Steganography using Graph Theory

P2-project
Group A325a
Software
Aalborg University
May 24, 2016

AALBORG UNIVERSITY
STUDENT REPORT

Faculty Office for Engineering and Science
Strandvejen 12-14
Phone: 96 35 97 31
Fax: 98 13 63 93
http://tnb.aau.dk

Title:

Steganography using Graph Theory

Project:

P2-project

Project period:

February 2016 - May 2016

Project group:

A325a

Authors:

Anders Langballe Jakobsen
Andreas Neil Jensen
Matias Røikjær Jensen
Rasmus Jespersen
Simon Nagel Linnebjerg
Theis Erik Jendal
Thomas Buhl Andersen

Supervisor:

Søren Enevoldsen

Pagecount: 108
Finished 24-05-2016

Abstract:

This report will cover the development of a steganographic tool, from the initial studies of the field of steganography and the associated social, legal and ethical context, to the design, implementation, testing and evaluation of the developed tool.
The developed tool is versatile, being able to handle a variety of carrier media, cryptography and steganography algorithms, and user friendly with an intuitive user interface.
As part of the development of this steganographic tool an algorithm using graph-theory has been implemented in order to minimize statistical detectability of the steganographic message.
This algorithm handles the edge finding by using a neighborhood search, a solution that enables the program to be able to handle large messages efficiently.
The algorithm is shown to be able to hide messages well, only needing to apply direct adjustment to approximately 0.1% of the vertices.

Preface

This report was written as a second semester software engineering project at Aalborg University's Faculty of Engineering and Science (TEK-NAT), by study group A325a, in the spring of 2016. Its main purpose being research, implementation and utilization of steganography.

The inspiration for the project lies with the versatility of the technology. It has wide applications and is already utilized for many copyright watermarking techniques throughout the world. It is also used for hidden messaging, where two individuals can share information, without anyone being aware of any hidden information being sent. Furthermore the subject presents both societal and mathematical challenges, that made it an interesting project to work on.

It is our hope that the end solution can help people share information, and be at the forefront of tools for freedom of speech and privacy, without them having to fear prosecution or outside involvement in private information sharing.

As for the mathematical challenges, we would like to thank the members of mathematical study group A402, Simon Nicolai Nielsen, Helle Rask and Mikkel Højlund Larsen from TEK-NAT, with whom we have had a collaboration, for helping us develop a graph theoretic algorithm to achieve our goal of imperceptibility in our steganographic algorithm.

We would further like to thank the members of computer science study group A419, Elyas Bassam Bahodi, Palle Thilemann, Ruben Mensink, Frederik Østerby Hansen, Joachim T. H. Nielsen and Jonatham Karlsson from TEK-NAT, that has helped with the development of the conceptual design for the end solution.

Contents

1 Project description **1**
1.1 Purpose . . . 1
1.2 Methods . . . 1

I Background **5**

2 Introduction **7**

3 Social and ethical context **9**
3.1 Users of steganography and steganalysis . . . 9
3.2 Moral and ethical issues . . . 14
3.3 Legal issues . . . 15

4 Technologies **17**
4.1 Steganographic approaches . . . 17
4.2 Compression of hidden messages . . . 19
4.3 Carrier media . . . 20
4.4 Communication channels . . . 24
4.5 Cryptography . . . 25
4.6 Existing solutions . . . 28
4.7 State of the art . . . 29

5 Problem definition **31**

II Solution **33**

6 Design **35**
6.1 Criteria . . . 35
6.2 Graph-theoretic approach . . . 36
6.3 File archive . . . 38
6.4 Digital signing . . . 38
6.5 Multiple encryption keys . . . 39
6.6 Use of cryptography . . . 39
6.7 Supported file formats . . . 40
6.8 Graphical User Interface . . . 42

7 Implementation **45**
7.1 General program flow . . . 45
7.2 Utility . . . 46
7.2.1 StreamExtensions . . . 46

7.2.2 RandomNumberList . . . 48
7.2.3 IconExtractor . . . 48
7.3 StegoMessage . . . 48
7.4 Algorithms . . . 50
7.4.1 StegoAlgorithmBase . . . 51
7.4.2 LSB Algorithm . . . 51
7.4.3 Graph Theoretic Algorithm . . . 53
7.4.4 Common Sample Algorithm . . . 59
7.5 Cryptography . . . 60
7.5.1 ICryptoProvider . . . 61
7.5.2 AESProvider . . . 61
7.5.3 RSAProvider . . . 62
7.6 Carriers . . . 62
7.6.1 ICarrierMedia . . . 63
7.6.2 AudioCarrier . . . 63
7.6.3 ImageCarrier . . . 64
7.7 Graphical user interface . . . 66
7.8 Exceptions . . . 69

8 Test **71**
8.1 Unit tests . . . 71
8.1.1 Test of cryptography algorithms . . . 71
8.1.2 Test of StegoMessage . . . 72
8.1.3 Test of RandomNumberList . . . 72
8.1.4 Mock tests . . . 72

IIIEvaluation and further work **75**

9 Evaluation **77**
9.1 Algorithm comparison . . . 77
9.2 Moral and ethical issues . . . 82
9.3 Final evaluation . . . 83

10 Future work **85**

11 Conclusion **89**

12 Appendix **91**
12.1 Coding style . . . 91
12.2 Glossary . . . 93
12.3 Inheritance diagram . . . 94

Bibliography **95**

Project description 1

In this chapter we will describe what is required of the report and program, what methods have been used to evaluate and delineate the interested parties and methods used in program development.

1.1 Purpose

The second semester of the software bachelor at AAU, has a curriculum that aims to express the purpose of P2-project and therefore this report.

The purpose being:

> That the student acquires skills working as a group in a problem oriented project, as well as knowledge on coherency between problem definition, the role of modeling, the construction of programs and programs as solutions to a problem in context to a given problem. Furthermore to enhance knowledge of the profession's contents and the professions potentials.

In parallel with the P2-project, courses in Discrete Mathematics, Object Orientated Programming and Computer Architecture were held. We will use the knowledge gained in the courses to improve and enhance the program throughout the process of the project.

A requirement for this project was, that we, in a group, should develop a "larger program of high quality" and describe the essential parts of the program.

1.2 Methods

During the conception of this report, certain methods have been used in order to achieve a level of understanding which is necessary for the proper product to take form. This section seeks to explain these methods and make it clear, why the methods in question are valuable tools for the creation of an end product.

Information gathering methods

Information gathering often proves to be a tricky subject, due to sources not always being objective, the information not always being current or the information being plain wrong. As such, information gathering often poses a problem for people wanting to make use of the given information in a objective context and therefore need the information to be current and correct.

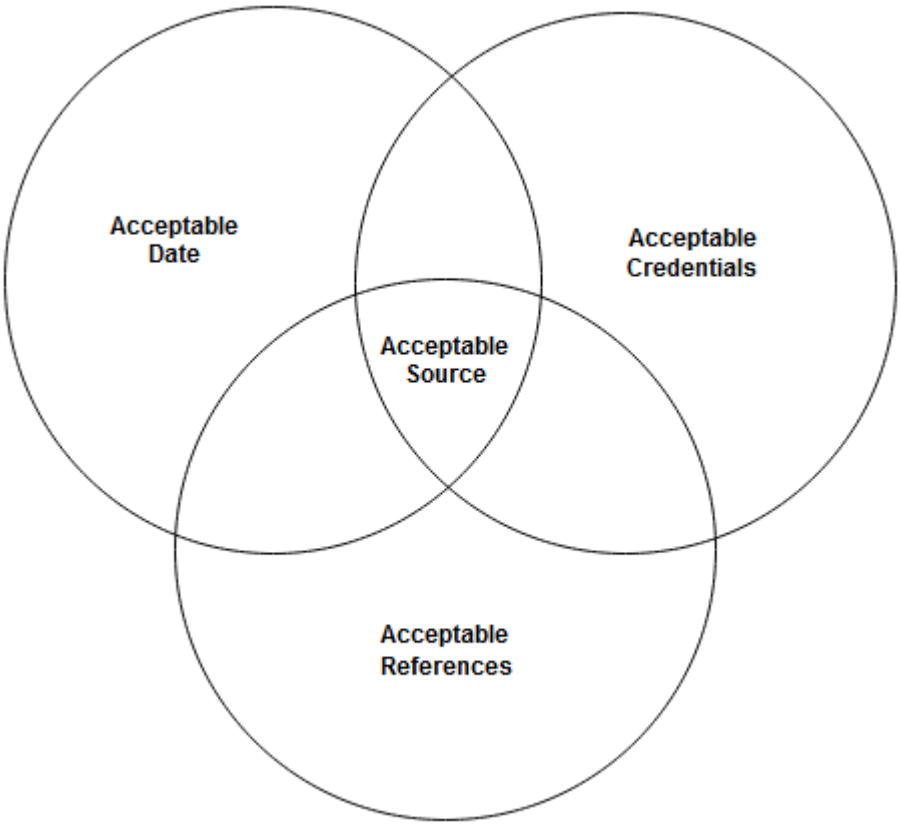


Figure 1.1. Source critique

In this report, most of the information gathering has been done through the Internet, through sites such as Google, Google Scholar and the likes. This has forced us to take the objectivity of each individual source into account, and as such we have taken certain precautions in order to avoid the usage of incorrect or biased opinions.

There has mainly been a focus on factors such as date, credentials and references. In the case of an old information source, a judgement has been made whether the information is still current or was obsolete, and if possible more recent and more relevant data on the subject was sought.

If the author of the information was deemed unreliable, the information has not been utilized. Unreliable, in this context, has been defined as not having the appropriate credentials in order to support the given claim, which could be anything from former acknowledged articles to relevant education and certificates.

If the information has been mentioned in other media, or is making reference to other media, the given media will also be under scrutiny. Should the given media fail to meet the criteria listed above, this of course raises the chance of the information not being utilized.

These requirements overlap with each other, creating a 'zone' of what is viable information, as represented in Figure 1.1, where the overlapping circles create the 'acceptable source zone' in which we have tried to stay while gathering information for this report.

Furthermore, we have chosen not to utilize information found directly on Wikipedia, or blogs of the same character, as these sources are open to alteration from any author. However, we have chosen to use some of the sources that have been used on Wikipedia, as these often refer to academic articles that relate to the subject at hand.

User analysis methods

A user analysis is meant to give an understanding of the individuals that potentially could use the end product. By utilizing a user analysis, we gain a better understanding of what features that might be needed in the end product.

The user analysis that we have performed, is based on information researched and analyzed through the Internet. This gives a base knowledge about potential users. However, because of the nature of steganography, it is hard to gather any relevant information about users of steganography and steganalysis, as the purpose of steganography is to remain hidden. However, we have come across various information regarding different uses of the technology, and have used this to gain a better understanding of the potential users of a steganographic tool.

During the user analysis we compare different user groups, which leads us to conclude on the information gathered on the different user groups, and consider possible scenarios around the specific user groups and what they may need in a steganographic tool. By utilizing these comparisons of the users, we have made an estimate of what our users would need in a solution, and how we should design the program.

Technology analysis methods

By utilizing a technology analysis, one can view a given product in the context of relevant technologies. The technology analysis helps determine which technologies are relevant or even necessary for any eventual products to be based on. It also serves as a early warning system, seeing as there could be interfering technologies.

In our specific report, the technology analysis has been created as a result of information gathering centered around the subject of steganography.

Our technology analysis, and the results thereof, have furthermore been utilized in our problem definition in order to better achieve a deeper understanding of what our end product should contain.

Design methods

Throughout the design chapter, we utilize the information that we have gathered in the technology and user analysis, as this information is the base around which we design our solution. Thus we have used a couple of methods that can graphically show the layout of the program, and how we intend to implement the solution, in the following chapter implementation.

The methods that we have used, to keep track of our ideas:

- **Mind mapping:** Used to help us with the overview of the ideas for the program solution and implementation of our problem definition.
- **Flowcharts:** After the initial idea conception, an initial program design was made as a flowchart, which served to visualize the desired flow through the classes and methods. Starting with the overall layout of the program and then subsequently going deeper into class specific details.
- **Group design:** A lot of the design process, has been debating different solutions within the project group to find the best solution to any given design problem in the design process.

Implementation methods

We will be implementing the program with an object oriented programming approach [1]. Through this approach we can implement the data types that we use, and control which operations that can be done on specific data types in the data structure. This way we can handle the data structure as objects.

This gives us a range of possibilities as this method allow us to create individual classes that will be treated as objects at run-time. This, in turn, makes it easier to read and maintain as the objects of one type would most likely have the same characteristics in any given context.

By implementing interfaces it is possible to make the program easier to maintain as an interface is the base set of rules for the object. Hence any new implementations of a similar object can follow these predetermined rules and it will function with the base object.

Furthermore, the program will be written in C# and will use the standard libraries from this coding language, namely the .NET Framework library.

Test methods

To ensure functionality of the program, unit testing has been used. Unit testing tests individual methods of the program, which allows us to predict an outcome and see if the output is as expected after running the method from the program. If the output is the same as our expected output the test passes, otherwise the test will fail. It is important to mention that a passed test does not necessarily mean that the method is free of errors. The use of unit tests allows us to more easily find errors since the methods tested is isolated from the rest of the program.

We have also used mock testing of our program. A mock test is a test that tests the whole program in one test. The purpose of this is to see if all parts of the program interact as expected and at the same time gets the expected results.

Ad hoc testing is another means of testing that we have used. It works by running the code continually while developing parts of the program and looking at the output to confirm that it behaves as expected.

Part I

Background

2 Introduction

According to Amnesty International, governments are increasingly trying to surveil different groups of individuals [2]. This creates a situation where individuals with relatively few resources available to them, compared to their government, try to gain or maintain their right to freedom of speech and privacy. In the ongoing debate of needed surveillance versus the right to privacy and freedom of speech, we think it is a worrying trend that governments are intentionally utilizing surveillance towards their respective citizens.

Given that the situation with government surveillance is increasing and the fact that information sharing on the Internet is becoming increasingly popular, it gives us an opportunity to delve into the possibilities of hidden communication. A way of communicating without other people knowing what is communicated is to use cryptography. However, a problem with utilizing only encryption is that some of the world's governments would potentially react violently with the bare knowledge of unlawful messages. This is why it is a good opportunity to utilize steganography that can hide messages discreetly in otherwise harmless media.

Like cryptography, steganography is the process of hiding information. However, these processes are different as steganography deals with hiding the existence of the information whereas cryptographically encoded information appears unrecognizable and may raise suspicion. Steganography has existed long before the digital age, aging back to Ancient Greece and Rome, but today it is utilized to hide messages in digital formats such as images and audio files [3]. Due to the nature of steganography, it can be difficult to evaluate its popularity as the use of it should go unnoticed. However, some uses of steganography have been uncovered by malware analysts in recent years which proves that the process is still in use [4].

This report will cover various aspects of steganography, including potential users and relevant technologies and techniques, as well as the creation of a tool that can reliably send messages without detection by using some of the more known steganographic methods and a graph theoretic approach. We will do this based on the initial problem definition, which is:

> How can we create a steganography tool that can create undetectable steganographic messages?

Social and ethical context 3

When dealing with hidden communication there are some moral, ethical and social issues that have to be taken into account. While hidden communication can be helpful to activists, journalists and the likes. It can also be equally harmful when used by terrorists. These issues are not to be taken lightly as a freedom fighter in one nation's perspective can be a terrorist in the eyes of another nation. As such, the definition of good and evil are not as consistent as one would think but instead consists of grey areas. This is for example seen in the Israel-Palestine conflict. Here, the Palestinians are considered terrorists by the Israeli government but considered freedom fighters by many in the western world.

This means that throughout this chapter each individual party that has an interest in steganography or the detection thereof (steganalysis) will be analyzed on an individual basis. This is meant to reveal each party's interests in using steganography and what they use steganography for and as such give an insight into their ethical, social and moral reasoning for using steganography.

3.1 Users of steganography and steganalysis

To enable a proper definition of a problem to be solved, it is necessary to examine the users in question in regards to steganography. The following analysis will cover two major areas of interest: The interest in using steganography and the interest in discovering steganography using steganalysis. These can be seen as opposing but not mutually exclusive interests.

Steganography users have an interest in hiding information. This information can be applied in many areas and has a usefulness not only for personal or criminal use, but also for government or corporate offices where secret information can be sent with reduced risk of unauthorized interception. This information could range from classified information such as state secrets to simply private information such as citizens taxes.

Steganalysis users have an interest in discovering hidden data. This is mostly done with statistical analysis of the data structure in question and is mainly used by governments or corporate offices. This may heighten a nation's or corporate office's security and enable them to preemptively make decisions to help their own interests such as increasing security based on intercepted messages and thus prevent an attack.

This section will be a basis for further research as different users have different requirements to a potential steganography program.

Activists

Activists are people in general need of a way to get information to individuals outside their country's border because of the current situation in their own nation. This can be defined as the situation like the one currently going on in Syria where the terror group, Islamic State, is currently operating [5].

The situation in Syria is that the Syrian citizens are battling not only a civil war against their leader, Bashar Al Azzad, but are also fighting an ongoing battle against Islamic State. This leaves open channel communication vulnerable to surveillance and the activists would therefore require a steganographic solution for hidden communication with the world around them.

If such a steganographic solution was made for activists they would be able to communicate with the outside world without fear of prosecution from their state or opposing parties. This would increase their chances of getting international support for their cause. However, should the amount of steganography used by activists start to rise the government in question would likely start to increase their efforts in detecting this method thus, increasing their efforts in steganalysis, effectively creating an 'IT arms-race'.

Government

The government is divided into a variety of agencies that, in the course of fulfilling their obligations, use steganography and steganalysis. These government departments have different needs, because of the different nature of their overall assignments, and is divided into their own segment of steganography and steganalysis.

Agencies like the American National Security Agency (NSA) or the Danish Politiets Efterretningstjeneste (PET) can use steganography to hide messages to and from different parties such as agents operating undercover in foreign countries. This will enable the agents to hide in plain sight as they spy for the interests of the particular nation in question. The fact that nations are spying on each other is a common occurrence, for example the infiltration of a core Syrian router in 2012 [6]. However not much concrete information can be found about how governments apply steganography and steganalysis as this information is closely guarded as what each nation defines as a matter of "national security". The disclosed information here about governments use of espionage is only known because of the actions of Edward Snowden who conducted the biggest national security breach in the history of the United States of America. This information is mostly about information gathering and country on country espionage and does not cover the area of steganography.

It is, however, proven by the cyber security firm FireEye that an alleged Russian hacker group is responsible for a malware named "Hammertoss" [7]. This malware is directly connected to Twitter where the group can send

steganographic images to a specific account and, by doing that, activate the malware of the same name on infected computers. The malware will then send various information to a cloud server where the perpetrators can download the data. This method could theoretically be used in a government perspective where data can be gathered from personal computers or corporation server systems if this data is not encrypted. At the same time this example can also determine that national security agencies have to increase their efforts to detect steganography on social networks like Twitter. This means that the national security agencies have to take part in both steganography and steganalysis.

