then after the embedding the coefficient would be 0. The receiver cannot distinguish between unused and decremented zeros, so algorithm has to embed this bit in the next coefficient available.

This approach leads to two problems: F3 created more zeros than ones in the LSB of coefficients, creating the peculiar statistically detectable histogram, and that the original JPEG coefficients contain more odd than even values, and so the *unchanged* part of a message contains more ones than zeros in LSBs.

F4 solves these problems by inverting the meaning of negative coefficients, so that odd negative coefficient means 0 and even negative means 1.

F4 and F3 both suffer from sequential using of the DCT coefficients. F5 removes this weakness by choosing the coefficients in a random way (based on a steganographic key). Additionally, F5 uses so called *matrix encoding* to increase effectiveness of embedding ratio (bits of information per modified coefficient).

F5 algorithm is currently used in many steganographic Android apps, mainly because of its open Java implementation. However, this algorithm can be successfully attacked with so called *S attack*.

S-family attacks

S-family is a general framework of attacks. The basic idea is that one can find (empirically) some specific macroscopic characteristic $S(p)$ of images, which changes predictably (e.g. monotonically increases) with the length of the embedded secret message p (i.e. this function is specific for every algorithm). One can then estimate critical of this function S (i.e. $S(0)$ — characteristic for an empty embedded message and $S(max)$ — for the maximal possible message length). Then he has to solve the equation $S(q) = x$, where x is a given image with embedded message and q is an estimation of this embedded message. $S(max)$ can be obtained by embedding of maximum possible message to obtained cover image (so it will rewrite the original hidden message), and $S(0)$ can be obtained by cropping the cover image and re compressing with the original quantization table. You can read more about general structure here [5].

For F5 algorithm, the S was chosen as "blockiness" of the image. The process is described in [6].

Complementary embedding algorithm (CE)

This algorithms was designed to be resistant against both S and chi-square attacks designed for F5 and JSteg.

Because it's the algorithm we had used in our application, we've decided to put the description of this algorithm in chapter 3 along with the descriptions of other used algorithms (you can find it in the subsection 3.2.1).

Attacks based on Benford's law

This approach was described in [7] and [8].

Benford's law states that in a random set of integer numbers the most significant digit would with the highest probability be 1, second highest would be 2, and so on. Formally, if we denote x as the most significant digit, the law could be written as this:

$$P(x = 1) > P(x = 2) > P(x = 3) > \ldots > P(x = 9),$$

where $P(x = i)$ is probability that the most significant digit is equal to i.

This law was given a numerical form:

$$p(n) = \log_{10}\left(n + \frac{1}{n}\right) \quad (i \in \{1, \ldots, 9\}),$$

where $p(n)$ denotes the probability that the most significant digit is equal to n.

This law is being used to detect elections riggings (usually authors use the Benford's law modified to describe second digit distribution). You can read more in these papers [9–11].

Fu et al. in [12] studied distribution of quantized DCT coefficients (specifically, the distribution of first digits) and proposed a Benford-like distribution (which they called *general Benford's law*):

$$p(n) = N \cdot \log_{10}\left(n + \frac{1}{s + n^q}\right) \quad (i \in \{1, \ldots, 9\}),$$

where N, s and q are parameters dependent on used quality ratio (note that the original form can be obtained when $N = 1$, $s = 0$ and $q = 1$).

The algorithm is easy: we need a precomputed table of parameters for each quality ratio. Then we extract DCT coefficients from a cover image, compare their first digits' distribution, compare to the expected distribution and decide whether to declare the image as "suspicious". Details can be found in the original paper [7].

This method is very interesting because of seeming effectiveness: authors claim they can detect even a 1KB of embedded data in the image of size 800×600.

Attacks using higher order statistics

While many steganographic algorithms aim to preserve first-order statistics of cover images (i.e. histograms of coefficients), higher order statistics (variance, skewness and kurtosis) appear to be harder to preserve.

Algorithm, proposed by Siwei Lyu and Hany Farid in [13], exploits this fact. Firstly, it decomposes a cover image by using separable quadrature mirror filters (QMFs) [14]. This filter essentially separates the image into several subimages by their frequencies (frequencies are meant as patterns, where more complex structures are referred to as *higher frequencies*). Secondly, it evaluates corresponding statistics (mean value, variance, skewness and kurtosis, mentioned earlier). Thirdly, it obtains additional information from errors of optimal linear prediction of coefficients magnitude. Fourthly, it evaluates the same 4 statistics of these data.

This process is applied on a training set of images, both with hidden messages and without. Then algorithm applies a general classification method (originally authors had used Fisher linear discrimination analysis [15], now they use support vector machines (SVM) [16]). After training, the model is ready to detect images with hidden messages.

Main drawback is a need of a representative training set.

1.3 Present state of free and/or open-source programs

In this section we will show common mistakes of available applications for Android.

We've analyzed several free applications in Google Play [17–35] and enlisted mistakes we've met.

1.3.1 Common mistakes

Closed implementation

Steganography and cryptography could be very complicated to implement correctly, so applications with no open code should always be treated with a grain of salt [17–34] .

Unknown steganographic and/or cryptographic algorithms

Digital image steganography is a very young field and new algorithms and attacks appear every year. In this context it is highly inadvisable to use old applications or applications with no such specification [17–34] .

No limitation for message length

As there are blind steganalytic methods, which does not depend on used steganographic algorithm (such as using higher order statistics or Benford's law, described in

section 1.2.2), it is inadvisable to send large messages [17–34] .

Low upper limit for a key

Some apps has upper limit for a key set as 8 symbols, which is considered low by any standard [17] .

Lossless images used as carrier

It is highly inadvisable to use lossless image formats as cover images. We've discussed this matter in section 1.2 [19, 20] .

No support of non-Latin symbols

Some applications has only ASCII support [20, 33] .

Possibility to use same images repeatedly

It is highly inadvisable to use the same image twice as a carrier, let alone having a predefined set of images [17–20, 23–35] .

Steganographic algorithm is not (mandatorily) key-dependent

Key less algorithms are very vulnerable to attacks (it would suffice in that case to use the original application to detect hidden message) [19–21, 23, 25, 27, 30, 32, 33, 35] .

No encryption nor compression

Most of steganographic algorithm work under the assumption that a message has uniform distribution of 0 and 1 (which is a case for encrypted message) [17, 20, 21, 23, 27]
.

Chapter 2

Goal of this work

In this chapter we will specify requirements on our application.

2.1 Motivation

Our initial motivation was to create an application for communication between people in countries with censored Internet (e.g. Syria, Egypt), with only limited communication channels available (e.g. e-mails, Facebook, ICQ, etc.). Particularly we are interested in creating a way to communicate with outer world securely and inconspicuously (e.g. correspondents).

We have chosen digital images as our communication medium because it is only natural for a mobile phone to have plenty of almost useless photos (e.g. photos of someone's dog or dinner) and these photos are being sent through various social media all the time.

In chapter 1.2 we discussed about different formats for this purpose, and we've chosen *JPEG* as the most suitable.

2.2 Functional requirements

In this section we will discuss *which* tasks our application has to be able to perform.

Embedding and extracting of messages into images

Main function is to embed text message into JPEG files from the phone's camera using steganography.

The steganographic algorithm must be key-based, as for in the opposite case it would be very for an attacker to extract the hidden message (you can read more on this in [36]).

The application should forbid to use images from phone's memory, for it could lead to multiple using of the same image, which is *very bad* for user's inconspicuousness.

Message encryption

The application has to be able to encrypt and decrypt the message. This also contributes to undetectability of the secret message, as the result of extracting still would be indistinguishable from a random noise.

We also have had to decide whether we will use symmetric or asymmetric encryption. We decided to stick with symmetric, as it has easier key management. Also, if a user would like to send a message to many people, he would have to send the same message many times with different keys, whether embedded in one image (which would make the detection easier) or in many images (which would also increase suspicion).

Message compression

The secret message has to compressed in order to take minimum possible space in medium, as bigger changes in medium are easier to detect.

Key management

Our application has to be able to generate secure keys for steganographic and cryptographic algorithms. It is also should be possible to share a key with someone else (in order to communicate).

Whenever an application has to deal with keys, it is a good idea to conceal them and reveal them only by entering a so called *master key*. We decided not to demand this feature, as if a user and his mobile phone are already in hands of the censors, it is only a matter of time when he would give up his master key. The whole purpose of our application is to avoid the contact with the censors.

Optimal conditions of undetectability

Our application has to be able to decide whether it is sufficiently secure to send a message of given length.

2.3 Technical requirements

In this section we will discuss requirements on *how* our application has to perform its tasks.

Support of UTF-8

Our application is meant to be used all around the world, and not all languages use Latin alphabet to encode it (e.g. Arabic, Russian).

Interchangeability of algorithms

As steganography and steganalysis develop very quickly, our application has to have the ability to easily change used algorithms.

Openness of the code

No one can trust an application that has closed source code (not only because this code could be harmful, but also because no one can check whether all methods were implemented correctly).

Ability to withstand additional compression

Many social media are known to compress any images that are sent through their systems. We'd like to analyze whether it is possible to save the information in the image instead of compression (e.g. by using error correction encoding).

Chapter 3

Theoretical aspects of the application

In this chapter we will talk about a structure of the main processes (embedding and extracting messages), describe used algorithms (with an accent on steganography) and overview various possible communication channels (carriers). We will also talk briefly about resistance of used algorithms against additional compression of image with embedded message. We will end this chapter with a list of "good practices" a user should follow to achieve maximum possible security and undetectability.

3.1 Main processes

In this section we will describe the pipelines of embedding and extraction.

Embedding works in the following way: we start with obtaining the cover image (from a camera) and the secret message we want to send. Firstly we compress the secret message. Secondly, we encrypt it with crypto key. Finally, we embed this message into the cover image using steganographic algorithm with stego key.

Extraction works in reverse order: we start with extracting embedded bytes from the cover image (using stego key). Then we decrypt the message using crypto key and finally we decompress it.