Specifics about law enforcement forensic methods, relating to steganography and steganalysis, are equally hard to come by. This may be because of the abilities of IT-criminals to turn that information to their own advantage. It is, however, safe to assume that government authorities, like the police, are able to, to some extent, use steganalysis to uncover criminals as this information can be crucial for a court proceeding. As such, the police has an interest in steganalysis, and therefore in steganography, because of a need to uphold the law of the nation in question.

Journalists

Journalists sometimes try to uncover stories about moral, ethical or legal wrongdoings and make these known to the world around them. Thereby journalism requires credible information as a primary source which could be an individual, or a document written by an individual, who has a direct involvement in the situation the journalist is writing about [8]. The source can live in a dangerous environment where secrecy is a vital part of communicating certain information. Often the safest option for the journalist and the source is to stay offline and meet in person, but this is not always an option and can also be very impractical.

As described in section 3.1 about activists, there are governments and groups that want to retain information from public knowledge and disclosing the information could lead to prison or death. A case, Goodwin v. United Kingdom, in 1996, the European Court of Human Rights stated that;

> "*Protection of journalistic sources is one of the basic conditions for press freedom ... Without such protection, sources may be deterred from assisting the press in informing the public on matters of public interest. As a result the vital public-watchdog role of the press may be undermined and the ability of the press to provide accurate and reliable information may be adversely affected*" [9]

Meaning that the public in some countries and some cases are afraid of sharing their information with journalists since this can be a risk to the journalistic source. Because of the danger of disclosing information in some countries, journalists can have difficulties finding sources unless they can communicate safely and without concern from their sources [10]. According to "Investigating the Computer Security Practices and Needs of Journalists" [10], the sources

rarely have access to secure communication technology. The journalist, according to this report, often use whatever the sources feel comfortable with using for communication. This means that the journalist may knowingly use technologies that are unsafe or unfit for communication to accommodate their sense of security.

The problems mentioned in this chapter can possibly be solved by using steganography, encryption and similar forms of concealment. This can ensure safety or drastically reduce the risk of getting uncovered by governments that do not want certain information to be shared with the public. If the sources feel safe they are more likely to share their information and the journalists then has the possibility to pass said information on to the public. Programs that can keep the communication secret could be a great asset to the journalist, but the program has to be easily obtainable and easy to use.

Privacy advocates

Privacy advocates are those who want to protect themselves from prying eyes whether those are government bodies, criminals with ill intentions or common individuals. As such, a subset of these advocates may be referred to as IT-activists as their sole intention with their battle for privacy is to counter mass surveillance. Some of these people may have an interest in hiding certain files on their file system, while others may wish to send messages to their friends, without anyone being able to surveil the conversation. The motivation for these groups may seem vague or difficult to understand for someone who does not share their view on privacy, but the matter is quite important for the privacy advocates.

An argument often used against this group is the 'nothing to hide' argument. People who support this argument will argue that mass surveillance does not threaten the privacy of individuals unless government agencies can uncover illegal activities done by this individual. In this case, the individual does not have the right for privacy according to the supporters of the argument. On the other hand, a privacy advocate would argue that the argument implies full trust in the state, which they argue can be hard to assume, even in fully developed nations. Steganography could be a solution for these individuals to achieve the privacy that they desire as they would be able to hide from anyone trying to abuse their privacy.

Corporations

Corporations, in the scope of steganography, are firms that have an interest in using steganography to identify their products. In this case we are not considering corporations that work with products that do not have a software element.

Steganography, in this sense, is used by these corporations to watermark their products. In this way they hide information in the product which can be used to identify the specific item and the rightful copyright owner. This can

help prevent unauthorized sharing of for example videos by identifying who is responsible for the leak and thus punish the responsible party and possibly blacklisting said person to prevent future leaks.

However, other uses of steganography pose significant risks for corporations, such as corporate espionage, as it enables the exchange of covert messages in otherwise harmless media. Many companies monitor employee emails and would be able to identify obvious encrypted data but not data obscured in harmless media such as a photo.

Criminals

It is easy to imagine the uses a steganography tool could have for a criminal or group of criminals. It could be used in order to keep their communication going back and forth between involved parties and hidden from the police. They would try to hide their communication so that they would not get prevented in doing their criminal act as planned. Coordination of illegal activities would in many cases be the only use a steganography tool would have to criminals not involved in cyber criminality such as robbers or smugglers. In other cases steganography could also be used to distribute illegal materials as for example child pornography or bomb blueprints.

Cyber criminals will in some cases utilize more complex steganography. Pierluigi Paganini wrote that hackers have previously used steganography to extract information from a corporation´s servers by doing the actual steganography on the corporation´s servers, thereby being able to get the information out of the corporation without them noticing anything but ordinary traffic [11].

Hackers have previously used steganography to control botnets [4]. A botnet is a network of computers that can be used by the hacker without the original computers owner knowing about it. This botnet could, for example, be used to do a distributed denial-of-service (DDoS) attack or other actions that would be faster or more efficient, by using the botnet, than if the hacker would only use his own computer.

As an example Trend Micro has written in a threat report that hackers have used steganography to control a botnet [12]. An infected computer would automatically download an image file without the owner of the computer knowing about it. This image file would than be subject to steganographic messages in order to utilise the botnet. If the owner of the infected computer where to ever see this image somewhere on their computer, it would just show an image of for example a sunset or a cat which would not cause any immediate concern by the owner.

Intended audience

In summation of the social aspect of this report, we choose to focus on activists working against regimes suppressing its people, as we see the greatest potential of increasing freedom of speech and information sharing, by developing a

steganographic alternative to open communication for this group of individuals. In addition, steganalysis will no longer be a focus for the remainder of the report, only in the context of how it can be evaded.

3.2 Moral and ethical issues

Certain moral and ethical issues inherent to the nature of steganography has been deemed necessary to document, in order to assure our own moral and ethical standing, and to prevent the negligence of potential ways of preventing the misuse of our steganography solution. The following is the result of the above mentioned considerations.

It is our belief that a steganographic program would be able to help the progression of freedom of speech, promoting understanding and tolerance across nations and cultures, but a program to send covert messages will also have the potential to be used for acts of terror. Some safeguards to be considered to ensure the program is not being misused to perform criminal or terrorist activities are as follows:

1. **Backdoor:** A backdoor [13] is a way for developers, or other interested parties, to gain access to a program, or a programs distributed content, remotely by utilising one of two ways:

   Program backdoor: A purposely placed backdoor through the program itself.
   Algorithm backdoor: A purposely placed flaw in the programs encryption algorithm.

   Having a program backdoor in the program could help authorities detect and shut down any criminals using the program. However, this would require the authorities to know the global Internet Protocol (IP) address of the given user and since the main goal of steganography is to remain hidden it serves no purpose to have such information located in a database to disclose at a later point. It also makes the program susceptible to unwanted attacks, as backdoors can be detected. If hackers or other interested parties try to gain access to the program and gain access to the hidden information distributed by the program user. Hence this method will most likely be ineffective or render the program unusable for hidden communication.
2. **Registration:** Another solution to our problem is to register our users. However, this will also work against the basics of steganography as said users are trying to remain anonymous and hence will not find any use of our program because of the lack of anonymity.
3. **Hard copy transfer:** A hard copy transfer on a Universal Serial Bus (USB) flash drive, or alternative secondary memory device, would make it possible to share the program with individuals that the programmers can judge before they get their hands on the program. However, this method is highly ineffective as distributing the program becomes highly complicated. It removes anonymity and has very little lifetime as removing the hard copy and making it into a soft copy would take very little effort to

> accomplish and the program would end up on the Internet anyway. At the same time, it does not safeguard anything other than the programmers, or other authorities, having to get involved in activism or terrorism which is most undesirable.

In summation, trying to implement security or safeguards against the misuse of the program, will either lead to a nullification of anonymity or work against the very ideas of steganography. However, as the creators of this program, it will be created for the purposes of promoting free speech. As such, the intended purpose of the program is to advance freedom of speech in countries where such ideals are not well met. And we believe that the program can do far more positive than negative. Hence, we feel that the moral and ethical reasons for developing a program to assist the promotion of free speech is fairly justified.

3.3 Legal issues

While working with steganography there is a series of legal issues that have to be taken into account. Every country in the world has some kind of legislation on the subject of cryptography. This has a series of different reasons where national security is one of the main reasons [14]. Due to the nature of these legislations we will only be focusing on the Danish law and the arrangements that the European Union and Denmark have in common.

According to Danish law there are no legislations against the use of cryptography [15]. However, because of Denmark's membership of the European Union, Denmark does have an obligation to uphold the Wassenaar arrangement [16] where dual-use technologies like cryptographic software is part of the arrangement.

The arrangement states different limitations for the use of cryptography, for example key length and algorithm type, on software that is not openly available to the public. Though these limitations are in place in the Wassenaar arrangement any and all software that is freely available to the public is not restricted by the Wassenaar arrangement. Be it through sales or open source sharing as shown in "General software note (GSN)" of the Wassenaar arrangement [16].

> (This note overrides any control within section D of Categories 0 to 9.)
>
> Categories 0 to 9 of this list does not control "software" which is either: a.
>
> Generally available to the public by being: 1. Sold from stock at retail selling points, without restriction, by means of: a. Over-the-counter transactions; b. Mail order transactions; c. Electronic transactions; or d. Telephone order transactions; and 2. Designed for installation by the user without further substantial support by the supplier; or N.B. Entry a. of the General Software Note does

not release "software" specified in Category 5 - Part 2 ("Information Security").

b. "In the public domain". [16]

To conclude, this means that the group can make and use any and all forms of cryptography in the development of our program as it will be open source. Hence we are not limited by any laws under the Danish or EU law. No further legal issues will be taken into consideration as it would be highly complicated and outside the scope of the project to design around all of the world's laws regarding cryptography.

Technologies 4

In this section various technologies connected to steganography will be examined. We will do this by examining different steganographic methods, algorithms and relevant technologies that could be used in combination with steganograhy to gain a better understanding of the technologies that could be used to help activists.

Thereafter possible communication channels, possibilities and limitations will be examined to find a suitable platform for sharing the steganographic messages.

In the end various existing programs will be evaluated followed by a brief look at the state-of-the-art of steganography research.

4.1 Steganographic approaches

This section serves as an overview of some steganographic techniques or methods that can be used to embed data in some carrier media. These techniques show two different kinds of methods. Two which utilizes a substitution-based algorithm and one that utilizes an insertion-based algorithm. The purpose hereof is to compare the pros and cons of these types of algorithms.

Least Significant Bit method

A method commonly used in digital steganography is the Least Significant Bit (LSB) method. The basic principle of this method is to make small changes to a large amount of data and as such this type of method is referred to as a substitution algorithm. LSB is useful in data formats where small changes do not cause big differences. Embedding hidden messages in text using LSB would be a bad idea as changes would be apparent since the characters would change, but on an image such changes may not be apparent [17]. In a lossless image format like PNG (see section 4.3) the decompressed data is represented as a 2D matrix of pixels. Depending on the pixel format sRGB, RGBA or perhaps even greyscale, each pixel has a set amount of color channels, where each value is a byte (8 bits) representing a number between 0 and 255 which determines the intensity of the color channel on that specific pixel. The purpose of the LSB method is to change the least significant bit in each of these bytes and perhaps even the second least significant bit as well. This means that the decimal value

may only change by 1 if modifying one bit and by 3 if modifying two least significant bits.

For example, say we want to embed the character '*' in a message. This ASCII character has a decimal value of 42, which can be represented as an 8-bit binary number:

$$00101010 \tag{4.1}$$

Since this message is 8 bits long, to embed it we would need to modify 8 bytes when modifying one bit per byte and 4 bytes when modifying two bits per byte. In the first case, this means that we would need two RGBA pixels as this gives us 2 * 4 bytes to hide our message in. For example, let's say we want to hide our image in the two pixels:

$$\begin{array}{cccc} 11111111 & 01001111 & 10110000 & 10000111 \\ 11111111 & 10010110 & 10001110 & 10000001 \end{array} \tag{4.2}$$

With the LSB method, we now need to look at the least significant bit in each of these bytes and negate the bit if it does not correspond to the bit we want to insert. When the message is embedded in the original pixels the produced result is:

$$\begin{array}{cccc} 1111111\mathbf{0} & 0100111\mathbf{0} & 1011000\mathbf{1} & 1000011\mathbf{0} \\ 1111111\mathbf{1} & 1001011\mathbf{0} & 1000111\mathbf{1} & 1000000\mathbf{0} \end{array} \tag{4.3}$$

The embedded message can then be extracted by simply combining each least significant bit of the 8 bytes. After doing this we end up with the message seen in Equation 4.1.

When comparing the pixel values of a modified image (now referred to as a stego-image) and the original image the change is clear on the bit/byte level, but may not be apparent to a human comparing the two images. This means that LSB is prone to steganalysis. With the stego-image and original image in possession it is trivial for a piece of software to detect that one of the images have been edited. In addition, the LSB method will typically increase the size of the image since adding noise to the least significant bits increases the amount of colors and can affect lossless compression used in formats such as PNG (see section 4.3). This happens because an increase in noise leads to more unique colors which results in a larger file after compression.

Graph-theoretic approach

While the method described in the previous section is a sufficient steganography method one might want a method that is less prone to steganalysis. After all, the point of steganography is to hide the fact that communication is taking place. In the method previously described pixels are overwritten and the color frequencies are changed, but with a graph-theoretic approach it is possible to exchange pixels so that the color frequencies are similar when comparing the original and stego-image. Depending on the size of the input message and the picture it might be possible to hide a message in a picture without overwriting any pixels at all.

EOF-method

An example of an insertion based method is the end-of-file (EOF) method. Using the end-of-file method, one simply inserts the steganographic data after the end of the file. This method does not necessarily require any steganographic tools as it can be done manually by using a text editor. This method can be applied universally but may conflict with some file types. It will work with two kinds of file types:

- File types where a header indicates how many bytes should be read. In this case, the steganographic data is inserted after the reader stops reading and therefore not considered by the reader as the header is not modified to read the hidden message.
- File types where an end-of-file character indicates the end of a file at which point the file reader will stop reading further data. The data is inserted after this end-of-file character.

The EOF method does have some issues. First of all, while it does not have a limited bandwidth, with the exception of the user's disk space, the size of the steganographic media is increased by the size of the hidden data which could make it obvious that there is some hidden data in the media. Secondly, there is an issue with some anti-virus programs falsely flagging the stego-media as having a virus because some Trojan viruses use this exact method in order to hide malicious data in executable files.

4.2 Compression of hidden messages

An issue with digital steganography is the bandwidth available in the information carriers, especially static data formats like images. For example, let us say we have a 100px x 100px image with 4 color channels (RGBA). If we use the least significant bit (LSB) method and use the two least significant bits. The amount of bits available in that image will be:

$$100 * 100 * 4 * 2 = 80.000 \tag{4.4}$$

This effectively means that we can fit about 10 kilobytes of data in this image since a byte is 8 bits long. Of course this is not a problem if we are simply hiding text messages, since this image would be 10.000 UTF8 characters without considering the header size, but one might want to hide files within the image. Whether it is text or files being hidden in the carrier media, compression is a very useful technology as it allows us to fit more data inside the image.

We can split this technology into two groups, lossy and lossless compression. In lossy compression, an algorithm finds redundant information and removes it. This is the same method used in the JPEG image format, where images are of worse, but acceptable, quality after compression. Lossless compression also finds redundant information but does not eliminate it. It utilizes statistical redundancy by analyzing repeating patterns in the information. For example, images often have repeating pixels. If we have 100 green pixels written as "green

pixel, green pixel, green pixel, ..." in the image, the compression algorithm can rewrite it to "100 **x** green pixel". This is partly what the DEFLATE algorithm does which is implemented in PNG among other file formats [18].

To conclude, one can use lossy compression for formats such as images and audio, where the loss of quality is insignificant. On the other hand, if one wants to recover information after compression, they would use a lossless compression algorithm like DEFLATE.

4.3 Carrier media

This section will serve as an overview of some potential carrier media and their pros and cons for the steganography tool. For every type of media, some of the most common file formats will be elaborated on.

Images

There are many different image file types to choose from, where some are easier to implement steganography on than others. In this section we will describe a few of the most popular image file types along with a general explanation of why images are subject to steganography.

All these image file types share some common principles, for example the use of pixels. Pixels are essential when talking about images and pixels are also what is used to hide messages in the images. The difficulty when hiding messages comes when altering the pixels, because the pixels are essentially the visual representation of the image and therefore a pixel cannot be altered too much in order to avoid detection.

Pixels are the smallest element in any digital image. They can be subdivided to make larger resolutions which means having more pixels within the same area. This would mean that each pixel would be divided into smaller pixels that take up the same amount of space on the screen as the pixels they derive from [19].

Pixels are a combination of channels. An example is the four pixel channel known as RGBA which is short for Red, Green, Blue and Alpha respectively. Each of these channels is one byte and can be changed independently of each other. The values of RGBA can each range from 0 to 255 where the alpha value is the transparency of the pixel. RGB is the brightness of the colors Red, Green and Blue respectively. RGBA is different from RGB in that RGBA consists of four bytes rather than the three bytes RGB uses [20]. This is very useful when the intention is to use steganography, as there will be additional bits you can use to store the data.

JPEG File Interchange Format (JFIF) is an image format that is a common standard in most modern digital capture devices such as digital cameras and image software alike. JFIF uses the Joint Photographic Experts Group (JPEG) compression method (for more info about compression see section 4.2). In JPEG you would achieve steganography by altering the Discrete

Cosine Transform (DCT) rather than the pixels themselves as you would do otherwise [21].

Portable Network Group (PNG) is another image format widely used in 8-bit palette images because of the support for transparency for every palette color and 24-48 bit truecolor with and without alpha channels.

BitMap Picture (BMP) is a format that is more focused around files within the Microsoft Windows OS. While BMP files are usually larger in file size due to commonly being uncompressed. Their advantage lies in simplicity and ease of use.

Graphical Interchange Format (GIF) is widely used to make small animations often to express humor as the GIF format can show multiple images in a sequence. However, GIFs have limited support for colors and transparency compared to other mentioned image file formats. For example GIFs can only use an 8-bit palette or 256 colors.

Based on all of this information we can conclude that the larger resolution, or the more pixels an image has, the larger of a message can be contained within. Our aim will therefore be to focus on larger image file types to prioritize data higher than file size. We can also conclude that using a format that supports RGBA is better because of the additional bits that can be used to store data.

Audio

Another media that can be used as a carrier media for hidden information, using steganography, is audio files. There exists a lot of different audio file formats such as the most popular MPEG Layer-3 and Waveform Audio File Format also known as MP3 and WAV [22].