Both pipelines are shown in the figure 3.1.

3.2 Used algorithms

In this section we will describe algorithms that we used in our application and discuss their suitability for our purpose. We have had to find a balance between potential quality of the algorithms and the cost of their implementation and usage.

We'd like to mention that these algorithms are not the product of our work.

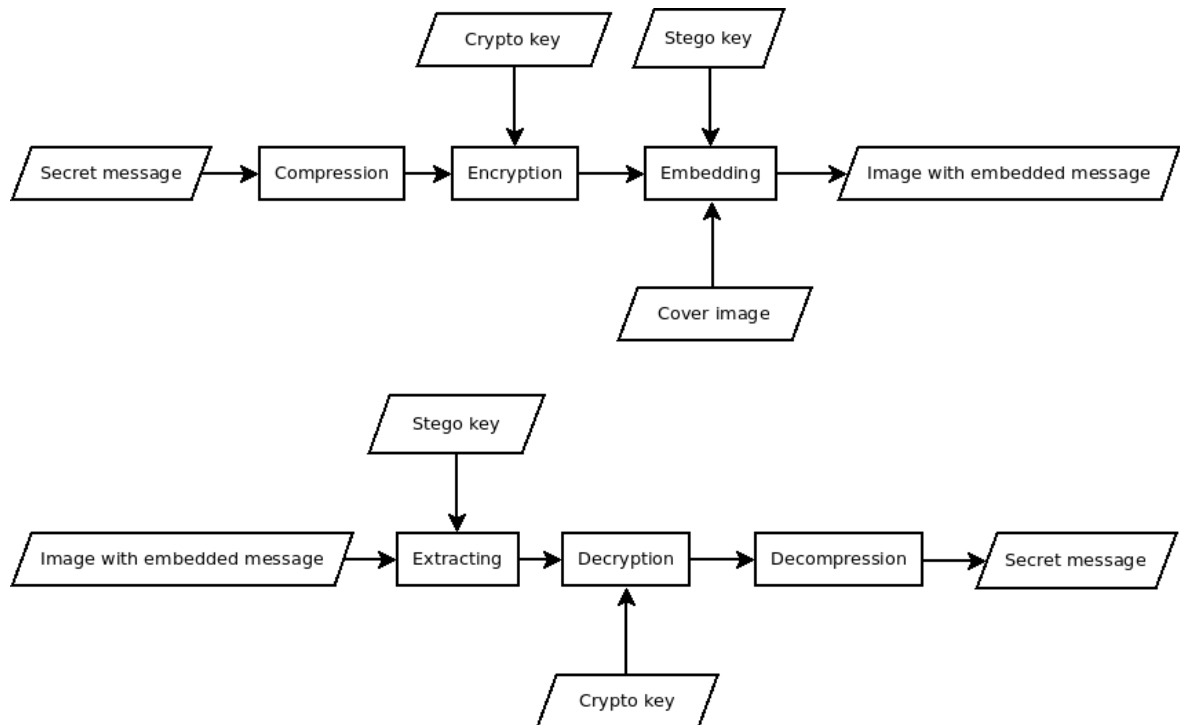

Figure 3.1: Main pipelines. Upper part: embedding process, lower part: extracting process

3.2.1 Steganographic algorithm: Complementary embedding (CE)

This algorithm was proposed by Chiang-Lung Liu and Shiang-Rong Liao in [37]. We've chosen this algorithm as it has better steganographic properties than the "unofficial standard" algorithm F5 and is relatively simple to implement and to verify its correctness.

This algorithm was primarily developed to withstand S-attacks (as opposed to F5 algorithm). It achieves this goal by dividing both message and DCT coefficients into two parts (according to steganographic key and predefined separation ratio α) and modifying coefficients in the first part by subtracting and in the second part by addition.

Embedding process

Input for the algorithm:

- K steganographic key (seed for random generator)
- B bytes to be embedded (secret message, encrypted and compressed)
- α separation ratio
- β adjustment parameter

- D quantized DCT coefficients sequence

Output of the algorithm:

- D' changed quantized DCT coefficients

Step 1: Use a stego-key K to create a permutation of quantized coefficients D. That is,

$$Q := P_K(D),$$

where $P(\cdot)$ is a key-dependent permutation function and Q denotes the permuted coefficient sequence.

Step 2: Divide Q into two parts, Q_1 and Q_2, according to separation ratio α. That is,

$$Q_1 := Q[0 : \alpha L(Q)],\ Q_2 := Q[\alpha L(Q) : L(Q)],$$

where $L(\cdot)$ denotes the length and $Q[a : b]$ denotes a slice of an array from a-th to b-th element (half-open interval, i.e. b-th element does not belong into slice).

Step 3: Separate the secret message bytes sequence B into to two parts B_1 and B_2, according to the separation ratio α (similarly to previous step).

Step 4: Let L_1 and L_2 denote the byte representations of $L(B_1)$ and $L(B_2)$, respectively. Let M_1 be the concatenation of the L_1 and B_1 ($M_1 := L_1||M_1$). Similarly, let $M_2 := L_2||B_2$. That would be our two parts we will embed into the cover image.