Audio file types can be split into 3 categories: Uncompressed, lossless and lossy compression audio file formats [23]. The difference between these categories is the way the audio file can be compressed.

Uncompressed audio formats can not be compressed at all unless they are first converted into a format with lossless or lossy compression. The point of uncompressed audio formats is that no loss of data can occur which will make sure that the quality of the audio file will remain the same. This type of audio file will typically contain more data and therefore be larger than types with lossy or lossless compression. This is because the areas of the file that does not contain actual sound still contains the bits which represents silence. Because of the extra amount of data the audio file types with no compression can carry more data if exposed to a steganography tool. However, audio files with no compression is not as commonly used for Internet sharing because the format is much larger than lossy or lossless audio types. This will make it stand out in the data stream of the Internet seen from a steganographic perspective, thus making it less hidden compared to other formats.

Lossless audio files will have a smaller size than uncompressed audio files because it uses less data to represent silence and similar repeating patterns.

Lossy audio files takes the lossless principle a bit further by cutting some of the actual sound away. However, the sound taken away is not audible to the human ear and should, in theory, have the same sound quality as an uncompressed audio format depending on the type of compression.

There exists various methods for performing steganography on audio files such as LSB, echo hiding, spread spectrum, phase coding and parity coding [24]. The different methods will not be described in detail here. However, we will take a look at the advantages and disadvantages of the different methods.

Echo hiding has a high data transmission rate and very good robustness. Spread spectrum and phase coding both have high robustness rates but phase coding has very low data transmission rate where the Spread Spectrum method has a moderate data transmission rate. Parity coding also has very good robustness. However, these methods on performing audio steganography also have downsides to them. Echo hiding, spread spectrum and parity coding all have the same fatal flaw, noise pollution, throughout the spectrum in the audio file. The human ear is very sensitive to noise and will often be able to detect the slightest noise changes, within what is audible to human hearing, in an audio file thereby making the hidden data easily detectable by the human ear.

Phase coding is a viable method to use in steganography. However, it is only capable of hiding small amounts of data to avoid creating to much noise pollution in the file meaning that it would, for example, not be able to hide an entire picture in an audio file.

Using the LSB method will not leave a noise trace in the same scale as the other methods and it is capable of hiding larger data sets, such as images of a certain size, inside the audio file. The size of the file, after the steganography has been performed, will be the same size as the original size. This adds a bit to the obscurity of the data if someone was to compare the stego-file and the original file size. Furthermore steganography using LSB can be performed on any kind of audio file and it is the simplest method to perform steganography on audio files with. Due to the change in the bitmaps the audio quality will degrade slightly from using this method. However, if the audio file has a 24-bit bitmap the changes will not cause any significant changes to the audio quality since the audioquality is relatively low.

The LSB steganography method can be divided into four steps:

1. Convert audio file to a bit string.
2. Convert characters in the secret message to a bit string.
3. Replace least significant bits of the audio file with the bits of the secret message.
4. Convert bit string from step 1 back into an audio file.

In summarization, uncompressed or lossless based audio types will be more suitable as a carrier media compared to lossy compression, since they do not lose any actual data during the compression. The best steganographic method for hiding messages in audio files will be the LSB method, since it will leave the least amount of noise pollution in the audio file, it has a larger capacity than the other methods and it is the most simple method to utilize.

Video

Video files as a carrier media can be seen as a combination of multiple pictures in sequence often with one or more accompanying audio tracks [25]. This is the case for uncompressed video which consists entirely of separate, individually complete pictures, however, the size of such a video would be much larger than necessary since, generally, most frames differ little from the frames preceding or following it. Therefore most modern video is compressed to contain certain 'key frames' that are individually complete pictures, independent of any other frames, followed by a number of 'predicted frames', and/or 'bipredictive frames', that simply indicate how it differs from one or more already decoded frames.

As such video steganography offers many of the same possibilities for hiding data as previously discussed in section 4.3 and section 4.3 but also offers some new possibilities native to the video format [26]. As an example of the former, it is possible to simply treat each key frame of a video as a separate picture to perform steganography upon. This approach still offers certain new possibilities since the presence of possibly thousands of key frames means that you might be able to spread any changes to the pixels over multiple key frames making the changes to a single key frame insignificant.

The multitude of key frames in a video can serve to obscure changes to a single key frame though this requires some care since it is necessary to consider, not only how any changes would fit into the actual key frame, but also how it fits with the surrounding frames. Especially if changes are made to an otherwise static homogeneous area, the sudden change can become more obvious, and as such it is recommended to not make any changes to areas with change or motion between frames.

Other techniques could make use of the attributes of a the video compression codec such as motion vectors, which are used to decode predicted frames, or use the correlation between image and audio. These offer some intriguing possibilities, however, as of the writing of this paper limited research has been done in these areas.

Network steganography

Another method for using steganography is network steganography [27]. Network steganography uses another kind of carrier media which is the network connection itself. There exists different methods to perform network steganography. These methods can be put into three categories. Methods that modify the structure of packets stream, methods that modify packets and hybrids of those methods. Packets are the form that data is transferred through networks [28]. For example whenever a website is visited all the information from the webpage is transferred to the entering computer through a packet stream containing packets, which contain information about the website and the assets it uses.

However, using network steganography is deemed to be out of scope for this report since we want to make it possible to share stego-files on social media in order to be able to hide the stego-files in the vast amounts of shared pictures and audio files on social media, so that the stego-files stand out as little as possible, making it as hard as possible to detect or intercept by non-intended audiences.

4.4 Communication channels

In this section, we will review two possible communication channels with different pros and cons. By communication channels we are referring to the channel in which the steganography user transmits and receives hidden messages.

Social media

Social media such as Facebook, YouTube and Twitter has seen widely used in conflicts where governmental systems prohibits external contact beyond the borders of said government [29]. This is in spite of them not necessarily providing the option for people to share content or information anonymously which would otherwise prevent opposing parties of knowing the identity of their opposition [30].

Twitter has 500 million tweets (posts) every day and has previously seen wide usage in conflicts such as the one that is currently taking place in Syria. There, Twitter is being used for communication between activists and sympathizers from other countries in order to document events and highlights that otherwise would never have reached beyond the borders of the country in question [31].

Activist groups and the likes have also taken an interest in the social media YouTube. YouTube has been used by groups trying to forward their agenda. For example, a YouTube channel heavily 'linked' to the jihadi community has had over 6.5 thousand subscribers and 1.7 million views. This is in spite of YouTube not utilizing anonymous content [31].

Each month Facebook draws in 1440 million unique active users and is being utilized by activist groups such as the National Whistleblower Center which is a nonprofit organization providing legal help and the likes for people revealing wrongdoings within organizations to the public or those in positions of authority. Facebook, however, does not utilize anonymous content sharing and also compresses pictures uploaded with a lossy algorithm which would deter the use of steganography in pictures [31].

Reddit is, compared to Twitter and Facebook, not very popular, but it still has millions of, mostly Western, anonymous users [32]. It is divided into communities, henceforth referred to as subreddits, which are dedicated to a niche or area of interest. While some of these subreddits serve a variety of content, such as discussion topics and links to articles, others are restricted to image links. Since reddit does not host images themselves, the images

are hosted at a third party host, such as Imgur, meaning that the use of compression is up to the host but ultimately up to the user. Hypothetically, one could use reddit to have an entire board dedicated to steganographic images. The board could be innocent looking and nobody, except the steganography users of course, would know that covert communication was taking place.

In summation social media could be utilized by people worldwide for things such as whistleblowing and spreading their agendas. Though one needs to keep in mind that most social media are found wanting in the area of anonymous content sharing this could, however, be helped by utilizing steganography. Social media, such as Facebook, would not be suitable for image steganography due to the inherent nature of compression [33].

E-mail

Another possible communication channel is through the use of e-mail. A clear benefit about using e-mail is that users can attach files within a certain file limit without risking loss of data due to compression. However, it is easy to imagine what issues there might be in using e-mail as a communication channel. The point of steganography is to hide the existence of the hidden message which could be accomplished by attaching innocent looking images. In contrast to social media where there is a broader reach e-mail is single- or two-way communication. As such, a government spying on activists would easily be able to tell who they are communicating with and it would definitely raise suspicion. With that said, it is probably safer for two individuals to communicate using steganographic messages, rather than communicating in clear text, since it is not obvious to government agencies that covert communication is taking place.

4.5 Cryptography

Cryptography and steganography are closely related since steganography is a subcategory of cryptography. While both of these fields are concerned with sending secret messages steganography is concerned with the principle of 'hiding in plain sight' and hiding its own existence. One might want to hide the existence of a message, while still keeping the message unreadable, which is where a combination of cryptography and steganography is very useful. This section discusses two kinds of cryptography algorithms, namely public- and symmetric key, and what their strengths and weaknesses are in relation to steganography.

Public-key algorithm

The Rivest-Shamir-Adleman cryptosystem, henceforth referred to as the RSA encryption, is a public-key cryptosystem, also known as an asymmetric cryptosystem, which is used for transmitting data in a safe and secure way [34].

First published by Ron Rivest, Adi Shamir and Leonard Adleman in 1977, the RSA encryption uses a public-key system which allows encryption of data using a public encryption key which works independently from the decryption key. The data can then be decrypted by anyone who has the corresponding private key [35].

RSA encryption also makes use of prime factorization and the difficulty of figuring out which two numbers multiplied has been used in order to encrypt the message or data in question [34]. The typical encryption key size for RSA encryption is between 1024 and 4096 bit. This will become important in future discussions for and against the use of asymmetrical or symmetrical encryption.

In steganography, one strives to conceal data within other data, where cryptography is trying to hide data in plain sight by making it unreadable. This does not mean that these are mutually exclusive to each other as cryptography can be performed on a plain text message before embedding.

Especially when using steganography the user might be interested in encrypting the plain text data using steganography in the event that the data falls into the wrong hands. This could be achieved using two different RSA encryptions with two agreed upon public encryption keys.

The issue with public-key cryptography, when compared to symmetric-key cryptography described in section 4.5, is that the key is in a format that is not very readable for a human. On the other hand, if one were to capture a public-key transmitted between two users that would not be an issue, but the capture of a symmetric key could compromise the hidden data. Public-key cryptography is more beneficial in some cases. For example, when confirming the authenticity of signed applications.

Symmetric-key algorithm

The principle behind the symmetric-key algorithm is that it utilizes the same key to encrypt and decrypt [36]. This means that both the receiver and the sender uses the same key and that both the encryption and decryption requires the same key to work. This method has five components:

1. **Plain text:** The plain text is the original message the sender wants the recipient to receive.
2. **Encryption:** The plain text gets encrypted using a encryption algorithm taking a secret key as input.
3. **Secret key:** This is the key that is used in the encryption. Changing the key will result in a different output results.
4. **Cipher text:** The encrypted text, referred to as cipher text, is transmitted to the recipient.
5. **Decryption:** The cipher text gets decrypted using a decryption algorithm that will take the same key as input.

Of course the encryption algorithms are not limited to plain text as this was just an example in the context of steganography. Since there is only one key, both the sender and the recipient have to know the key. This means that the

key has to be distributed to all the recipients which may prove to be equally as difficult as sending the message [37]. An example of this method is Caesars Cipher where the key is the number of shifts to make. This means that if the key is 3 the letter 'A' would equal 'D' after the encryption algorithm so:

Plain text: This is the plain text

would equal

Cipher text: Wklv lv wkh sodlq whaw

Using the key in the decryption algorithm would have the plain text as output. This means that if the message is intercepted it would be useless since only the generals knew the key [38].

A symmetric-key only have to be between 128- and 256-bits long to be secure compared to the asymmetrical key which require a key between 1024 and 4096-bit [39]. The reason for this is that the asymmetric cryptosystem needs to do some form of mathematical calculation to generate the key. This means that even though it has more bits, there are not as many possible keys while the symmetric-key can have all possible key combinations.

When discussing steganography and cryptography, it should be considered that symmetric-key encryption allows for a human readable key which could be passed on to decrypt the hidden message also it can be shorter and easier to memorize if this is necessary.

Stream and block cipher

The block cipher consists of two algorithms: The encryption algorithm and decryption algorithm called block encryption and block decryption [40]. It uses a symmetric-key and a plain text input with a fixed length and turns it into cipher text of the same length.

The stream cipher works in a very different way than stream cipher [41]:

> "*Block ciphers* operates with a fixed transformation on large blocks of plain text data; *stream ciphers* operate with a time-varying transformation on individual plain text digits." [41]

This means that the stream cipher encrypts each individual bit that the plain text consists of, and that it can handle an input of variying sizes, in a stream of data. However, using a block cipher algorithm enables it to repeatedly use the block cipher on data structures bigger than the block [42]. The stream cipher has two advantages: one being that it takes less computation to achieve the same security level as a block cipher. Another being the fact that the encryption and decryption happens very fast with a pre-computing keystream [43]. The block cipher can be more powerful than a stream cipher, meaning more security, but uses a lot of memory and time [42].

The stream cipher is better for a small amount of data that is required to be fast but not necessarily as secure. We also have the block cipher which would be more secure but require more memory and time. On the other hand, block

cipher encryption should be used when working with a fixed length block of plain text and the time complexity is not important.

4.6 Existing solutions

In order to develop a steganography software solution it is useful to review existing solutions. The solutions will be analyzed based on their carrier media, cryptography and carrier engine that could, for example, be the algorithm that is used to encode data. As opposed to other types of software it is not easy to give each tool a market share. This is because it would go against the principles of steganography to announce the use of it to anyone. As such, these reviewed tools are somewhat randomly selected and not in order of popularity.

DissidentX

DissidentX is an example of a newer steganographic tool developed by the developer of the BitTorrent protocol. It is written in Python and available on GitHub but no license is stated [44]. As the name indicates this tool's intended audience is political dissidents. This is in contrast to the other tools being reviewed in this section as most steganographic tools seem to have no intended audience. Additionally, it is open source and is built in a way that encourages encoder extensions.

It is quite unique compared to other tools as it implements some new technologies among old ones:

- Like in many other steganographic tools the message cannot be decrypted without the key. It uses symmetric encryption as the author deems human readable keys to be best fit.
- It has a single decoder for all file types. The author claims that all future encoders are supported by this decoder.
- Format-specific encoders can easily be implemented on top of the solution. For example, the author has added line endings encoding.
- Multiple messages with different keys are supported. This is meant to counter 'rubber hose decryption' where the secret key is extracted by force from the person knowing the key.

The author has stated that the optimal use of this tool would be to implement it in a browser. That way images, CSS files and other static content could be checked for messages hidden with DissidentX on run-time. This is not implemented in the current version of the tool.

Steghide

Steghide is an open-source steganography tool developed in C++ and released under the GPL license [45]. It supports compression and encryption of data using zlib and AES. It implements a hash, using the CRC32 algorithm, which verifies the integrity of the data. This means that the messages cannot be edited by a third party without them updating the hash as well. The tool supports JPEG, BMP, WAV and AU formats as cover files.

The tool uses a graph-theoretic approach which is implemented in the following way:

- The secret data is compressed and encrypted.
- A sequence of positions of pixels is created based on a pseudo-random number generator.
- The algorithm iterates through these pixels and check if their value corresponds to the hidden data. If not, then the graph-theoretic algorithm finds an existing value that it can be interchanged with.
- Remaining pixels that do not have a possible replacement are overwritten like in the LSB method.
- For audio files the principles are the same except audio samples are used instead of pixels.

PNGDrive

PNGDrive is an open-source steganography tool written in JavaScript released under the MIT license [46]. It allows users to inject both text and binary files into PNG format images. It uses the LSB method to write information to the image where it stores data in the RGB channels of each pixel. If there are four channels, which will happen if the image has an alpha channel, the value of this channel is set to 0xFF, which is the max byte value, for every pixel. This means that all the stego-images produced by this tool will be opaque, also known as non-transparent, even if they were transparent before. The author states that this is due to limitations in the Canvas API.

4.7 State of the art

State of the art steganographic algorithms seek to uncover more secure ways of hiding data or hiding significantly more data in the available data structures. This is a continuous battle between steganalysis and steganography that continuously requires better software from both parties.

One such advance has come quite recently in form of an algorithm documented by V. Sedighi, R Cogranne and J. Fridrich in 2016 [47]. This algorithm is meant to adapt steganographic content to minimize statistical detectability. This is done mainly by "swapping" pixels with already existing pixels to fit the bit pattern of the intended message. This way it is effectively making the need of altering bit patterns in the data structure much less relevant than previously. By applying this algorithm the need for altering bit patterns is reduced to three percent thus making it much harder to detect whether a data structure has been tampered with [47].

Using this method is a significant improvement from older methods like LSB which is highly susceptible to statistical detection methods like χ^2 because it changes bit patterns. This is significantly reduced with the new algorithm as shown in [47] because this algorithm does not change bit patterns but rather changes the position of pixels in the data structure.

Thereby information can be stored and sent safely in data structures without risking detection. However, this method does not encrypt what is written in the data structure and thus if the algorithm, which is public domain, should be discovered the information is easily extracted from the data structure. This could be alleviated by encrypting the data before it is placed in the data structure hence making this a very safe method of communication as of the most recent research.

5 Problem definition

In our examination of the various parties interested in the use and development of both steganography and steganalysis, we have made the decision that activism will be the main course of inquiry from this point forward as this is somewhere we judge that a potential steganographic tool would be able to serve as an intermediary channel of safe communication. However, developing safe communication tools for activists may result in misuse of the said tool. It is deemed as a necessary evil that cannot be avoided without removing anonymity from the tool. Having a focus on activism will require that the programs interface is intuitive and easy to use since activist are not necessarily experienced IT users.

To make the tool more versatile it is in the best interest of the development focus to implement the program to be able to function with social media. Such social media as Twitter, Facebook or Reddit which have a very large amount of pictures flowing through them on a daily basis.

As an example, Facebook has more than 300,000,000 pictures shared on a daily basis and we have decided that pictures is the optimal choice of carrier media for a steganographic tool. However, for added functionality it would be purposeful to develop a multi-functional tool that can also embed messages into WAV (audio) files. This is to be implemented to increase the diversity of the program and give the user more options in how to hide their message.

Due to the nature of steganography there is a limitation on the capacity available in the cover media, otherwise referred to as the bandwidth. We have figured out that by using compression of hidden messages we can effectively increase the bandwidth without losing valuable information. As such, we can conclude that we will be using compression in our solution.

While steganography focuses on a hidden message escaping detection to remain secure there still exists the possibility of detection and interception. In these cases steganography offers little inherent impediment against the message being read. Encryption serves to make a message unreadable in case of interception but does nothing to obscure the existence of a message and, in fact, often makes the message stand out. We will therefore combine the methods by first using encryption methods on our message to be hidden and then embedding it in a carrier media with steganography. In this way we seek to achieve a message that is both hidden from detection but also secure in case the camouflage fails and it is intercepted. We choose this despite the complications it will cause for the sharing of keys.

Our analysis has led us to the following problem definition:

How can we allow activists secure and covert communication, through public channels, with intuitive interface and versatile carrier media selection. Namely:

- How can we ensure message imperceptibility against known statistical analysis methods?
- How can users safely exchange cryptographic keys?
- How can an intuitive interface ensure a broad userspace?

Part II

Solution

Design 6

In this chapter we are going to delve further into steganography and its inner workings. Where as previous chapters have described technology regarding steganography in broad terms this chapter contains the result of all our considerations as well as the solutions to the problems we found regarding steganography.

The purpose of this chapter is therefore to inform the reader as to what our plan of implementation and design choices are going to be and our reasoning behind implementing the features in question.

6.1 Criteria

In order to evaluate a steganographic tool it is necessary to establish which criteria such a tool should be evaluated against. Our choice is to define four different criteria as our main focus:

- Bandwidth
- Imperceptibility
- Time complexity
- Robustness

Other secondary criteria we have deemed less important include, but is not limited to:

- Versatility of carrier media selection.
- Popularity of the compatible carrier medias.
- User interface ease of use.
- User security.

The main criteria will be explained below:

Bandwidth

In order for a passed message to be meaningful it requires a certain length. While a simple yes/no which could be represented by a single bit might suffice for some messages, a more sizable message is often needed and could, for example, be a word, a sentence or entire files. As such it is important how much data can be hidden in a carrier file by the steganographic technique. Thus bandwidth is defined as the amount of data that can be hidden relative to the size of the carrier.

Imperceptibility

The point of steganography is to hide the existence of the hidden message. It is therefore important that it, ideally, should not be possible to detect the changes made to the carrier file or at least be difficult to do so. This criteria can be further split according to the two major means of detection:

- Human imperceptibility
- Statistical imperceptibility

Human imperceptibility is the hidden messages ability to evade the human senses, for example sight, hearing or both, while statistical imperceptibility is the messages ability to evade statistical analysis.

Time complexity

While the effectiveness of the technique is important so is the efficiency. A technique capable of achieving complete imperceptibility for massive messages, while ensuring total robustness, would still be useless if said technique would take years for each message to be embedded or extracted. Thus it is significant to consider how complex the used algorithm is. Time complexity is thus defined as the amount of computational operations that needs to be carried out to embed a message in the carrier file.

Robustness

It is important that once a message has been hidden in a carrier file it can then later be extracted. As such it is important that the hidden message can survive the journey from sender to recipient. Robustness is then defined as the survivability of the message regarding conversions, compressions or transfers that might occur between sender and recipient.

6.2 Graph-theoretic approach

Graph theory is the study of graphs which are mathematical structures used to model pairwise relations, called edges, between objects called vertices. In simpler terms, graph theory can compare any two parts of a whole, and establish relations between these parts. This means that it is ideal for comparing either pixels in a picture or waves in an audio file. This can be used to examine and exchange different bit patterns of any given file, and make exchanges between vertices instead of having to rewrite the bit pattern as done in LSB. Thus we reduce the introduction of new unique bit patterns, and so both decrease the chance of detection by statistical analysis and prevent excessive inflation of the carrier media file size which is consistent with our criteria of imperceptibility.

Some of the inherent properties of using graph theory for steganography is shown in S. Hetzl and P. Mutzel's paper "A Graph–Theoretic Approach to Steganography" [48] in which they state:

"1. It does not depend on the type of the cover data (e.g. image, audio,...).
2. It is easily extendable concerning the question which exchanges are allowed by defining additional restrictions on the set of edges. This allows for modular addition of visual and statistical criteria to the embedding algorithm.
3. It reduces the problem of finding a steganographic embedding to the well investigated combinatorial problem of finding a maximum matching in a graph." [48]

Even though graph theory does give us a variety of ways to improve on our overall algorithm design, it does not give the algorithm total imperceptibility to steganalysis and as such it is still detectable although it is a lot harder to detect than the LSB method which does not utilize graph theory.

The actual graph theoretical method we intend to use will be explained by the following example:

Graph Theory Example

Suppose you have a picture with 24 pixels with colors corresponding to the values in row 2 in section 6.2. If you take mod4 (modulo 4) for each of these values (section 6.2, row 3), add the remainders together in pairs and take mod4 for the sum again (section 6.2, row 4), you now have each pair of pixels representing a value from 0 to 3 corresponding to 2 bits and as such a total of 24 bits or 3 bytes represented by 12 pairs of pixels. With 3 bytes available for embedding we can hide 3 characters assuming standard ASCII character values.

Pixel number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Color value	65	198	45	46	58	46	48	48	94	65	41	68	138	44	165	46	85	215	61	89	47	198	51	65
Single mod4	1	2	1	2	2	2	0	0	2	1	1	0	2	0	1	2	1	3	1	1	3	2	3	1
Paired mod4	3		3		0		0		3		1		2		3		0		2		1		0	
Paired goal	1		0		3		1		1		3		0		3		1		2		1		3	
Single new	3	0	2	3	1	1	1	1	0	3	3	2	0	2	=		2	0	=		=		2	0

Table 6.1. Graph theory example.

In this case we want to hide the message "Msg". To do so we must first convert the bit pattern for these characters into 12 values from 0 to 3 which can be accomplished by simply splitting the bit pattern into 2 bit chunks. The values for these 2 bit chunks of the bit pattern for "Msg" can be seen in row 5 in section 6.2.

With simple comparison between row 4 and 5 we see that the vertex values corresponding to pixels '15&16', '19&20' and '21&22' already correspond to the desired bit pattern. For the rest of the vertices we need to find which mod4 value for each of the vertex pixels that would make that vertex correspond to the correct value, as can be seen in row 6 of section 6.2.

We will then compare the different vertices, looking for pixels that have the desired value for the opposites. For example, pixel 3 has a mod4 value of 2 while a mod4 value of 1 is desired and pixel 6 has a mod4 value of 2 but a mod4 value of 1 is desired. If these pixels were swapped we would get two additional vertices with correct values without having changed the color palette at all. This relation between different vertices is called an edge.

In Figure 6.1 all the possible edges are represented. As seen, the vertices have a varying amount of edges and though in this case none has 0 edges not all vertices could use an edge. Since only a single exchange can be performed per vertex either vertex '3&4' or '17&18' would be left without valid edges once the edge between vertex '5&6' and either of the prior vertices was applied. This is in part caused by the small size of the pixel sample as more pixels, and thus more vertices, would give more possibilities for edges and with larger pictures it would be increasingly likely that all pixels would have at least some edges and thus making it possible that no outright changes would be necessary.

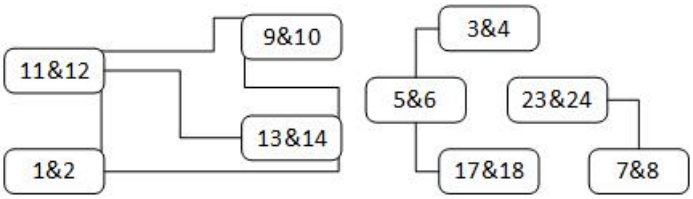


Figure 6.1. Graph theory edges.

6.3 File archive

In a style similar to that of PNGDrive, described in section 4.6, our solution should support the ability to hide files of any format in the carrier media. With an additional implementation of folders and compression our solution could serve as an alternative to the likes of ZIP and RAR files. Both of these file types are archive file formats that contain one or more files or directories and permits lossless compression algorithms such as DEFLATE. Of course this functionality should not limit the use of text messages as it should still be possible to add text messages even if files are embedded. This file archive should support some basic file operations such as importing, exporting, renaming and editing.

6.4 Digital signing

When creating a stego-file the user should have an option to sign files with a digital signature. This feature should be optional and left only to advanced users as it, to some extent, reveals who the sender is. This is an undesired outcome when using steganography because of the anonymity dilemma. The file should be signed by using asymmetric cryptography in which the sender signs the file with a private key and the recipients confirm the authenticity of the message by validating it with the sender's public key. The use of

digital signatures raises another issue as it would be difficult for the sender to deny ownership of any messages sent with their signature in case their private key is compromised [49].This is another reason why the feature should be left optional.

6.5 Multiple encryption keys

The solution should have the ability to support multiple encryption keys that unlocks different content. This feature could be used as a decoy key. For example something that can mislead or distract. It could especially be useful to activists who may be forced to provide decryption keys in case of interrogations. In this scenario the interrogated would be able to give away the decoy key which would only show the decoy message also hidden in the same file. This method could also be used to send multiple messages in one stego-file so that individuals with different keys could retrieve different data. The content contained by the primary key should also contain the additional keys in order to prevent data from being overwritten when the stego-file is re-embedded by using the second key. Adding this feature to the program will help achieving the secondary criteria 'User security' established in section section 6.1.

6.6 Use of cryptography

In order to further the security and imperceptibility of communication through public channels, certain cryptographical decisions need to be made in order to clarify the pros and cons of symmetrical and asymmetrical encryption.

- **Symmetrical encryption:** Symmetrical encryption uses the same key for encryption and decryption. This poses a certain risk for the parties involved in using the program seeing as both parties could unintentionally leak the key. Additionally, if covertly transmitting the encryption key was easily achieved there would be no need for a program providing covert transmission.

  Another thing to consider is the size of the key needed for encryption. Typically the size of the encryption key in symmetrical encryption is smaller than that found in an, equally hard to brute force, asymmetrical encryption key as stated in section 4.5.
- **Asymmetrical encryption:** In general asymmetrical encryption requires a larger encryption key than that of a symmetrical encryption in order to be as hard to brute force as a symmetrical encryption. Furthermore because asymmetrical encryption such as RSA encryption takes advantage of a public encryption key and a private decryption key. This enables the use of non-covertly distributed encryption keys and in turn the transmission of data encrypted by the before mentioned public encryption key. The issue with asymmetrical encryption is that, as opposed to symmetrical encryption, it typically has a very large and humanly unintelligible key. With symmetric encryption the choice of key would be up to the user.

In general an as small encryption key as possible without compromising the security of said key is wanted. This helps ensure that the encryption key, and in the case of symmetrical encryption being the decryption key as well, is easy to memorise. The key should also have the possibility of being in a totally inconspicuous format. For example the sentence "My dog ate a steak." could be, or contain, a symmetrical encryption key in a simple string format. However, some users might want to use asymmetrical encryption and therefore it should available as an advanced option.

6.7 Supported file formats

The following section will discuss the pros and cons of different file formats as carrier media, which file formats our program should support and to which extent they should be supported.

Compression

Since steganography in image- and audio files mainly consists of changing the bytes in the photo or audio file and extracting changes, it would not be wise to use file formats that use lossy compression. Uncompressed formats are usually straight forward to read and write changes to. For example in the case with WAV and BMP.

Uncompressed file formats can be good if the intention is to keep the data exactly the same way as it was made, while not particularly caring about the size of the file. And the associated downside to this would be that the file has a larger filesize than a lossless compressed file which also keeps the data intact.

In the case of some formats, such as WAV, the raw sample data is found in a sequence followed by some headers and making small changes to these bytes will not lead to big changes in the actual samples.

Our intention is to ensure that the program can, at the least, handle uncompressed files, and depending on the ease of implementation possibly lossless and lossy compression formats as well.

Images

In PNG the principles are largely the same as WAV with headers encompassing raw data, except that the raw data is compressed and will need to be decompressed before changes are made. In addition, making changes to PNG is trivial in C# as there are built-in classes which automatically decompress the data and present it as either pixels or raw data. These file formats are in contrast to files using a lossy compression where the implementations are vastly different. For example JPEG, which is lossy compressed, has a quite complex implementation which uses discrete cosine transform (DCT) values in an attempt to achieve very small file sizes. This would make it difficult, but not impossible, to implement in our solution.

We estimate that the implementation of some uncompressed file formats can be done within a reasonable timeframe. As JPEG is a widely used format, especially on social media, it would make sense to attempt to implement it, but it may turn out to be difficult to implement. In any case, our solution should support the importation of JPEG files, but not necessarily the exportation/recompression of them. Using lossy compression will also alter the robustness of the message since some of the hidden data might be removed when the image is compressed.

Audio

WAVE files are a common file format used to store audio files [50]. The WAVE format has the ability to contain every other audio file format. This also means that WAVE files are a really big audio file format. The WAVE format is also an uncompressed audio format which increases the robustness of the steganography message that eventually will be stored in the file after we hide information in it by using steganography. Because the WAVE file is a large format meaning it has a high amount of bits, compared to other lossy or lossless audio file types, it also has a high bandwidth which fits with our capacity requirement.

A WAVE file is composed by different data chunks containing different data where some contains metadata and some containing the actual sound data [51]. More different kinds of chunks exist and can be used to different activities but the metadata chunk and the actual audio data chunk is the important chunks for us. The different chunks is always in a specific order. The first chunk of a WAVE file contains metadata and is known as the format chunk. Thereafter comes the next important chunk which is the audio data chunk.

The format chunk contains metadata about the chunk itself needed to specify the WAVE format which is:

- Which chunk it is.
- Length of the chunk in bytes.
- That the data is PCM (Pulse Code Modulation) format. PCM is used to encode analogue sound digitally.
- Number of channels in the audio.
- Sampling rate.
- Number of multichannel audio frames per second which is used to estimate the memory needed to play the file.
- Number of bytes in a multichannel audio frame.
- Bit depth of the audio samples which is the number of bits per sample which can be 8-bit, 16-bit or 32-bit.

The actual sound data chunk is much more simple than the format chunk as it only consists of the data to which chunk it is, the size of the chunk and the actual sound data. A thing to be aware of here is that the type of the array holding the sound data will vary depending on the bit depth.

- 8-bit is stored in a byte array.
- 16-bit is stored in a short array.
- 32-bit is stored in a float array.

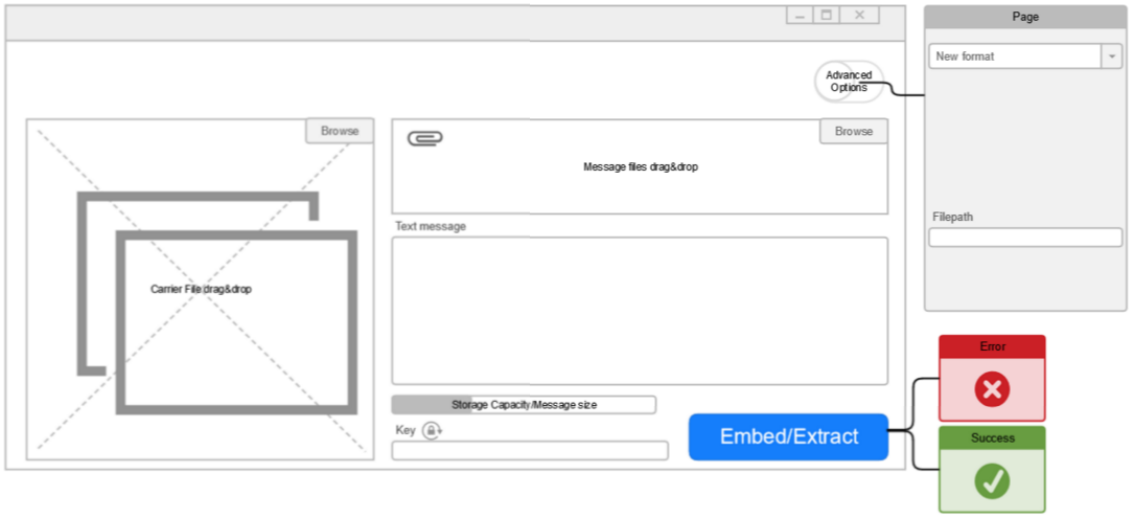
of which all elements are signed so that it can be positive, zero or negative elements.

All in all the WAVE format will be the format we are going to use as audio carrier media. Using the WAVE format will make the program easier to use because the WAVE files are as big as they are. The size of the WAVE files also gives opportunity to hide larger amounts of data meaning it has a very good bandwidth. However, the WAVE format is only somewhat common to share on the Internet depending on where you are sharing it since there exists different websites designed to share audio in different kinds of formats including WAVE files such as wavsource.com, wav-sounds.com and others.

6.8 Graphical User Interface

Since our software solution has a broad audience of users who may or may not be experienced with this type of software it is important that there is an intuitive and simple Graphical User Interface (GUI). This helps us achieve one of the less important criteria 'User interface ease of use' established in section 6.1. For the same reason, it should also be possible to edit advanced settings.

Figure 6.2. Proof of concept for the graphical user interface.



As seen in Figure 6.2 which shows a proof of concept for our solution, the application requires very little input from the user:

- **Carrier media:** Since this is a steganography application some type of media is required. The user should have the possibility of dragging carrier files from their desktop into the application in order to increase ease of use.
- **Files (optional):** The user should have the ability to easily embed files into the carrier media. They should also be able to drag and drop files from their desktop.
- **Message (optional):** If the user chooses not to embed any files, or if they simply want to add a message to the embedded files, they should have the ability to do so.

- **Key (recommended):** Users can choose to encrypt the embedded data using a clear text key. Using a key is optional but highly recommended and the program should warn users who choose not to use a key and those who use a short key. One possibility is to add a bar which shows the strength of the key.

As previously mentioned, a button should activate an interface which offers advanced settings to the user. This button should trigger a one-time warning which alerts users that they should not edit the advanced settings if they do not know what they are doing. This advanced settings section could offer options such as selection of cryptography and steganography method, but it could also offer additional features. One such feature could be the possibility of adding a so-called decoy key which the user can disclose in situations such as interrogations as previously mentioned in section 6.5. Another feature could be the integration of a communication channel such as e-mail or social media.

For optimal user security the interface should have a built in guide, in addition to the warnings, which guides the user in proper ways of using the software and sharing the stego-file created with it. This feature will also add to achieving the 'User interface ease of use' criteria.

7 Implementation

This chapter will focus around the implementation of the features described in the previous chapter. There might be small alterations from the original design as the process of implementing the functionality of the program has shown us some ways in which we could increase performance, versatility or the fact that not every design decision has been prioritized equally. This will show the reader how we implemented the design choices that we made. To show how this is done the chapter will contain code examples from the program solution as well as a description of its functionality and the purpose of structuring the code shown. We assume that the reader understands how the C# syntax works and has a basic understanding of C#'s libraries.

We have decided to use **Stegosaurus** as the name of the program. There may be references to that name in this chapter.

7.1 General program flow

This section aims to visualize the flow of the program. As such, both the embedding and extraction processes will be explained. Common to both flowcharts is that the dashed arrows are conditional but the solid arrows will always be followed.