Step 5: Convert byte sequences M_1 and M_2 to bit sequences M_1' and M_2' (by using little endian encoding).

Step 6: Embed M_1' into the non-zero coefficients of Q_1 using algorithm E1. Note that the adjustment parameter β is used here to make it more resistant against statistical attacks.

Step 7: Embed M_2' into the non-zero coefficients of Q_2 using algorithm E2.

Step 8: Combine Q_1 and Q_2 to form a single coefficient sequence Q'.

Step 9: Using stego-key K, depermute the coefficient sequence Q' into D'. That is,

$$D' := P_K^{-1}(Q'),$$

where P^{-1} denotes the inverse permutation.

Step 10: Return D'.

Algorithms E1 and E2 are essentially just for-loops through all bits of the secret message and coefficients with a complex if-statement inside, describing how to change coefficients (changes of coefficients depend on oddness of the coefficient and value of the secret bit). Detailed pseudocode can be found in the original paper [37].

Here we show our implementation of these algorithms' if-statements in a single function (in Java programming language). Variable *type* denotes whether we are using E1 or E2 algorithm.

```
public int embedBitUnsafe (final int type, final int c, final int bit) {
    // type: 1 for E1, 2 for E2 algorithm
    // c: (non-zero) coefficient we are changing
    // bit: secret bit to embed
    if (type == 1) {
        if (c > 0 && isOdd(c)) {
            if (bit == 0 && c-1 == 0) return c-2;
            else if (bit == 0 && c-1 != 0) return c-1;
            else if (bit == 1) return c;
        }
        else if (c > 0 && isEven(c)) {
            if (bit == 1) return c-1;
            else if (bit == 0) return c;
        }
        else if (c < 0 && isOdd(c)) {
            if (bit == 1) return c-1;
            else if (bit == 0) return c;
        }
        else if (c < 0 && isEven(c)) {
            if (bit == 0) return c-1;
            // in paper the condition is (c == 1), but it doesn't make much sense
            else if (bit == 1) return c;
        }
    }
    else if (type == 2) {
        if (c > 0 && isOdd(c)) {
            if (bit == 1) return c+1;
            else if (bit == 0) return c;
        }
        else if (c > 0 && isEven(c)) {
            if (bit == 0) return c+1;
            else if (bit == 1) return c;
        }
        else if (c < 0 && isOdd(c)) {
            if (bit == 0 && c + 1 == 0) return c+2;
            else if (bit == 0 && c + 1 != 0) return c+1;
            else if (bit == 1) return c;
        }
        else if (c < 0 && isEven(c)) {
            if (bit == 1) return c+1;
            else if (bit == 0) return c;
        }
    }
}
```

Algorithm E1 additionally change a part of used coefficients using the adjustment parameter β. The adjustment is simple: we need to look at first $\beta L(M)$ coefficients

and change values -2 to value 1.

Extraction process

Input for the algorithm:

- K steganographic key (seed for random generator)
- α separation ratio
- D quantized DCT coefficients sequence with embedded message

Output of the algorithm:

- B extracted bytes of the secret message

Step 1: Use a stego-key K to create a permutation of quantized coefficients D. That is,

$$Q := P_K(D),$$

where $P(\cdot)$ is a key-dependent permutation function and Q denotes the permuted coefficient sequence.

Step 2: Divide Q into two parts, Q_1 and Q_2, according to separation ratio α. That is,

$$Q_1 := Q[0 : \alpha L(Q)],\ Q_2 := Q[\alpha L(Q) : L(Q)],$$

where $L(\cdot)$ denotes the length and $Q[a : b]$ denotes a slice of an array from a-th to b-th element (half-open interval, i.e. b-th element does not belong into slice).

Step 3: Extract the length L_1 of the embedded message from the Q_1, then extract $8L_1$ bits of the embedded message B'_1 from Q_1 by using the following algorithm:

```
if (c > 0 && isEven(c)) return 0;
if (c < 0 && isOdd(c))  return 0;
if (c > 0 && isOdd(c))  return 1;
if (c < 0 && isEven(c)) return 1;
```

Step 4: Similarly, extract the length L_2 of the embedded message from Q_2 and $8L_2$ bits of the embedded message B'_2 by using the following algorithm:

```
if (c > 0 && isOdd(c))  return 0;
if (c < 0 && isEven(c)) return 0;
if (c > 0 && isEven(c)) return 1;
if (c < 0 && isOdd(c))  return 1;
```

Step 5: Convert bit sequences of the secret messages B'_1 and B'_2 to byte sequences B_1 and B_2.

Step 6: Combine B_1 and B_2 into single secret message B.

Step 7: Return B.

Tuning the parameters α and β

The original paper suggests that the separation ratio α should be chosen from a segment $\left[\frac{3}{4}, \frac{5}{6}\right]$ and the adjustment parameter β should be chosen from a segment $\left[\frac{1}{8}, \frac{1}{3}\right]$ (bigger β positively correlates with better undetectability of the message). In tests the authors have chosen values $\frac{4}{5}$ and $\frac{1}{3}$, respectively.

Limit for a message length

The authors of the algorithm implies that for chosen values of the parameters it is safe enough to use even 50% of nonzero coefficients, i.e. the algorithm is still resistant to chi-square and S families of attacks. As for general measures of changing of the cover image, authors tell us that it is safe enough to use all nonzero coefficients available.

3.2.2 Cryptographic algorithm: AES

We are using AES-256-CBC algorithm for encryption. It is currently considered as a standard solution for symmetric encryption. The algorithm is described here [38].

One of the main problems of cryptography is a big chance to use it incorrectly. Therefore, we've decided to use an open implementation of it, provided by the package `com.scottyab.aescrypt.AESCrypt`.

3.2.3 Compression algorithm: GZIP

We are using GZIP algorithm mainly because it has been implemented in standard Java library as `java.util.zip.GZIPInputStream` and `java.util.zip.GZIPOutputStream` classes. This compression is based on widely used algorithm DEFLATE, which is based on Huffman encoding and LZ77 algorithm. You can read more about this in [39] and [40].

3.3 Resistance against additional compression

In this section we will shortly describe the possibilities of compression-resistant steganography.

Recompression of a JPEG image has a lot of parameters. Main are the size of an image and the quality ratio (described in section 1.2.1), but there is a plenty of secondary ones, like encoding the pixels in RGB or in YCrCb, rounding of Cb and Cr channels (8 bytes or 4 bytes), and so on.

Good news are that the most of these factors can be dealt with by simply choosing a cover image with the same parameters as the result of the additional compression (assuming these parameters aren't going to change).

The are two problematic ones: the size and the quality ratio, as they could be change from device to device (e.g. Facebook Messenger generally shows images in better quality for Apple smartphones that Android ones).

We can deal with the size of an image by choosing the one the communication channel would use for the given ends of a channel (i.e. the two given mobile phones). Otherwise, this issue is quite hard to reliably solve, as changing of the size would result in merging the adjacent blocks of an image, erasing the values of last bits of the DCT coefficients, ruining any DCT-based steganography.

We can, in theory, deal with the change of quality ratio by using error-correction codes. The main issue is that lowering of the quality would result in nullifying the coefficients, thus shifting the space the algorithm would search for a hidden message. It can be seen as the erasing of message symbols in general coding theory. It could be dealt with by permuting the whole blocks, not the individual coefficients.

Another issue is that the error-correction would result in longer message and, therefore, higher change of detection.

In our application, we decided not to implement the compression-resistant algorithm.

3.4 How to use the application correctly

In this section we will describe common mistakes and give several recommendations on how to achieve maximum inconspicuousness.

Never use the same image twice

Imagine a situation: the censor sees two visually identical message with little differences in their digital representation. He may make only one conclusion: the sender tries to manipulate the images to send some information. Therefore it is highly inadvisable to use similar images to steganography.

Never send the same message twice

Imagine another situation: the censor sees two images, compare their DCT coefficients and sees the the pattern in their last bits (remember, the secret bits are embedded into the same positions). There is only one natural conclusion: the sender uses the steganography.

Use images with high number of colors

Also avoid computer art, images with semantic context (like texts).

Change your keys frequently

Firstly, the keys have tendency to be leaked. Secondly, with enough tapped communication, keys can be deduced with prior knowledge. Thirdly, modern stochastic statistical approaches (neural networks, nonlinear programming, etc.) could theoretically distinguish images with embedded messages from "clean" even without any knowledge of the algorithm, not mentioning the keys. Fixing the key (i.e. fixing the positions of the coefficients to be changed) would only help the attackers.

Generate your secret keys by app's utilities, not by hand

Another general advice: small human-generated passwords are easier to break (dictionary attack, brute force, etc.). All basic recommendations for choosing a password are applicable here: long passwords with nonstandard symbols (numbers, special symbols, uppercase, etc.) are generally better.

We recommend to use standard in-app utility to generate the keys.

If you decide to generate the password by yourself, we recommend to stay in a set of ASCII characters for simpler key distribution (although keys are stored as UTF-8 strings, so you can potentially use even non-Latin symbols (Japan hieroglyphs, Arabic symbols, Cyrillic, etc.)).

Use only secure channels to distribute your keys

The best option is to physically meet with your correspondent.

Try to write your messages as short as possible

This rule has a simple explanation: shorter message means less changed coefficients, thus lesser distortion in statistical properties of an image, therefore better undetectability. We recommend to send messages with length less than 1KB (based on claimed effectiveness of steganalytic methods).

Use only lossless communication channels

As described in the section 3.3, additional compression could and probably would destroy the hidden message. There are many lossless channels: e-mail, Dropbox, Google Drive, Google Photos, Telegram messenger, Skype, etc.

Chapter 4

Implementation

In this chapter we will talk about main implementation details of our application.

4.1 How to build the application

In this section we will show how to build and install our application.