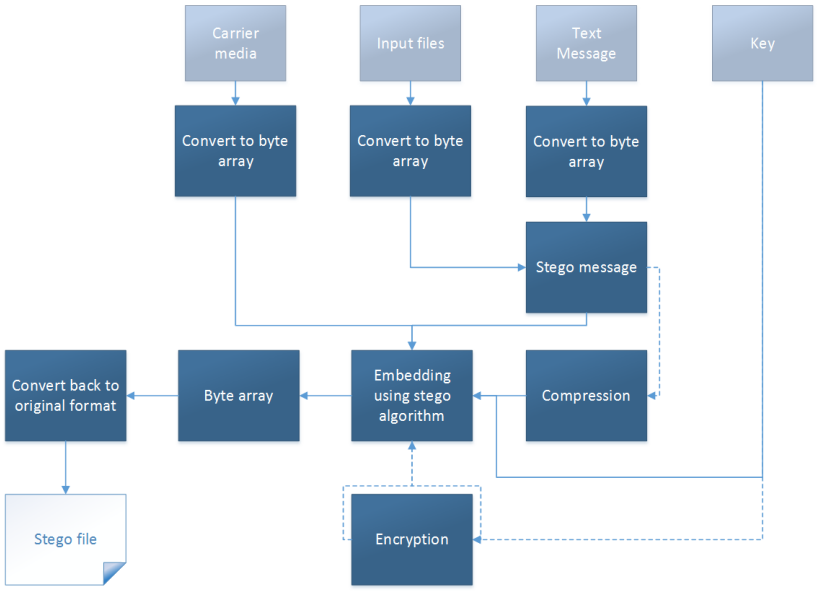


Figure 7.1. Flowchart of embedding process.

In the flowchart seen in Figure 7.1, the embedding process is visualized. The first step in the embedding process is to convert both the carrier media, the input files and the text message into byte arrays. The byte arrays created from the input files and the text message are then combined in an instance of the *StegoMessage* class (described in section 7.3). Next step is to figure out whether or not the compression of *StegoMessage* is appropriate, in which case, it is then compressed. Compressed or not the *StegoMessage* is then embedded within the byte array representing the carrier media (described in section 7.6). Finally the resulting byte array is converted into the original carrier media format, which now contains the steganographically embedded input files as well as the given text message.

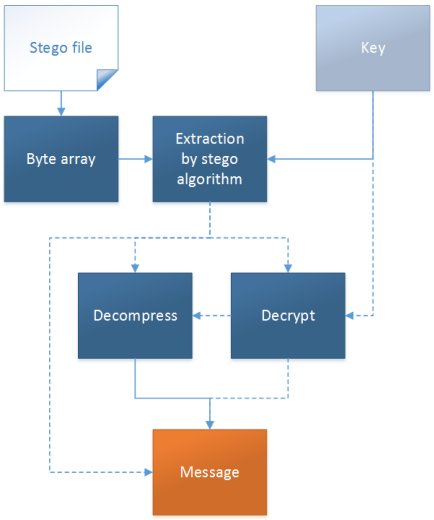


Figure 7.2. Flowchart of extraction process.

In the flowchart seen in Figure 7.2, the general extraction process is visualized. The extraction process is more or less the reverse of the embedding process, where the stego-file is first converted to a byte array and extracted using a steganographic algorithm. Afterwards, based on flags in *StegoMessage*, the data is decrypted and/or decompressed resulting in the message hidden in the stego-file. In this figure all routes leading to the 'Message' box will be dashed at some point, however one route will always be chosen depending on the flags previously mentioned.

7.2 Utility

This part of the implementation covers some of the implementations we have made in regards to utility features. What these features and classes have in common is that they are public and/or static, meaning that they can be accessed from everywhere and mostly do not require instantiation.

7.2.1 StreamExtensions

StreamExtensions is a public and static class containing extension methods for the *Stream* class. Since a number of classes inherit from the *Stream* class such

as the *MemoryStream* class, the extension methods will work for the derived classes as well. The purpose of these methods is to improve readability of the code and reduce code redundancy. For example, let us say we want to write the integer '5' to a *MemoryStream*. Without our extension methods this could be done in the following way, where *memStream* is an instance of *MemoryStream*:

```
1 memStream.Write(BitConverter.GetBytes(5), 0, sizeof(int));
```

Code fragment 7.1. Write the integer '5' to a MemoryStream without extension method.

While this solution works it is not very readable and this is especially true when writing multiple integers at once. To reduce complexity we have implemented the extension method *Write* which is an overloaded method that can write a variety of types to a stream. This also ensures that the written data is type safe so the correct length of different types is used:

```
1 public static void Write(this Stream _stream, int _value)
2 {
3    _stream.Write(BitConverter.GetBytes(_value));
4 }
```

Code fragment 7.2. Extension method *write*.

To make the extension methods as intuitive as possible there is a corresponding *Read* method for every type that can be written to the *Stream*. For example the method *ReadInt* reads the size of an integer from the current position in the *Stream* and returns the converted integer:

```
1 public static int ReadInt(this Stream _stream)
2 {
3    return BitConverter.ToInt32(_stream.ReadBytes(sizeof(int)), 0);
4 }
```

Code fragment 7.3. Extension method *ReadInt*.

We have also implemented extension methods which can read and write some of our own classes. For example for the class *InputFile* which is a simple class containing two properties. In the *Write* method which is seen in Code fragment 7.4, three calls are made to other variants of the *Write* method:

```
1 public static void Write(this Stream _stream, InputFile _value)
2 {
3    // Write name
4    _stream.Write(_value.Name);
5
6    // Write content with length
7    _stream.Write(_value.Content.Length);
8    _stream.Write(_value.Content);
9 }
```

Code fragment 7.4. Writes *InputFile* class to Stream.

First of all the *Name* property of the *InputFile* is written to the stream. This is done with another extension method which writes the length of the string

and its byte array in UTF-8 format. Afterwards, the length of the file's content is written as well as the actual content which is a byte array.

7.2.2 RandomNumberList

The class *RandomNumberList* is used to generate a set of unique integers within a specified range. In its constructor two integers are passed: a seed and a maximum value. The seed is then used to generate a private *Random* object. The maximum value determines the range of the unique integers, which is between 0 and the maximum value. When an instance of the class can be constructed the next integer in the sequence can be called using the property *Next*:

```
1 // Generate an integer which has not yet been generated.
2 do
3 {
4    generatedInt = random.Next(maxValue);
5 } while (!generatedIntegers.Add(generatedInt));
6
7 return generatedInt;
```

Code fragment 7.5. The property Next generates a unique integer

As seen in Code fragment 7.5, the property returns an integer that has not been returned previously and throws *RandomNumberOutOfRangeException* if there are no more possible values. For example, if one wants to generate a set of integers with a maximum value of 500, there are 500, in the range of 0-499, possible integers and requesting an amount of 501 would result in an exception.

7.2.3 IconExtractor

In order to make the GUI more intuitive the utility class *IconExtractor* has been implemented. The purpose of this class is to dynamically extract icons from their associated file type as there is no functionality in the .NET Framework to do so. This adds the ability to show the user the icons of the files to hide. The main functionality is found in the static method *ExtractIcon* which, given a file extension, returns an instance of *Icon*. It does so by calling the native Windows function *SHGetFileInfo*.

7.3 StegoMessage

StegoMessage is a public class used for handling in- and outgoing steganographic messages. Methods in this class are meant to handle these messages by either encoding or decoding the information that the user sends or receives. It uses functionality from other classes which makes the implementation of other compression or encryption algorithms very easy and improves the readability of the code. The class is therefore very versatile and can be changed upon without big changes to the structure of the overall program layout.

To avoid unnecessary computations in the program we have implemented flags to identify whether or not the information is encrypted, compressed, encoded or signed. This makes it possible for the program to identify the state of the data in the *encodedData*. The constructor then performs a series of checks to identify the necessary operations on the data received through *_fromArray*.

A check is performed to verify the hash of the *encodedData* meaning that the extracted data is, with nigh certainty, the same as the original data after it was encoded. This is done by reading the hash following *encodedData* and creating a new hash of *encodedData*. It then verifies that the two hashes are the same. If they are not, an exception is thrown to stop the process as the data in the message may be corrupt.

Now that we know that the intended data is present in the *encodedData* we can check if the data in *encodedData* is encrypted or not by checking the before mentioned flags and making sure that an instance of *ICryptoProvider* is sent with the information so it can be decoded with the proper key.

Two checks are made to verify that the information is encoded or compressed. The flag indicating that data has been encoded should always be set, so this is simply an added safety measure. Additionally a flag indicates whether the data has been signed, in which case the constructor attempts to find the signer from a list of known public keys, as seen in Code fragment 7.6.

```
1  byte[] signedData = inputStream.ReadBytes();
2
3  RSAProvider rsa = new RSAProvider();
4  foreach (SavedPublicKey key in PublicKeyList.GetKeyList())
5  {
6     // Set the RSA key to the current public key
7     rsa.SetKey(key.Key);
8
9     // Check if hashed data matches
10    if (!rsa.VerifyData(encodedData, signedData))
11    {
12       continue;
13    }
14
15    // Set sign state and who signed it
16    SignState = StegoMessageSignState.SignedByKnown;
17    SignedBy = key.Alias;
18 }
```

Code fragment 7.6. Signer of data is found

Decryption of the data is done outside of *StegoMessage*, in one of the classes inheriting from *ICryptoProvider*. After the decryption the data is decompressed if the compression flag is set and is now representing the original input in the form of a byte array.

This byte array is passed to the *Decode* method (see Code fragment 7.7) where the information is converted to a *MemoryStream* and the information is sorted

into the *TextMessage* string and *InputFiles* list where it can be accessed by other parts of the program.

```
1  private void Decode(byte[] _byteArray)
2  {
3     using (MemoryStream tempStream = new MemoryStream(_byteArray))
4     {
5        // Read input files
6        int numberOfFiles = tempStream.ReadInt();
7        for (int i = 0; i < numberOfFiles; i++)
8           InputFiles.Add(tempStream.ReadInputFile());
9
10       // Read text message
11       TextMessage = tempStream.ReadString();
12    }
13 }
```

Code fragment 7.7. Decode method in StegoMessage

The *ToByteArray* method translates the information from the various inputs in the GUI into a byte array that is used throughout the rest of *StegoMessage*. The information is first encoded and then compressed and the proper flags are added to identify the actions taken in the method. After this, the information is combined into a single array as shown in Code fragment 7.8. The data is combined in an order which corresponds to the data representation in the constructor.

```
1  // Do not write at the beginning of the stream. Allocate space for
       int and byte.
2  tempStream.Seek(sizeof(int) + sizeof(byte), SeekOrigin.Begin);
3  tempStream.Write(encodedData, true);
4  tempStream.Write(encodedData.ComputeSHAHash(), true);
5
6  // Sign if private key is specified
7  (...)
8
9  // Go back to beginning of stream and write length and flags.
10 tempStream.Seek(0, SeekOrigin.Begin);
11 tempStream.Write((int)tempStream.Length - sizeof(int));
12 tempStream.WriteByte((byte) Flags);
13
14 return tempStream.ToArray();
```

Code fragment 7.8. Data is combined ToByteArray

7.4 Algorithms

The purpose of this section is to describe how the steganographic algorithms have been implemented. This includes the base class specifying what methods are needed in order for a full algorithm object to be instantiated and used in our program. Classes derived from *StegoAlgorithmBase* are furthermore described.

7.4.1 StegoAlgorithmBase

StegoAlgorithmBase is a public abstract class with the purpose of creating a base for all compatible algorithms. This class also adds the ability of adding algorithms on run-time.

The class implements several properties as well as a couple of methods, namely:

- **public abstract string Name:** The property *Name* is meant to function as a way of distinguishing between multiple different algorithms.
- **protected abstract byte[] Signature:** This protected property is to be used by algorithms to verify that a given *CarrierMedia* has been embedded with a specific algorithm. Each algorithm should have a unique signature.
- **public virtual CryptoProvider CryptoProvider:** This is an optional property, meaning that the usage of a *CryptoProvider* should be up to each individual algorithm to define, and as such it is merely implemented within the class in order to ensure compatibility with future algorithms incorporating cryptography.
- **public virtual CarrierMedia CarrierMedia:** In order for the given algorithm to embed information within a *CarrierMedia* the algorithm needs a reference to the *CarrierMedia*.
- **protected virtual int Seed:** This property will get a seed from the *CryptProvider*. If there is no *CryptoProvider* the returned value is 0. This property is protected as it is only used by the algorithm.
- **public abstract long ComputeBandwidth():** When embedding information in the bits of the *CarrierMedia* a computation of the upper limit specifying the max amount of bytes able to be changed before 'running out' of storage space is calculated for the *CarrierMedia*. *ComputeBandwidth* is a method called with the purpose of calculating and returning the above mentioned storage space within the given *CarrierMedia*.
- **public abstract void Embed(StegoMessage, IProgress<int>, CancellationToken):** It is within this method that a *StegoMessage* is passed as a parameter and embedded into the given *CarrierMedia*. This method also requires an *IProgress<int>* that is used to report progress on how the embedding is going as well as a *CancellationToken* which is used in case the user cancels the embedding process. Implementation is required since it is an abstract method.
- **public abstract StegoMessage Extract():** This method enables one to extract a *StegoMessage* from the *CarrierMedia* property. Extract returns a *StegoMessage*.

7.4.2 LSB Algorithm

The class *LSBAlgorithm* is derived from the abstract class *StegoAlgorithmBase*. A pseudo-random sequence of integers (*RandomNumberList*) is created from the *Seed* with the purpose of determining the pattern in which the carrier media is changed.

The class *LSBAlgorithm* also defines the method *Embed* in which a for loop iterates through the given message to be embedded. As seen in Code

fragment 7.9, this method iterates through all the bits of the local variable *messageBits* which is an array of bits representing the message we want to embed.

```
1  for (int index = 0; index < messageBits.Length; index++)
2  {
3     int byteArrayIndex = numberList.Next;
4     byte sampleValue = CarrierMedia.ByteArray[byteArrayIndex];
5
6     // Get the least significant bit of current position.
7     bool carrierBit = (sampleValue & (byte) WorkingBit) == (byte)
          WorkingBit;
8
9     // Flip working bit if no match.
10    if (carrierBit != messageInBits[index])
11    {
12       CarrierMedia.ByteArray[byteArrayIndex] ^= (byte) WorkingBit;
13    }
14 }
```

Code fragment 7.9. The Embed method iterates through all bits in messageBits

The iteration ensures that each individual bit representing the message to be embedded is stored within the bit representation of the given carrier media. It also checks whether or not the bit representation of the carrier media already corresponds to the message bit in which case the given iteration is skipped. It should be noted that this algorithm supports more than just modification of the least significant bit. We have implemented a property, *WorkingBit*, which indicates which bit is to be modified and read from. This option can be edited by the user, in the 'Advanced Options' tab of the GUI, and adds the ability to embed and extract messages from various bits. This means that one could potentially store 8 messages in one carrier media, since there are 8 bits in one byte.

The algorithm also defines the method *Extract* in which the bits corresponding to the pattern set by the seed are read and the given signature corresponds to the one used for extraction.

```
1  RandomNumberList numberList = new RandomNumberList(Seed,
       CarrierMedia.ByteArray.Length);
2
3  // Read bytes and verify magic header.
4  if (!ReadBytes(numberList,
       Signature.Length).SequenceEqual(Signature))
5  {
6     throw new StegoAlgorithmException("Magic header is invalid,
          possibly using a wrong key.");
7  }
8
9  // Read data size.
10 int dataSize = BitConverter.ToInt32(ReadBytes(numberList, 4), 0);
11
```

```
12 // Return new instance from read data.
13 return new StegoMessage(ReadBytes(numberList, dataSize),
       CryptoProvider);
```

Code fragment 7.10. The signature is verified and data is copied to instantiate a StegoMessage.

If the signature/magic header does not match, a *StegoAlgorithmException* is thrown.

Within the class *LSBAlgorithm* we also have the method *ReadBytes*, which is seen in Code fragment 7.11. This method takes in a *RandomNumberList* representing the positions in the byte array to be read as well as an integer representing the amount of bytes to read.

```
1  // Allocate BitArray with count * 8 bits
2  BitArray tempBitArray = new BitArray(_count * 8);
3
4  // Iterate through the allocated amount of bits
5  for (int i = 0; i < tempBitArray.Length; i++)
6  {
7     tempBitArray[i] = (CarrierMedia.ByteArray[_numberList.Next] &
          (byte) WorkingBit) == (byte) WorkingBit;
8  }
9
10 // Copy bitArray to new byteArray
11 byte[] tempByteArray = new byte[_count];
12 tempBitArray.CopyTo(tempByteArray, 0);
```

Code fragment 7.11. Bytes are read in a random order and the value of the working bits are extracted.

The method returns a byte array which is copied from the *BitArray*.

7.4.3 Graph Theoretic Algorithm

The graph-theoretic algorithm (GTA), like the other algorithms, is derived from the abstract class *StegoAlgorithmBase*. There are several properties in the class made available for user adjustment through the 'Advanced Options' tab of the GUI (see section 7.7).

SamplesPerVertex: The number of samples collected in each vertex. Higher numbers means less bandwidth but more imperceptibility. (Default = 2, Range = 1 - 4)

MessageBitsPerVertex: The number of bits hidden in each vertex. Higher numbers means more bandwidth but less imperceptibility (Default = 2, Range = 2^0 - 2^2)

DistanceMax: The maximum distance between single sample values for an edge to be valid. Higher numbers means less visual imperceptibility but more statistical imperceptibility. Higher numbers might also decrease performance depending on DistancePrecision. (Default = 8, Range = 2 - 128)

DistancePrecision: The distance precision. Higher numbers significantly increases performance with high DistanceMax. (Default = 4, Range = 2^0 - 2^5)

VerticesPerMatching: The maximum number of vertices to find edges for at a time. Higher numbers means more memory usage but better imperceptibility. (Default = 50,000, Min = 10,000)

Embed

Embedding with a graph theoretic approach to steganography requires several processes, each of which is handled in their own method as follows:

- *GetMessageChunks*: Splitting the message.
- *Sample.GetSampleListFrom*: Finding the samples of the carrier media.
- *GetVerticeLists*: Collecting these samples into vertices.
- *GetEdges*: Finding the edges between these vertices.
- *Swap*: Swapping samples between the vertices according to the edges.
- *Adjust*: Directly adjusting the samples of vertices that could not be swapped.

Before these processes are handled, some variables are calculated based on the user adjustable properties as seen in Code fragment 7.12.

```
1 modFactor = (byte)(1 << messageBitsPerVertex);
2 bitwiseModFactor = (byte)(modFactor - 1);
```

Code fragment 7.12. Calculation of modulo variables.

modFactor corresponds directly to the mod4 of section 6.2 where the *bitwiseModFactor* is calculated in order to enable the modulo calculations through the fast bitwise *&* operator rather than the slow arithmetic *%* operator. It should be noted however that this only works as long as *modFactor* is a power of 2, though this is not a problem in our case.