The process is very simple: you have to download all files and open them in Android Studio. Some of the code is written in C++, so you'd need to use NDK if you want to build the application without Android Studio.

The application has been developed and tested on Android OS 4.4 .

4.2 Overview of the design

In this section we will show the general architecture of the application and describe the main parts.

Our goal was to separate the interface and internal logic. We've achieved that by creating "junction" class `Solution`, which has been inserted between Android-specific interface code and steganographic core, so that code, responsible for the interface, is working only with this layer.

Now we will walk through all major functions of our application and describe what is going on inside.

4.2.1 Embedding process

An user types a secret message he wants to embed into an image. Then he takes a photo with phone's camera. Then he clicks a button "Start embedding". The program checks, whether an image is accessible, and then calls function `Solution.embed()`.

`Solution` object decodes a string with the secret message into a byte array, compress it, encrypts it and then calls a function `info.guardianproject.f5android.Embed()`,

which we borrowed from another open-source steganographic application *Pixelknot* by Guardian Project (mainly the correct implementation of encoding/decoding images into and from the JPEG format). This function extract DCT coefficients from the taken image, applies our steganographic algorithm, implemented as `DCTsteganographyCERep` class, and then envelops the changed coefficients back to the cover image, which is then saved into the phone's Gallery.

4.2.2 Extracting process

An user chooses an image with embedded image from a Gallery and clicks on "Start extracting" button. The program calls function `Solution.extract()`.

`Solution` object sends the image into the `info.guardianproject.f5android.Extract()` function, which extracts DCT coefficients from the image and then applies the reverse pipeline: calls the extraction function `DCTSteganographyCERep.extract()`, which returns the embedded bytes, then decrypt and decompress them. Then the `Solution` object calls the callback function with the resulting message back into the interface code, which shows it on the display.

4.3 Algorithms' settings

In this section we will show concrete parameters of all used algorithms.

Steganographic algorithm CE

Separation ratio α We've chosen the same value as in the paper: 0.8.

Adjustment parameter β We've chosen the same value as in the paper: $\frac{1}{3}$.

Length of a message We've set maximum length of a block as 2 byte word, so the maximum length is $2^{16} = 65536$ bytes. Therefore, the maximum length of embedded message is $2^{16}(2 - \alpha) = 2^{16}(2 - 0.8) = 78643$ bytes.

Cryptographic algorithm AES

We use AES-256 algorithm in CBC mode with a random initial vector (IV), implemented by the library `com.scottyab.aescrypt.AESCrypt`.

Secret passwords generation

Both steganographic and cryptographic passwords are arbitrary UTF-8 strings. We create them by generating 60 random bytes with `SecureRandom` class from the stan-

dard Java library `java.security` and then encoding them into *Base64* strings. We've chosen the length of the passwords to be divisible by 3 so that the resulting Base64 string would not have a padding. You can read more about Base64 encoding here [41].

We've chosen the Base64 encoding as it encodes the bytes into the most common set of printable characters and is faster (and easier to implement correctly) than performing a random pick from a set based on random bytes.

Cryptographic key is derived from a cryptographic password by `AESCrypt` library. Steganographic algorithm uses stego password as a seed for random generator.

Compression algorithm GZIP

We use default Java library classes for GZIP compression with default settings.

4.4 Testing the steganographic algorithm

We decided to test whether the main parts of the steganographic algorithm are correct, i.e. whether the paired transformations are mutually inverse.

The steganographic algorithm has three main pairs of transformation functions:

- `encodeBits2Byte` and `encodeByte2Bits` — encoding a byte as a bit array (little endian encoding)
- `encodeInt2Bytes` and `encodeBytes2Int` — encoding an integer as a byte array of a fixed length
- `embedBit` and `extractBit` — embedding a secret bit into a DCT coefficient and vice versa according to our steganographic algorithm

As the functions has only small finite amount of possible inputs, we've decided to test their mutual reversibility exhaustively (i.e. on every possible input).

During the tests, we've found that the embedding algorithm from the original paper [37] was incorrect. We've been able to pinpoint and correct the error thanks to a decryption of the inverse process in the paper. In this paper we give the correct implementation of this algorithm (and by that we mean that the functions `embedBit` and `extractBit` are mutually inverse). You can find the description of the error in the listing in the subsection 3.2.1.

4.5 How to replace used algorithms

In this section we will describe the interfaces of used algorithms.

Steganographic algorithm interface

`DCTSteganography` interface has four functions.

Function `loadKeyFromString` takes one argument — a string representation of a steganographic key. It returns `true` if the key was successfully extracted and `false` otherwise.

Function `generateStegoKeyString` returns a valid string representation of a newly created steganographic key (this string should be parsable by function `loadKeyFromString`).

Function `embed` takes two arguments: an integer array `coeffOrig` representing quantized DCT coefficients of a cover image and an `InputStream` with bytes of a secret message we want to embed into the cover image. This function returns nothing, but it embeds the secret message into the DCT coefficients.

Function `extract` takes two arguments: an integer array `coeffOrig` representing quantized DCT coefficients of a cover image with embedded message and an `ExtractionListener` that have to be called with the extracted message as an argument.

```
public interface DCTSteganography {
    boolean loadKeyFromString (final String s);
    String generateStegoKeyString ();

    void embed (int[] coeffOrig,
            final InputStream embeddedData);

    void extract (int[] coeffOrig,
            ByteArrayOutputStream fos, Extract.ExtractionListener listener);
}
```

Cryptographic algorithm interface

`SymmetricEncryption` interface has four functions.

Function `loadKeyFromString` takes one argument — a string representation of a cryptographic key. It returns `true` if the key was successfully extracted and `false` otherwise.

Function `generateCryptoKeyString` returns a valid string representation of a newly created cryptographic key (this string should be parsable by function `loadKeyFromString`).

Function `encrypt` takes one argument — a byte array that represents a message to be encrypted and returns a byte array representing the encrypted message.

Function `decrypt` takes one argument — a byte array that represents an encrypted message and returns a byte array representing the original message.

```
public interface SymmetricEncryption {
    boolean loadKeyFromString (final String s);
    String generateCryptoKeyString ();
    byte[] encrypt (byte[] text);
    byte[] decrypt (byte[] message);
}
```

Compression algorithm interface

`Compression` interface has two functions: `compress` and `decompress`.

Function `compress` takes one argument — a byte array representing a message and returns a byte array with compressed message.

Function `decompress` takes one argument — a byte array representing a compressed message and returns a byte array with an original message.

```
public interface Compression {
    byte[] compress (byte[] message);
    byte[] decompress (byte[] message);
}
```

Summary

In first chapter of this paper we have given an overview of several modern approaches to digital image-based steganography (LSB embedding, F5 embedding, complementary embedding) and steganalysis (chi-square attack, S-family attack, attacks using higher order statistics and attacks based on Benford's law). We have ended this chapter with analysis of currently available free applications and concluded, that most of them have many flaws.

Based on this knowledge, we have deduced a set of requirements to a steganographic application, that would eliminate most of these mistakes, in chapter 2.

In third chapter, we've described algorithms, used in the application. Then we've discussed the possibility of using lossy channels to send images. We ended this chapter with a set of "good policies", that a user should follow to achieve maximum effectiveness of our application.

In fourth (and last) chapter of this paper, we've given an overview of technical aspects of our application and described how to substitute used steganographic and cryptographic algorithms with new ones.

During our work, we've implemented complementary embedding steganographic algorithm in Java, so future Android applications won't have to use inferior F5 library no more. We've also found a mistake in CE algorithm description in the original paper, which we've fixed in our implementation (see chapter 3 for details). We also developed a skeleton of an image steganography application, so future developers won't have to implement JPEG compression and decompression algorithms by themselves.

Future work

Our application could be enhanced in several ways:

- possibility of using asymmetric cryptography
- key management (something like a contact book)
- more eye-candy interface
- possibility to send cover images through lossy channels (briefly discussed in third chapter)

- self-testing with modern steganalytic tools
- running the calculations as background processes
- in-app possibility to send cover images (e.g. “Share” button)

Bibliography

[1] Andreas Westfeld and Andreas Pfitzmann. Attacks on steganographic systems. In *Information Hiding*, pages 61–76. Springer, 1999.

[2] Gregory K Wallace. The JPEG still picture compression standard. *Consumer Electronics, IEEE Transactions on*, 38(1):xviii–xxxiv, 1992.

[3] Niels Provos and Peter Honeyman. Detecting steganographic content on the internet. Technical report, Center for Information Technology Integration, 2001.

[4] Andreas Westfeld. F5—a steganographic algorithm. In *Information hiding*, pages 289–302. Springer, 2001.

[5] Jessica Fridrich, Miroslav Goljan, and Dorin Hogea. Attacking the outguess. In *Proceedings of the ACM Workshop on Multimedia and Security*, volume 2002. Juan-les-Pins, France, 2002.

[6] Jessica Fridrich, Miroslav Goljan, and Dorin Hogea. Steganalysis of JPEG images: Breaking the F5 algorithm. In *Information Hiding*, pages 310–323. Springer, 2002.

[7] Panagiotis Andriotis, Theo Tryfonas, George Oikonomou, Theodoros Spyridopoulos, Alexandros Zaharis, Adamantini Martini, and Ioannis Askoxylakis. On two different methods for steganography detection in JPEG images with Benford's law. In *proceedings of the 7th Scientific NATO Conference in Security and Protection of Information (SPI 2013)*, 2013.

[8] Panagiotis Andriotis, George Oikonomou, and Theo Tryfonas. JPEG steganography detection with Benford's law. *Digital Investigation*, 9(3):246–257, 2013.

[9] Walter R Mebane Jr. Election forensics: the second-digit Benford's law test and recent American presidential elections. In *Election Fraud Conference*. Citeseer, 2006.

[10] Boudewijn F Roukema. Benford's law anomalies in the 2009 Iranian presidential election. *Unpublished manuscript*, 2009.