GetMessageChunks

The GetMessageChunks method joins the *GraphTheorySignature* and the encrypted message and converts the result to a *BitArray*. It then creates a list of message values each corresponding to a number of bits as indicated by the *MessageBitsPerVertex* property. The code for a single value generation can be seen in Code fragment 7.13.

```
1 // Find current chunk value.
2 byte messageValue = 0;
3 for (int byteIndex = 0; byteIndex < messageBitsPerVertex;
      byteIndex++)
4 {
5    messageValue += messageBitArray[indexOffset + byteIndex] ?
         (byte)(1 << byteIndex) : (byte)0;
6 }
7 // Add chunk to list.
8 messageChunklist.Add(messageValue);
```

Code fragment 7.13. The calculation of a message value.

It should be noted that the bit pattern of the *messageValue* is the reverse of the pattern of bits of the given section of *messageInBits*.

Sample.GetSampleListFrom

The creation of samples is handled by the static method *GetSampleListFrom* of the *Sample* class. Samples are created by splitting the *CarrierMedia.ByteArray* into parts with a length as indicated by *CarrierMedia.BytesPerSample*. As can be seen in Code fragment 7.14 each sample is created by copying number of bytes from the *CarrierMedia.ByteArray* to a local byte array, *sampleValues*. After instantiating a *Sample* with these byte values, the mod value is calculated.

```
1  byte[] sampleValues = new byte[bytesPerSample];
2
3  for (int i = 0; i < bytesPerSample; i++)
4  {
5     sampleValues[i] = _carrierMedia.ByteArray[currentSample++];
6  }
7
8  // Add new sample to list.
9  Sample sample = new Sample(sampleValues);
10 sample.UpdateModValue(_bitwiseModFactor);
11 sampleList.Add(sample);
```

Code fragment 7.14. The creation of a sample.

GetVerticeLists

The initial part of vertex creation is very similar to a sample. The main difference is that the samples added to the *vertexSamples* array are pseudo-randomly selected with by using a *RandomNumberList*. This is to ensure that the message is spread throughout the carrier media.

Once the requisite number of samples have been collected and the corresponding value calculated, this value is compared to the corresponding *messageValue*. If they are equal nothing is done since we are not interested in adjusting parts of the carrier media that represent the correct *messageValue*. If not equal, a vertex is instantiated, as seen in Code fragment 7.15, and a *TargetModValue* is calculated for each *Sample* contained in the vertex.

```
1  Vertex messageVertex = new Vertex(tmpSampleArray) { Value =
       vertexModValue };
2  messageVertices.Add(messageVertex);
3
4  // Calculate delta value.
5  byte deltaValue = (byte)((modFactor + _messageChunks[i] -
       messageVertex.Value) & bitwiseModFactor);
6
7  // Set target values.
8  foreach (Sample sample in messageVertex.Samples)
9  {
```

```
10      sample.TargetModValue = (byte)((sample.ModValue + deltaValue) &
            bitwiseModFactor);
11  }
```

Code fragment 7.15. Instantiating a *Vertex* and calculation of the corresponding samples *TargetValue*.

When all message values are accounted for, any additional vertices are added to the *reserveVertices* list with both vertex *Value* and sample *TargetModValue*'s set to *modFactor*.

FindEdgesAndSwap

With a high number of vertices the average number of edges found rises squared [48]. With large messages, and so large numbers of vertices, this memory usage can rise prohibitively high. In order to accommodate large messages, it has thus been necessary to implement a limit on the number of vertices the algorithm will attempt to find edges for at once. This limit is set by the *VerticesPerMatching* property allowing the user to set it according to the memory capacity of the local machine.

The implementation of this limit is done through the method *FindEdgesAndSwap* which iterates through all the vertices, each iteration handling the edge-finding and swapping for a subset of vertices. Instead of each iteration handling the amount indicated by *VerticesPerMatching* which could lead to the last iteration handling a very small remainder of vertices, each of these iterations handle the average of the total number of vertices divided by the number of runs needed, rounded up. The iteration is done through a loop as can be seen in Code fragment 7.16.

```
1  while (verticeOffset < startNumVertices)
2  {
3      (...)
4
5      // Calculate how many vertices to use this round.
6      int verticesThisRound = verticesPerRound > _vertices.Count ?
           _vertices.Count : verticesPerRound;
7      verticeOffset += verticesThisRound;
8
9      // Take this amount of vertices.
10     List<Vertex> tmpVertexList = _vertices.GetRange(0,
           verticesThisRound);
11
12     (...)
13
14     // Add leftover vertices to tmpVertexList.
15     tmpVertexList.AddRange(leftoverVertexList.GetRange(0,
           leftoverCarryover));
16
17     // Remove the transfered vertices from the list.
18     leftoverVertexList.RemoveRange(0, leftoverCarryover);
19
```

```
20      // Get edges for subset.
21      GetEdges(tmpVertexList, _progress, _ct, roundProgressWeight);
22
23      // Swap edges found for subset and add leftovers to list.
24      leftoverVertexList.AddRange(Swap(tmpVertexList));
25
26      // Clear edges for subset.
27      tmpVertexList.ForEach(v => v.Edges.Clear());
28  }
```

Code fragment 7.16. *while*-loop iterating through all vertices calling *GetEdges* and *Swap* in each iteration.

In order to minimize the number of vertices needing direct adjustment, the leftover vertices, which could not be swapped, are carried over from each iteration to the next. In order to avoid excessive bloating each iteration checks that the amount of leftover vertices from the previous iteration, in the *leftoverVertexList* list, is no more than a quarter of *VerticesPerMatching*. If the amount of leftovers is above this limit the maximum carryover allowed is transfered and removed from the list with the excess leftovers carrying over to the next iteration again. Once the loop is finished the *leftoverVertexList* is returned.

GetEdges

GetEdges is the main part of the algorithm. In accordance with the approach described in section 6.2, for an edge to exist between 2 vertices the *ModValue* and *TargetValue* of one vertex's sample must respectively be equal to the *TargetValue* and *ModValue* of a sample of the other vertex. Finding these relations in an efficient manner is the main challenge of a graph theoretic algorithm.

Initially we attempted to implement an algorithm based on an outline provided by a group of math students. This algorithms main principle was to iterate through all vertices and check them against all the remaining vertices.

```
1  V = {vertex[1], vertex[2] ... vertex[N]}
2  for vertex[i]: i = 1 -> N
3     for vertex[j]: j = i+1 -> N
4        Check for edge between vertex[i] and vertex[j].
```

Code fragment 7.17. Pseudocode of an approach to edge finding.

This algorithm would work well with relatively small messages, but the number of checks needed for this algorithm would quickly become extreme for high amounts of vertices being $numChecks = (n^2 + n)/2$.

Though the total set of vertices for larger messages could be split into smaller subsections and handled independently, the small number of vertices that could be handled efficiently at once would significantly reduce the number of edges found. This would reduce imperceptibility against both visual and statistical

detection and as such we decided to try and find another approach to edge finding.

The approach we came up with was to not check pairs of vertices for the existence of an edge between them, and subsequently the quality of the edge if it exists, but instead check only whether any vertices existed with the values that we knew would give an edge. Essentially the edge finding algorithm would be reduced to approximate the process seen in Code fragment 7.18.

```
1 V = {vertex[1], vertex[2] ... vertex[N]}
2 for vertex[i]: i = 1 -> N
3    Check for edge-valid vertices in the neighborhood.
```

Code fragment 7.18. Pseudocode of another approach to edge finding.

As such the number of checks would be reduced significantly since it would be linearly proportional to the number of vertices instead of polynomially proportional as the previous algorithm. The challenge with this algorithm was to find a method of efficiently checking the neighborhood for valid vertices. Our approach was to first populate a collection with every vertex, identified by their defining values for edge-finding purposes, namely the *Sample.Values*, *Sample.ModValue* and *Sample.TargetValue* values, for each sample of the vertex. In this the neighborhood search, each vertex would simply check for each value set that we knew would give a valid edge whether any vertices were located here. If so, edges would simply be made between the searching vertex and all vertices at the current location. Two different types of collections were tried for this, each with their own pros and cons.

	Pros	Cons
Array	Fast	Memory Costly
Dictionary	Low memory cost	Slower

Table 7.1. The pros and cons of different collection types for edge finding.

The memory cost of an *Array* stems from the need of allocating space for an item at every possibly index. Assuming a *Sample* with 3 *byte* values, as is the case for pictures, and 2 bits hidden per vertex resulting in 2^2 possible unique values for *ModValue* and *TargetValue* this would need a 5-dimensional array with a total of $256 * 256 * 256 * 4 * 4 = 268,435,456$ elements. If each element is a reference to a *List* identifying all vertices located there, this would mean a minimum size of $268,435,456 * 8bytes = 2,147,483,648bytes = 2GiB$ for just the array not including the size of the lists themselves.

This is inefficient since many locations in the array may not have any vertices located there, and the references thus being *null*. A *Dictionary*, on the other hand, only has the items that are actually needed thus saving a lot of space in most cases. The downside to a *Dictionary* is that the elements cannot simply be directly accessed as one can with an *Array* but needs to check for the existence of an element with a key. Although the *TryGetValue* method approaches O(1) complexity, it isn't quite O(1) [52], and necessitates the calculation of the keys *HashCode*.

An optimization was required regardless of the chosen type and the solution we came up with was to reduce the precision of the collection. In the initial trials each index or key would exactly identify the values of the sample located there, but with the precision reduced so a index/key identified a collection of samples with slight variances in the actual values we could ameliorate both problems.

We could reduce the number of necessary elements thus reducing the size of an *Array* and we would also not need to search the collection as many times. The method to do this was simply to divide the *Sample.Values* values with a common factor. Thus depending on the division factor, (D), and the number of values, V, in *Sample.Values* each index/key would correspond to D^V different value sets. This division would not be applicable to the *Sample.ModFactor* or *Sample.TargetValue* since these uniquely determine whether an edge is possible, with the *Sample.Values* only determining the weight of an edge.

In the end we chose to go with a collection of *Arrays* due to the speed it supplied and with a division factor of only 2, the size of the array, assuming an *ImageCarrier*, would be reduced to only 256 MiB which we deemed acceptable. In order to avoid expensive *division* operations we have limited the possible division factors to different powers of 2 thus allowing the usage of the fast bitwise >> operation instead.

In order to avoid every edge being found for each vertex it applies to we only search approximately half the nodes in range. In this way, even if we only search the space 'above' the current vertex we still get any edges with vertices 'below' the current vertex since they will be found for those vertices.

We therefore need to check approximately $(N + 1) * (2N + 1) * (2N + 1) = 4N^3 + 8N^2 + 5N + 1$ nearby nodes for each vertex with $N = MaxDistance/2^{DivisonPrecision} = MaxDistance >> DivisonPrecision$.

Swap

The Swap method first sorts the vertices according to the number of edges they have and then iterates through them starting from the vertex with least edges. For each vertex its edges are sorted according to *weight*, and the first edge with both vertices not already having been swapped is applied, swapping the values stored in the respective *Sample.Values* field.

Adjust

For the vertices that could not be swapped, a single value in a single *Sample*'s *Sample.Values* is directly modified by the difference between that samples' *ModValue* and *TargetValue*. Which of the vertex's samples and which of the samples' *Values* is adjusted starts as the last ones decrementing with each vertex until reaching the first one before resetting, ensuring an even spread.

7.4.4 Common Sample Algorithm

In order to illustrate the diversity of steganography algorithms we have implemented an algorithm with the name Common Sample Algorithm (CSA).

This class is also derived from the abstract class *StegoAlgorithmBase*.

This algorithm is a middle ground between the previous LSB and graph-theoretic algorithm. In terms of time complexity it is theoretically slower than LSB, but still significantly faster than the graph-theoretic algorithm. In terms of imperceptibility the amount of unique samples is is theoretically decreased with this algorithm, whereas LSB will increase it and the graph-theoretic algorithm will attempt to keep it the same.

Since the amount of unique samples is typically decreased this can cause differences between input and output carrier media that are quite noticeable to a human, making it vulnerable to known-cover attacks. However, a smaller amount of unique samples could theoretically make the carrier less prone to detection from known steganalysis methods. The algorithm follows the following steps:

1. A list of *Samples* is generated from the *CarrierMedia* using the property *BytesPerSample* to determine how many bytes should be read for every *Sample*.
2. All *Samples* are grouped so that each unique sample combination is in its own group. The *Samples* are then ordered by their frequency meaning the most frequent *Sample* is the first in the list.
3. For every bit that is embedded, a random *Sample* (**R**) is picked from the unsorted list of *Samples*.
4. If **R** has a target value that matches the bit it will be skipped. Otherwise a matching *Sample* (**M**) is picked from the most frequent *Sample* list. **M** should not differ too much from **R** such that the matching *Sample* is the one with the lowest distance.
5. If a suitable match **M** is found, the byte values of **R** are replaced with the ones of **M**. Otherwise if no match is found the byte values of **R** is modified so that it has a matching value.
6. When all *Samples* have been replaced or changed, they are encoded back into the *CarrierMedia* by iterating through all of them and writing their byte values.

We have introduced two properties that can be modified by users who wish to use advanced options:

MaxDistance: The maximum distance between two samples. A higher value means that the changes are more noticeable, but a value that is too low will reduce the amount of suitable matches. (Default = 250)

MaxSampleCount: The maximum amount of frequent samples to search for replacements. Higher values increase time complexity, but too low values reduce the amount of suitable matches. (Default = 1000)

7.5 Cryptography

This section covers the implementation of the cryptographic algorithms we have chosen to work with. These algorithms have been added to increase the imperceptibility and security of our steganographic messages. As cryptography

gives a pseudo random output according to the key and our steganographic algorithm works around bit patterns, we have the possibility of creating a pattern that are highly randomized and obscure to individuals that might want to detect these messages. Furthermore, these messages are very hard to decrypt, should an individual try to gain access to the information, without the encryption key.

7.5.1 ICryptoProvider

The classes containing cryptographic algorithms all implements the interface *ICryptoProvider*. By using interfaces we can ensure the ability to choose from different cryptography algorithms. An interface acts as a blueprint for the implemented classes which must contain the same properties and implement the same methods contained in the interface. Utilizing interfaces makes different classes, that implements the same interface, behave the same way, which is practical when using classes that are doing the same thing in different ways. This is the case for our program's carrier media and cryptography algorithms. In the case of *ICryptoProvider*, the classes that implements this interface must implement the following methods and properties:

- **byte[] Key:** Read/write property indicating the key being used in encryption or decryption.
- **int HeaderSize:** Read-only property indicating how many bytes the algorithm uses in its header. This is used when computing available bandwidth.
- **string Name:** Read-only property indicating the name of the algorithm.
- **int KeySize:** Read-only property indicating the key size in bits. At the moment the interface does not support dynamic key sizes.
- **int Seed:** Read-only property representing a seed that is used by steganographic algorithms to determine pseudo-random patterns.
- **void SetKey(string):** Creates a key from a string.
- **byte[] GenerateKey():** Generates a random key, but does not support algorithms using key pairs.-
- **byte[] Encrypt(byte[]):** Returns encrypted data from an existing byte array.
- **byte[] Decrypt(byte[]):** Returns decrypted data from an existing byte array.

By having different methods for encryption and decryption it is possible to implement both symmetric- and asymmetric key encryption, which is what we have chosen to do.

7.5.2 AESProvider

The class *AESProvider* implements the *ICryptoAlgorithm* interface with an implementation of the symmetric algorithm AES. This implementation is a wrapper around the *AesManaged* class which is the .NET Framework implementation of the algorithm.

Since AES requires an initialization vector (IV) when using cipher block chaining (CBC) that is generated in the *Encrypt* method. AES also requires a

key, and as such, *Encrypt* should not be called until a key has been set through either the *Key* property or the *SetKey* method which converts plaintext to a valid key:

```
1 public void SetKey(string _keyString)
2 {
3     Key = string.IsNullOrEmpty(_keyString) ? null :
          KeyDeriver.DeriveKey(_keyString, KeySize);
4 }
```

Code fragment 7.19. Create key from plaintext

The method in Code fragment 7.19 uses the utility class *KeyDeriver*, given that the provided *keyString* is not null. This utility class has the simple purpose of converting a plain text key to one that corresponds to the provided *keySize*. This has to be done since AES does not support all key sizes. When a *Key* has been set the *Encrypt* method can be called. Before the actual encryption takes place the IV is prepended to the encrypted data. This is done because the decryption part has to know the IV.

In the *Decrypt* method, the first data read from the input stream is the initialization vector that is always 16 bytes long.

7.5.3 RSAProvider

RSAProvider is another implementation of *ICryptoProvider* and enables the ability to use asymmetrical-key encryption. It uses the *RSACryptoService-Provider* class which is a part of the .NET Framework. Since RSA is not able to encrypt big chunks of data at once, a hybrid encryption scheme has been implemented. This works in the following way:

- Generate a random encryption key **K**.
- Symmetrically encrypt the entire block of data **P** using **K** which gives us the encrypted data **C**.
- Asymmetrically encrypt **K** and prepend it to **E**.

In our implementation we have used AES as our symmetric-key algorithm. When the data is decrypted the above process is reversed:

- Read the encrypted key **K** and decrypt it asymmetrically.
- Use the decrypted key **K** and decrypt the encrypted data **C**. This gives us the original block of data **P**.

7.6 Carriers

In the following section everything to do with carrier media will be explained. This includes the interface (*ICarrierMedia*) which describes the least amount of methods, properties and so on needed for a carrier media to be considered a carrier by our program. It also includes actual implementers of the interface such as *AudioCarrier* and *ImageCarrier* which carry audio and image files respectively. Having an interface also enables us to account for future updates

to file formats, seeing as all the necessary methods and properties is already defined within the interface *ICarrierMedia*.

7.6.1 ICarrierMedia

Since we want our program to support as many carrier media as possible we have created the interface *ICarrierMedia*. This enables steganographic algorithms to interact with any type of carrier media.

The interface implements the following properties and methods:

- **byte[] ByteArray:** Read/write property containing a byte array with the inner data of the media. For example, for an image this would be an array of color samples.
- **string OutputExtension:** Read-only property indicating the default output extension for the carrier.
- **Image Thumbnail:** Read-only property which provides a thumbnail associated with the carrier. This is used by the GUI to give a visual representation of the carrier.
- **int BytesPerSample:** Read-only property indicating how many bytes there are for each sample. For example, we refer to pixels as samples of **n** bytes where **n** is the amount of channels.
- **bool IsExtensionCompatible(string):** This method will check if a specified file extension is compatible with this algorithm. This is different from *OutputExtension* since a carrier can support multiple file extensions.
- **void Decode():** The purpose of this method is to insert the inner data of the media into the *ByteArray* property.
- **void Encode():** The purpose of this method is to insert the content of the *ByteArray* property back into the original media.
- **void LoadFromFile(string):** The purpose of this method is to load a file into the carrier media.
- **void SaveToFile(string):** The purpose of this method is the ability to directly save a carrier media without having to worry about its type.

7.6.2 AudioCarrier

The *AudioCarrier* class handles an audio file. It implements *ICarrierMedia* interface. The only variable in *AudioCarrier* that does not exist in *ICarrierMedia* is *audioFile* which is of the type *AudioFile*. The *AudioFile* class is a type that can hold an audio format. It can be inherited by classes that can handle specific audio extensions making it is easy to implement code to support other audio formats.

When *LoadFromFile* is called on an instance of *AudioCarrier* the *audioFile* variable is set to the relevant *AudioFile* subclass. The only currently supported format is the WAV-format. After validation of the input file the *Decode* method is called.

The *Decode* method assigns the inner data of the audio file to *ByteArray*. The inner data contains the data from the original file without the header data. This data can be manipulated without corrupting the file since it cannot change the

meta data. The data is extracted by calling the method *CopyInnerData* in *AudioFile* and assigning it to *ByteArray*.

The *Encode* method encodes the inner data of the audio file. This means that it transfers the data from *ByteArray* back to the same position that *Decode* removed it from, combining the header and inner data again. This changes the inner data of the *audioFile* to the manipulated version of the *ByteArray* by executing the *SetInnerData* method in *AudioFile*.

The audio file will always be encoded when the *SaveToFile* method is called and thereby ensure that the file is encoded before saving. It is also the only place that calls the *Encode* method. *SaveToFile* takes a path as a parameter which is where the encoded audio file will be saved. It utilises the .NET Framework class *File* to access the *WriteAllBytes* method that takes a path defined as *destination* and a byte array as parameters and writes the audio file to the specific destination.

7.6.3 ImageCarrier

Like *AudioCarrier*, the *ImageCarrier* class implements the *ICarrierMedia* interface. It has a property *ImageData* of the type *Image* which has the same purpose as the variable *audioFile* in the *AudioCarrier* class.

When the *LoadFromFile* method is called on an instance of *ImageCarrier*, it will load the image from a file using the method *LoadImageFromFile*. The *ImageData* property is set to the returned value from this method.

When the *ImageData* property is set to a new value the image is automatically converted to an image of PNG format with a 24-bit depth as seen in Code fragment 7.20.

```
1  get { return imageData; }
2  set
3  {
4     // Use original image if it meets standards, otherwise convert
          it to 24 bpp
5     if (Equals(value.RawFormat, ImageFormat.Png) &&
          value.PixelFormat == PixelFormat.Format24bppRgb)
6     {
7        imageData = (Bitmap) value;
8     }
9     else
10    {
11       // Copy image
12       Bitmap newImage = new Bitmap(value.Width, value.Height,
             PixelFormat.Format24bppRgb);
13       using (Graphics graphics = Graphics.FromImage(newImage))
14       {
15          graphics.DrawImage(value, new Rectangle(0, 0, value.Width,
                value.Height));
16       }
17
18       imageData = newImage;
```

```
19      }
20  }
```

Code fragment 7.20. Images are automatically converted to 24-bit depth PNGS.

The *Decode* method gets the data of the image, without the header data, and saves it in *ByteArray*. In order to manipulate the sample values of the image the method, *LockBits*, is used. This method allows direct access to the sample values, but since these are in a reverse order, BGR instead of RGB, and may have some padding, the data has to be copied in a loop as seen in Code fragment 7.21.

```
1  // Iterate through pixels
2  int dstPosition = 0;
3  for (int i = 0; i < backingImageData.Width; i++)
4  {
5     for (int j = 0; j < backingImageData.Height; j++)
6     {
7        // We have to 'reverse' each pixel as they are in BGR format
             (we want RGB)
8        int basePosition = j * imageData.Stride + i * BytesPerSample;
9        ByteArray[dstPosition++] = scanPtr[basePosition + 2];
10       ByteArray[dstPosition++] = scanPtr[basePosition + 1];
11       ByteArray[dstPosition++] = scanPtr[basePosition + 0];
12    }
13 }
```

Code fragment 7.21. Sample values are copied to ByteArray in the correct order.

The *Encode* method works a lot like the *Decode* method. It locks the bits of the image, but copies the contents of *ByteArray* back into the image. This also has to be done in a reverse order since we have RGB but want BGR. This process is seen in Code fragment 7.22.

```
1  // Copy the pixel array from ByteArray to the innerImage
2  int srcPosition = 0;
3  for (int x = 0; x < backingImageData.Width; x++)
4  {
5     for (int y = 0; y < backingImageData.Height; y++)
6     {
7        // We now have to turn RGB into BGR
8        int basePosition = y * imageData.Stride + x * BytesPerSample;
9        scanPtr[basePosition + 2] = ByteArray[srcPosition++];
10       scanPtr[basePosition + 1] = ByteArray[srcPosition++];
11       scanPtr[basePosition + 0] = ByteArray[srcPosition++];
12    }
13 }
```

Code fragment 7.22. Sample values from ByteArray are copied back into the image in the correct order.

The *SaveToFile* method calls *Encode* to ensure that the saved image contains the manipulated data. The image will always save the image with a PNG

extension because it has lossless compression (see section 6.7) lowering the risk of losing data in the exchange process.

7.7 Graphical user interface

In this section we will explain the setup of the program's GUI and why we made it as it is.

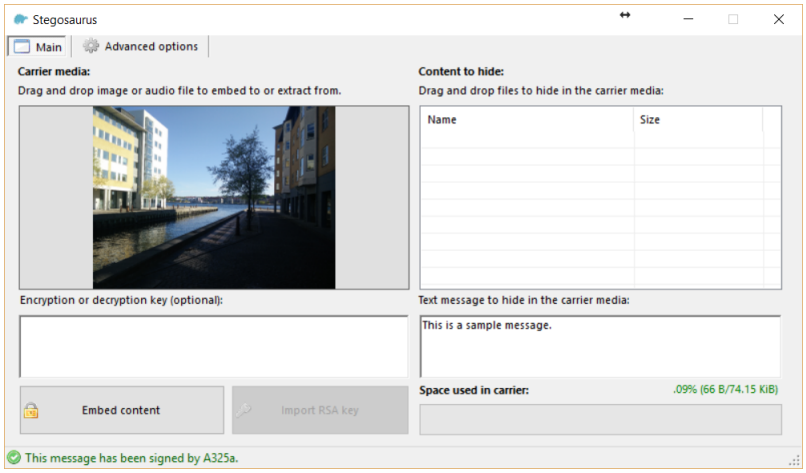


Figure 7.3. Main tab of GUI.

The GUI is split into two tabs. The default tab is the 'Main' tab, shown in Figure 7.3, and the other is an 'Advanced Options' tab as seen in Figure 7.4.

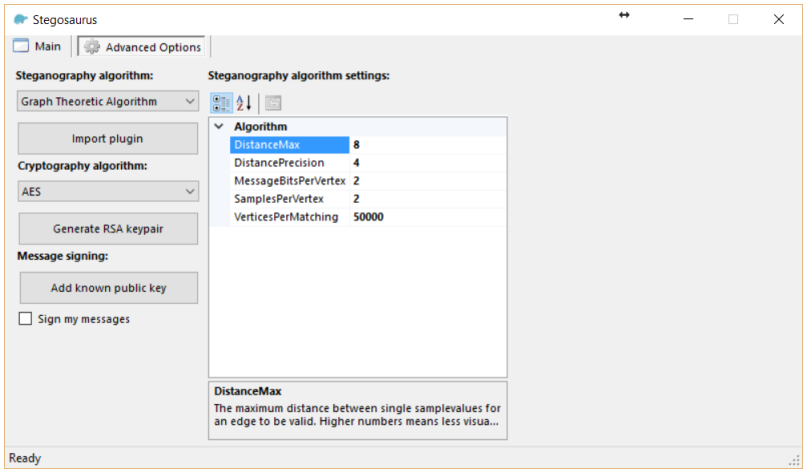


Figure 7.4. Advanced Options tab of GUI.

In the 'Main' tab the main part of the GUI is located. Here we have an area to drag and drop the carrier file that the user wants to hide content within. When a carrier medias is dropped into this area it will set the content of the *PictureBox* to the *Thumbnail* property of the carrier. Additionally the program gets the file path and creates an instance of *CarrierType* with the file path and calls the method *HandleInput* with the *CarrierType* as parameter. We have chosen to implement drag and drop and not have a visible browse option as

we find drag and drop very intuitive and easy to use, which is important for the GUI we want. We also have the idea that having many buttons will be daunting for a first time user which is another reason that we do not have the browse button visible. However, browsing for files is still possible by right clicking the drag and drop sections. This will also be displayed as a tool-tip when hoovering the mouse over the drag and drop sections.

Next to the carrier media drag and drop area there is a 'content to hide' drag and drop area. When one or more files are dragged and dropped into this area it will display the selected files in a list. This area gets the file path the same way as the carrier media area and creates an instance of *ContentType* with the file path, it also calls the function *HandleInput* with the *ContentType* as parameter.

HandleInput takes *IInputType* as formal parameter which both *ContentType* and *CarrierType* inherits from. Handle input creates an instance of *InputFile* and *FileInfo* which is used to display the file info to the user. It then checks if the actual parameter is either *ContentType* or *CarrierType* and handles it accordingly. If the input is of type *ContentType*, an instance of *ListViewItem* is instantiated and retrieves info to display the files in the list of 'content to hide' to the user, such as the type of file, size and icon. Then the *InputFile* is added to the list of *InputFile*'s. If the parameter is of the type *CarrierType*, an iteration of all possible carrier media will attempt to find a carrier class that supports the given file extension as seen in Code fragment 7.23. If there is no matching carrier class an error is shown to the user.

```
1  // Find suitable carrier media
2  foreach (Type carrierType in carrierMediaTypes)
3  {
4     // Skip types that are not ICarrierMedia.
5     if (!carrierType.GetInterfaces().Contains(typeof(ICarrierMedia)))
6     {
7        continue;
8     }
9
10    // Create instance.
11    carrierMedia = (ICarrierMedia)
          Activator.CreateInstance(carrierType);
12
13    // Validate extension
14    if (!carrierMedia.IsExtensionCompatible(fileInfo.Extension))
15    {
16       carrierMedia = null;
17    }
18    else
19    {
20       carrierMedia.LoadFromFile(fileInfo.FullName);
21       pictureBoxCarrier.Image = carrierMedia.Thumbnail;
22       carrierExtension = carrierMedia.OutputExtension;
23       break;
24    }
```

```
25 }
```

Code fragment 7.23. A carrier class which supports the given file extension is found.

Beneath the carrier media drag and drop area there is a text box for an encryption- or decryption key. This key should be written by the user. It is not necessarily required, but it is strongly recommended to add a layer of security. However, if the encryption algorithm used is RSA encryption, which can be changed in the 'Advanced options' tab, the key can be imported using the button below called 'Import RSA key'. This will have the user select a file in XML format containing a RSA key. If the user does not have a RSA keypair in XML the keypair can also be generated by a button in the 'Advanced Options' tab called 'Generate RSA keypair'. This will ask the user where to save the two RSA keys. The key entered or imported will be passed to the *ICryptoProvider* instance's method *cryptoProvider.SetKey*.

In the GUI there is another text box beneath the content to hide drag and drop area. In this area users can manually write a text message to embed along with the files or without files if this is needed. The property *TextMessage* in the instance of *StegoMessage* will be set to this value.

In the bottom right corner there is a 'Space used in carrier' bar. This bar shows the user how much space is used in the carrier media to save content and how much space is left. Just above the actual bar the space used will be shown as a percentage and the used space and maximum space will also be shown in KiB. These indicators will be updated every time a file is inserted or removed from content to hide, the carrier media is changed or the message string is changed. A downside to this is the fact that, with larger amounts of data, there may be a slight delay whenever changes are made to the content or carrier media as the data will have to be re-compressed.

The last button in the main section of the GUI, is the 'Embed'/'Extract' button in the bottom left corner. This will execute the actual action of embedding or extracting. If only one carrier media is present in the program the button will show 'Extract' and execute the extraction procedures if clicked. If any content to hide is present the button's text will change to 'Embed' and the procedures to embed is executed if clicked. The button is disabled if no carrier media is present.

We have chosen to separate the advanced options from the main part because there are certain security risks for the user when these settings are changed as mentioned before.

In the 'Advanced Options' tab there are a few options to change. in the top left corner there is a drop down menu which allows the user to change the algorithm used to hide the data. Below there is another drop down menu to change the cryptographic algorithm used to encrypt the hidden data. Below the two drop down menus the button 'Generate RSA keypair' is located which was explained earlier. A button also allows the user to import plugins that will search dynamic libraries for the types *StegoAlgorithmBase*, *ICarrierMedia* and *ICryptoProvider*.

Next to the two drop down menus and the 'Generate RSA keypair' button, an area to change the settings for the steganographic algorithm is located. This allows the user to modify some of the public properties of different steganographic algorithms.

7.8 Exceptions

Dealing with exceptions in the program has lead us towards making our own program specific exceptions, as catching *System.Exception* would make exception handling to unexpected behavior from the program difficult. By avoiding using *System.Exception* to handle unexpected behavior in the program, and using custom exceptions instead, we allow ourselves to segregate different exception types for *catch* exception. Doing this poses the risk of creating redundant exceptions that basically could be handled with *System.Exception*. However we deem it necessary to create such a library to ease future functionality implementation and maintenance of the program. The custom exceptions also makes it possible to give the user meaningful and describing error messages if an expected error occurred.

The following is a short description of the custom exceptions:

StegosaurusException: This exception is used as a base for all of the custom exceptions.

InvalidCarrierFileException: This exception is used as a base class for exceptions related to invalid carrier media.

InvalidImageFileException: Thrown when the program fails to interpret an *Image* from a byte array.

InvalidWaveFileException: Thrown when the class *WaveFile* fails to parse different parts of the WAVE-file.

RandomNumbersOutOfRangeException: Thrown when a random integer is requested from *RandomNumberList*, but there are no integers left to generate. This should not happen unless an algorithm has an invalid implementation of *ComputeBandwidth*.

StegoAlgorithmException: Thrown when algorithms encounters an error. For example, this exception is thrown if the magic header does not match.

StegoCryptoException: Thrown when an implementer of *ICryptoProvider* encounters an error. For example, this exception is thrown by some algorithms if a wrong key is used for decryption.

StegoMessageException: Thrown when *StegoMessage* encounters an error. For example, this exception is thrown if the hash value of the encoded data is invalid when constructing a *StegoMessage* from a byte array.

StegoCarrierException: Thrown when a carrier media is used incorrectly.

Test 8

The following sections will explain how we have used and implemented two different kinds of tests namely unit testing and mock testing. The informal nature of ad hoc testing means that we do not have any documentation from these tests, as the code for these tests generally only survived until any problems they uncovered were fixed. An issue with mock tests is that we can not test all combinations of the different algorithms, so our mock test only uses the default algorithms. The reason we use unit tests is because they allow us to isolate issues in case the mock test fails. Although our usage of testing means that we have an expectation of functionality, one has to keep in mind that tests do not guarantee that there does not exist errors in the program. For example, a logical error in a unit test that could give a wrong estimate leading to an incorrect testing result.

8.1 Unit tests

8.1.1 Test of cryptography algorithms

Since we have implemented a couple of different cryptography algorithms we want to make sure that they all give the correct output given the correct key. For symmetric algorithms, this means that encrypting data with a key and using the same key to decrypt the data should return the original data. We have implemented two tests for all of our cryptography algorithms and they follow a similar scheme:

- Encrypt random data with a randomly generated key, then decrypt with the same key and checks that the decrypted data equals the previously generated random data.
- Encrypt random data with a randomly generated key, then decrypt with a new randomly generated key. This test expects a *CryptographicException* and fails if it is not thrown.

Test of AES and RSA

The *Decrypt_ CorrectKey_ CorrectOutput()* test checks if a decrypted byte array is the same as the original byte array when it is decrypted after it has been encrypted with the program.

The test fails in case of the decrypted output being different from the original input.

This test works by making a new byte array with some random bytes in it as well as an encryption key, or private and public keys in the case of RSA. This byte array is then encrypted and decrypted, using the appropriate keys, and an assertion checks whether the decrypted byte array contains the same bytes as the original byte array. If that is the case, the decryption most likely works as planned.

The *Decrypt_ WrongKey_ ThrowsCryptographicException()* test is essentially the same as the above, except this is to confirm the error handling of the method. This method has an *ExpectedException* attribute which indicates that the test is expecting an exception. To confirm the error handling we change the key, the private key for RSA, after the encryption and then a wrong key is used to decrypt it. This will of course not work as the key will differ from the encrypted data and the exception will be thrown.

8.1.2 Test of StegoMessage

The *Decode_ EncodedTextMessage_ CorrectOutput* test checks if the given string is the same string when we encode and decode it with the program.

The test fails if decoded cipher text as output is not the same as the plain text which is the variable *testString*.

The *Decode_ EncodedInputFile_ CorrectOutput* test follows the same approach as the test above except the fact that this time we test the implementation of file handling rather than handling the *TextMessage*.

The test checks the file names as well as the contents.

8.1.3 Test of RandomNumberList

The *GenerateNumbers_ HasNoDuplicates* test checks if the number list can contain duplicates. Duplicate numbers are not wanted in the number list and therefore this test is relevant by confirming that duplicates will not happen.

The test fails if the random number list contains any duplicates.

The test works by making a new *RandomNumberList* with 0 to *count* entries and an empty *List*. Then we iterate through the *randomNumbers* list and add each number to the *existingNumbers* list. If the number in *randomNumbers* is already in *existingNumbers* we will add one to *duplicateCount*. Finally, we check that the *duplicateCount* is 0 with an assert.

8.1.4 Mock tests

Our mock tests are tests that allow us to test the flow of our program from start to finish and make sure that all the parts can work together as we had first envisioned.

If our mock tests succeeds we can be somewhat certain that the program is working as intended. However, as with any other test we can not rely on the tests as a definitive proof of the program working.

The mock test is defined in the *TestSpecifiedAlgorithms* method and is used by the test methods *MainImplementation_ DefaultAlgorithms_ ExpectedOutput* and *MainImplementation_ CommonSampleAlgorithm_ ExpectedOutput*. The mock test has some default input that we use for all the mock tests. We also generate a bitmap image as a test carrier media in which to store the data. As the *TestSpecifiedAlgorithms* method takes the interface for *ICryptoProvider* and the base class *StegoAlgorithmBase* we can run the same test on different algorithms and cryptographic methods that our program supports.

Finally, we test whether or not the message encoded is the same as the message we get out of the carrier media. We test the file that we get by testing if the content of the file is the same as well as the file name.

Part III

Evaluation and further work

9 Evaluation

Throughout this chapter we will be evaluating on the requirements set for the program solution. This is done to clarify the reasoning behind choices that we made in the design and implementation chapters as well as to evaluate the quality of the product against our problem definition and solution criteria.

More specifically this evaluation will cover our primary criteria:

- Robustness.
- Time complexity.
- Imperceptibility.
- Bandwidth.

In addition, we will be evaluating on secondary criteria:

- Versatility of carrier media selection.
- Popularity of the compatible carrier media.
- User interface ease of use.
- User security.