[11] Joseph Deckert, Mikhail Myagkov, and Peter C Ordeshook. Benford's law and the detection of election fraud. *Political Analysis*, 19(3):245–268, 2011.

[12] Dongdong Fu, Yun Q Shi, and Wei Su. A generalized Benford's law for JPEG coefficients and its applications in image forensics. In *Electronic Imaging 2007*, pages 65051L–65051L. International Society for Optics and Photonics, 2007.

[13] Siwei Lyu and Hany Farid. Detecting hidden messages using higher-order statistics and support vector machines. In *Information Hiding*, pages 340–354. Springer, 2002.

[14] Palghat Vaidyanathan. Quadrature mirror filter banks, M-band extensions and perfect-reconstruction techniques. *ASSP Magazine, IEEE*, 4(3):4–20, 1987.

[15] Max Welling. Fisher linear discriminant analysis. *Department of Computer Science, University of Toronto*, 3, 2005.

[16] Vladimir Vapnik. *The nature of statistical learning theory*. Springer Science & Business Media, 2013.

[17] Jan Meznik. Steganography. `https://play.google.com/store/apps/details?id=com.meznik.Steganography`. Last accessed on May 10, 2016.

[18] Dino Trnka. Steganography master. `https://play.google.com/store/apps/details?id=com.dinaga.photosecret`. Last accessed on May 10, 2016.

[19] Newidget Dev. Niastego. `https://play.google.com/store/apps/details?id=dev.nia.niastego`. Last accessed on May 10, 2016.

[20] Talixa Software. Pocket stego. `https://play.google.com/store/apps/details?id=com.talixa.pocketstego`. Last accessed on May 10, 2016.

[21] Shweta Gupta. Hidden secrets. `https://play.google.com/store/apps/details?id=stego.lsb.hiddensecret`. Last accessed on May 10, 2016.

[22] FlyJam. Secret messages cryptapp. `https://play.google.com/store/apps/details?id=flyjam.CryptApp`. Last accessed on May 10, 2016.

[23] icokocev. Vipsecret. `https://play.google.com/store/apps/details?id=com.vipcontest.vsecret`. Last accessed on May 10, 2016.

[24] TROTTA Gianpaolo Francesco. Ptbox free. `https://play.google.com/store/apps/details?id=gftrotta.dpatruno.ptboxfree`. Last accessed on May 10, 2016.

[25] Studio Mastodonte. Steganosaurus. https://play.google.com/store/apps/details?id=app.steganosaurus. Last accessed on May 10, 2016.

[26] s.r.o. Panacom. Stegos. https://play.google.com/store/apps/details?id=sk.panacom.stegos. Last accessed on May 10, 2016.

[27] abeatte. Stegosaurus. https://play.google.com/store/apps/details?id=com.artbeatte.stegosaurus. Last accessed on May 10, 2016.

[28] Roman Chinkais. Stegais. https://play.google.com/store/apps/details?id=com.romancinkais.stegais. Last accessed on May 10, 2016.

[29] Phoebit. Secret tidings. https://play.google.com/store/apps/details?id=com.phoebit.secrettidings. Last accessed on May 10, 2016.

[30] RADJAB. Da vinci secret image. https://play.google.com/store/apps/details?id=jubatus.android.davinci. Last accessed on May 10, 2016.

[31] Francisco Ruiz. Passlok. https://play.google.com/store/apps/details?id=com.fruiz500.passlok. Last accessed on May 10, 2016.

[32] Bearded Man. Stegolite. https://play.google.com/store/apps/details?id=jepaweb.stegasaurus. Last accessed on May 10, 2016.

[33] Batsakidis Athanasios. Kryfto. https://play.google.com/store/apps/details?id=it.kryfto. Last accessed on May 10, 2016.

[34] QitVision. Cryptego. https://play.google.com/store/apps/details?id=com.qitvision.cryptego. Last accessed on May 10, 2016.

[35] The Guardian Project. Pixelknot. https://play.google.com/store/apps/details?id=info.guardianproject.pixelknot. Last accessed on May 10, 2016.

[36] Richard Ostertág. Počítačová steganografia. Master's thesis, Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava, 1996.

[37] Chiang-Lung Liu and Shiang-Rong Liao. High-performance JPEG steganography using complementary embedding strategy. *Pattern Recognition*, 41(9):2945–2955, 2008.

[38] NIST-FIPS Standard. Announcing the advanced encryption standard (AES). *Federal Information Processing Standards Publication*, 197:1–51, 2001.

[39] L Peter Deutsch. DEFLATE compressed data format specification version 1.3. 1996.

[40] L Peter Deutsch. GZIP file format specification version 4.3. 1996.

[41] Simon Josefsson. The base16, base32, and base64 data encodings. 2006.

[42] Lifang Yu, Yao Zhao, Rongrong Ni, and Zhenfeng Zhu. PM1 steganography in JPEG images using genetic algorithm. *Soft Computing*, 13(4):393–400, 2009.

[43] Tuomas Aura. Practical invisibility in digital communication. In *Information Hiding*, pages 265–278. Springer, 1996.