9.1 Algorithm comparison

We have debated three different algorithmic solutions to the steganographic problem presented by our problem definition. As LSB and CSA are implemented to give the program versatility the main algorithm of our program is the GTA. However, to attain an understanding of algorithmic differences we will, in the following section, compare them according to our criteria and debate them accordingly.

Test with sample carrier media

We have done a series of tests with each algorithm embedding the same 53.72 KiB message into 4 different pictures, all $900 * 675$ pixels, with varying amounts of unique samples. The result of these tests can be seen in section 9.1.

Picture Info		Common Sample			Least Significant Bit			Graph-Theoretic		
Name	Unique Colors	Time	Unique	Forced	Time	Unique	Forced	Time	Unique	Forced
Picture 1	42,451	30.4 sec	-3,185	7.00%	0.13sec	+6,205	100%	271.9 sec	+6	0.03%
Picture 2	189,715	30.87 sec	-2,212	44.48%	0.14sec	+4,196	100%	133.5 sec	+9	0.14%
Picture 3	166,225	32.1 sec	-7,049	32.11%	0.13sec	+5,582	100%	241.8 sec	+3	0.03%
Picture 4	126,856	31.7 sec	-5,411	37.15 %	0.15sec	+1,799	100%	161.1 sec	+30	0.07%

Table 9.1. Tests on 4 different pictures with a 53.72 KiB message. 'Time' is the time from start to completion of the embedding process. 'Unique' is the difference in number of unique samples in the carrier before and after the embedding process. 'Forced' is the percentage of units (bytes, samples or vertices) that needed direct adjustment.

From these tests it can be seen that although the LSB algorithm is indeed very fast it also, as expected, significantly increases the amount of unique colors present in the carrier. While the Common Sample algorithm was somewhat slower it still completed in a reasonable time. Although the number of unique colors in the carrier was lowered after embedding the magnitude of difference in unique colors still provides possibility for statistical detection. Visual inspection also revealed more obvious differences compared to the LSB algorithm though still within what we deem acceptable. The graph-theoretic algorithm performed the slowest, as expected, but we still deem the performance well within the limits of the acceptable considering the relatively large message size. Also as expected, the difference in unique colors has been kept extremely low, well under 0,01% in all cases, though some of this must be attributed to the forced adjustments hitting existing values.

Of note is that the 'Forced' columns of section 9.1 indicate the percentage of units which needed forced adjustment of all those that needed correction, meaning, the units whose value already corresponded with the correct message value are not part of this calculation. If these units were part of the calculation the numbers for LSB and CSA would on average be halved, while the GTA numbers would be approximately three quarters of the given numbers.

Robustness

Robustness is a criteria which we have tried to implement into the program. However, it has proven a difficult challenge to implement as some social media, such as Facebook, have their own compression algorithm of which we have no knowledge or means to get information about. We have decided to accept the incompatibility with these social media as a necessary flaw due to the unavailability of the algorithm information. Instead we have found alternatives to such social media sites that can be used since they do not use a compression algorithm. These alternatives include Imgur and Skype which are both popular.

In the initial phase of the project we found that Twitter was a viable alternative as most images were not compressed. At the final stages of the project, we found that 24-bit PNG images are compressed on Twitter but not 32-bit ones, which are not supported by our algorithm at the time of writing. This means that in terms of robustness it is not really an optimal solution since any kind of lossy compression of the carrier media which, for example, is used by Facebook, would corrupt the message we would try to hide. This goes for any of the algorithms we have implemented in our program.

To cope with these compression issues we have added signatures for all of our algorithms as well as a hash value in the embedded messages. The signature is used to validate that the algorithm used for extraction is the same as the one used for embedding. The hash value validates that no data has been corrupted or changed. In this case, the user will at least know whether the data is corrupted or not.

Time complexity

Of all the algorithms we have implemented LSB has, as shown in practice earlier, the lowest time complexity by far. The computational time only depends on how many bits are embedded which is denoted with **n**. Its complexity can be described with the formula:

$$O(n)$$

We consider CSA a middle ground of all the implemented algorithms. It does not only depend on the amount of bits to embed **n** but it is also dependent on the amount of common samples **m**. For every iteration of **n** a list containing **m** elements will have to be searched and sorted by its distance to **n**. Its time complexity can be described with the formula:

$$O(n * m^2 * log(m)))$$

For GTA the complexity calculations are a bit more complex. Instead of being directly dependent on the number of bits in the message, it is instead dependent on the number of vertices **V** which is $\mathbf{n}/MessageBitsPerVertex$. The main part of the GTA, the part that finds edges, is further dependent on the size of the neighborhood **N** which is defined with the *MaxDistance* and *DivisonPrecision* factors as described in subsection 7.4.3. As such the complexity of this process is:

$$O(V * N^3)$$

The algorithms in the program utilize a fairly short time to accomplish its assignments. However, we feel that with more time and knowledge the algorithms can still be improved upon, particularly GTA. This does not mean that the algorithms are not effective in their current state, but there is definitely space for improvement in regards to this criteria.

Imperceptibility

Regarding the imperceptibility of LSB it is difficult for a human to detect a change in the carrier media as the changes done to an image is so subtle that it would not be noticeable to the human eye. However, it is prone to be detected by almost any means of steganalytic methods as the LSB method has been frequently used as a base of steganographic messaging on computers where steganalysists have worked around it for some time.

The CSA is theoretically harder for a computer to detect, compared to our LSB algorithm, as the bit pattern of the individual bytes has not been changed but has instead just swapped places with already existing byte patterns. However,

it will be more prone to human detection than LSB if the original image is known meaning that a picture will possibly look different from the original image depending on what settings are used by the algorithm.

The GTA should be the most difficult algorithm to detect both statistically and by a human comparing the cover- and stego-file. The reason for this is that the amount of unique samples is closer to the original amount after embedding. LSB and CSA would respectively increase and decrease the amount of unique samples in most cases.

To add to the imperceptibility of our steganographic algorithms we have implemented the option to encrypt the data before it is hidden in the carrier media. This is done so that if the carrier media containing the data is somehow found by an unwanted audience and they are able to extract the data from the carrier media, they would still need to decrypt the data.

We have chosen to implement two different cryptography algorithms, namely AES and RSA. We have chosen to implement both a symmetric and asymmetric cryptosystem to give the user the option to choose whichever suits them best. We have chosen AES as the default algorithm as it allows custom plain text keys and uses CBC.

In summation the GTA, as our main algorithm, provides excellent imperceptibility. However, it comes at a cost of high memory use and a squared time consumption. This remains one of the key problems with the program. Should the imperceptibility be improved it would require a way of working around the time and memory requirements that are currently limiting the program. A general rule for these algorithms is that the imperceptibility depends on the complexity meaning that a more imperceptible stego-file will take longer to compute. We have attempted to implement properties that can be used to balance between these two factors.

Bandwidth

In the spectrum of bandwidth the LSB algorithm is the best algorithm. Based on what we can achieve in testing, the LSB algorithm has a maximum bandwidth of the total amount of bytes describing the carrier media divided by 8 since one bit is stored per byte. Finally the length of the algorithm's signature is subtracted. For example, a 1024x1024 picture with 3 color channels would have a maximum bandwidth of 393212 bytes enabling the embedding of equally as many UTF-8 characters by changing the least significant bit of every pixel. The bandwidth of the LSB algorithm can be expressed with the following formula:

$$\frac{NumSamples * BytesPerSample * 1bit/byte}{8bit/byte} - Signature.Length$$

With CSA the maximum bandwidth is calculated from the amount of bytes in the carrier media divided by the amount of bytes per sample. This gives us

the number of samples where we can potentially store a single bit in each of them. To get the bandwidth in bytes we divide the amount of samples by 8 and subtract the length of the algorithm's signature. So with the same 1024x1024 picture with 3 channels as previously mentioned we would get 131068 bytes, which is a third of the LSB bandwidth. This can be expressed with the formula:

$$\frac{NumSamples * 1bit/sample}{8bit/byte} - Signature.Length$$

The GTA is a little more complex when it comes to the calculation of the available bandwidth. Nonetheless, the algorithm has the same ideal bandwidth as CSA, which is 131068 bytes, assuming ideal conditions. However, we can hide up to 4 bits per vertex which means we can hide significantly more data. This does also mean that the data will not be very well hidden, but the theoretical bandwidth is 524284 bytes.

$$\frac{NumSamples * BitsPerVertex}{SamplesPerVertex * 8bit/byte} - Signature.Length$$

As shown by each algorithm's bandwidth equation there is a significant difference in the amount of data that we can hide in a carrier media. This is of course without taking imperceptibility into account as hiding such a large quantity of data, with any of the respective algorithms, will make it susceptible to detection either by human means or statistical analysis. However, we observe that we can hide quite significant amounts of data with any of the given algorithms which should be more than sufficient for sending hidden messages of quite a large length. This does however come with the same memory and time requirements as the imperceptibility requirement as embedding a large set of data would require a large primary memory source and quite some time.

Versatility and popularity of carrier media

We have somewhat succeeded in keeping the versatility of carrier media selection as the solution does support the importation of most common image formats, but only exports in the PNG format. The 'somewhat' comes in relation to the audio files, as the solution only works with PCM WAVE files, which is not supported by the main algorithm at the moment. In addition to this, the other algorithms do not produce satisfying results when embedding into WAVE files.

For increased versatility, we have added the ability to import plugins which can contain definitions for new carrier media. This means that dynamic extensions can be made to the program.

User security

To give the user some form of security in the case that the stego-message is detected and intercepted, we have implemented cryptography which makes sure

that the message will not be interpretable by the interceptors unless a private key is also intercepted.

As earlier debated in section 4.5 and section 4.5, an encryption or decryption key is needed for our cryptographic algorithms to function. The sharing of these keys is largely an issue we leave up to the individual users, but should the users want to optimize the security of their keys it is possible for them to encrypt a message, containing an asymmetric encryption key, with a symmetric-key. This solves the issue of RSA keys being easily recognizable, but the users obviously still need a way to share the symmetric-key with each other.

User interface ease of use

The GUI has a simple interface where we have attempted to use as few buttons as possible. We evaluate that this makes it easier for inexperienced users to use the program combined with the fact that all critical components have been described by adjacent texts, which aim to aid the user in their understanding of each component. In addition to this, every section that is not necessarily needed is labeled '(optional)' such as encryption key, text message- and files to hide.

Advanced options have been separated from the regular options through the use of a separate tab. The advanced options tab aims to ensure that users will think twice about using it seeing as the advanced options tab enables the user to change critical variables which, in worst case scenario, could lower the security of the steganographic solution. Warnings for what could happen when changing these settings is also shown below the settings section.

The GUI features a drag and drop function implemented in order to make the program as intuitive as possible. It makes it easy to get the files needed and does not require too much thinking from the users.

Unavailable buttons are greyed out meaning that they can not, and should not, be pressed until the user activates the corresponding function under the 'Advanced Options' tab. This serves to further make the GUI more intuitive and helps guide the user through the entire process.

All in all we deem the GUI satisfactory intuitive because everything the user can do with the program is described in the corresponding section. Furthermore, relevant warnings are shown to the users where the settings can be changed.

9.2 Moral and ethical issues

Regarding the moral and ethical issues, as earlier debated in section 3.2, we deem that the potential of free communication and the increased understanding between ethnic groups and countries have a significantly higher positive outcome than the possible negatives. It is not that we are not aware of the possibilities of the program being used by groups that would do harm to others, but more that the groups that would do harm to others is represented by a very small margin compared to the potential of free communication.

Furthermore, it is to be noted that the program does not reinvent steganography, but rather it aims to assist individuals with low to medium tech knowledge create a steganographic message with relative ease. The program provides a tool to ease this process and could potentially help whistleblowers, journalists and activists alarm the world about moral or ethical problems. Generally put, it might help other parties take action against unacceptable behavior.

9.3 Final evaluation

The fulfilling of the program criteria leads us to believe that the program would be able to fulfill the role as a steganographic tool that would be able to ease the use of steganography for users that may not have the larges of tech knowledge as the program was intended to do. However, the program does still have insignificant flaws, compared to the overall objective of the program, that should still be improved upon in the future.

Future work 10

This chapter will cover a discussion on the various subjects that we have covered in our evaluation. This includes, but is not limited to, features we did not implement, why it was not implemented and what general influence it had on the end solution.

Decoy keys

In the design chapter we argued that it would heighten the security of our end solution, if it would be possible to use a decoy key, and thereby have a key that the user could give away in emergency situations.

This method has only been implemented as a byproduct of our LSB algorithm as this algorithm has the possibility to select which bit should be read from and written to. However, we deem that it does not have a large impact on the end solution as using the other algorithms would give the user a large advantage to how well the information is hidden.

Although we have come to this conclusion regarding the decoy key it would be beneficial to have it implemented in the future as an extra layer of security to the user. However, it had a smaller priority than almost every other feature that we have implemented in the program.

User guide

In the design section we mentioned that we would make a built in user guide. This would have the purpose of guiding the user through the use of the program and make sure that the user would not make any bad decisions that would compromise the security of the user. We only made this guide to some degree in the form of precise descriptions of what to do in different sections of the program as well as showing warnings about what could happen that would compromise security at critical places. For example, the user is warned if no encryption key is used and also warned when trying to access the 'Advanced Options' tab.

Key size warning

We also mentioned in the design section that we would give the user a warning if the key chosen was not long enough. We would do this to make sure there was a decent layer of security. We choose not to do this but rather hash the key with a salt. This, however, will not alter the risk of brute-force attacks since

the actual key to write as input will still be the same with the salt added. As future work it would be purposeful to add the previously mentioned warning to have a decent key length and thereby key strength to minimize the risk of said attacks.

Share button

As an extra feature, a share button could be implemented, to ease the uploading process and guide the user to social media sites that are compatible with the program.

Generic *Sample* class

Currently the various algorithms do not handle different carrier media sample types any different. A seemingly straightforward way to rectify this would be to make the *Sample* class generic. However, the inability to set a generic type constraint specifying value types makes it more complicated as we would be unable to handle the samples in the ways we need for the algorithms by default. To make it possible it would necessitate the implementation of custom operator overrides for every operation we would like to use on the samples.

Correction of minor bugs

There are some minor bugs in the program, such as the capacity bar being off by a few bytes, although we have yet to determine what the exact problem is. Another issue with the capacity bar is that it is frequently updated, which can cause some delay in the thread serving the GUI.

Progress bar improvements

For the algorithms we have implemented so far, the progress bar does not fully reflect the actual embedding progress. For GTA, only the edge finding is taking into account, so the part of the algorithm which finds vertices is not accounted for. This means that the progress bar can be stuck at 0% for several minutes if a very large carrier is used.

Dynamic Algorithm Settings

Currently the steganography algorithm are set manually or left at default. A possible improvement would be to make them dynamic in the sense that the program would automatically select the optimal settings according to the chosen carrier media and message size. Settings could also be scaled based on available hardware, since the default algorithm can require a lot of memory with the default settings.

GTA Reserve Matching

A feature we considered implementing but did not do, is adding a step in the GTA between swapping and forced adjustments where the leftover vertices are

matched with the reserve vertices that were not necessary to represent the message. This would further improve the statistical imperceptibility.

Conclusion 11

In this chapter we will conclude on our end solution and its ability to achieve its determined requirements. We will also answer the problem definition and sub questions we have established earlier.

In regards to the problem definition we have ensured imperceptibility against known statistical analysis by utilizing a graph-theoretic approach to hiding data into a carrier file, making it possible to send steganographic messages, with a reasonable bandwidth, undetected. This can be said because graph theory results in roughly the same amount of unique pixels, as in the original image, before embedding. To add further security, encryption keys can be shared by embedding asymmetric keys and encrypt them with a symmetric key thereby making it possible to make an inconspicuous plain text key. This makes the initial exchange of keys safer. We cannot completely remove key sharing as a factor, but only make it harder for other parties to detect the key exchange.

GTA fulfills the requirements of imperceptibility and bandwidth, but the requirement of time complexity only to some extent. However, if a faster method is needed by the user the CSA algorithm can be used which lacks a little in the imperceptibility but has a better time complexity. If even better time complexity is needed the LSB algorithm can be used which has the worst imperceptibility of the three algorithms but is the fastest.

With security in place, we have created a GUI that is set up to feel and look intuitive and easy to use. This should hypothetically broaden the user space as utilization of the program because of the minimalistic GUI and general focus on user friendliness. Furthermore, technologically experienced users has the possibility of using the advanced options tab, should they so desire, once again ensuring a wide target audience and optimizing each steganographic session to the user's own liking.

Through these implementations our solution presents a program that can enable secure and covert communication by creating steganographic messages using one of three different algorithms and one of two different secure encryption methods. It will use the specified methods to embed the message into either a picture file (PNG) from any image file format or an audio file (WAV). These messages will be able to be shared through different social media such as Imgur and Skype, but not the likes of Facebook and Twitter due to their use of compression.

We have utilized a unified coding convention in order to ensure that future development of the program is as easy as possible while also allowing other

developers, who might have an interest in the program, to get a better understanding of the inner workings of the program. We have also added the possibility of importing algorithms on run-time further increasing the versatility of the program.

Unit tests have helped us isolate issues within smaller chunks of the program when the errors were not apparent. Additionally, mock tests helped us test the combined program in action. This provided us with the means to detect errors in the program that would otherwise have been errors in the end solution. Though utilizing unit and mock tests do not guarantee that bugs would not make it into the end solution, it does minimize the potential. According to the tests established, which returned a positive answer, the program functions well and runs as expected.

The moral issues of the project leads us to understand that even though we want to implement ways of securing the program from ill intent we have not found a way of creating such security without removing the anonymity of our solution, hence rendering it useless to our intended audience.

The legality of the program has been a subject of in-depth research as we have no intent of breaking any laws with our end solution. However, we have concluded that it would not be necessary to incorporate laws outside of the European Union as these are not of our concern. With that said, a potential user should be aware that laws regarding cryptography in their region or country may differ from that of this report.

The versatility of the carrier media is not as versatile as we would have wanted. Even though a partial solution is in place for using WAV files as carrier media it is, however, still necessary to make alterations to the algorithms to make these compatibilities function as intended.

Future work with the program would also require further study into the robustness of carrier media as corruption of carrier media, due to compression, is still an issue meaning that the stego-files cannot be shared wherever and still retain the embedded data.

In summation, our solution provides a safe and intuitive tool for steganographic messaging by providing security, user friendliness and a high imperceptibility to combat the challenges that an activist might face in regards to free communication.

Appendix 12

12.1 Coding style

Writing a program of this scale, and with multiple participants, it is important to set some coding style guidelines. These guidelines aim to keep the code easy to read and get an overview of. When all members of the group has agreed on a specific coding style there will be less confusion as to what a specific part of the program means or why it's written as it is.

The coding style we have decided to follow is as follows:

- All code must be written in English.

Code is easier to read when it is written in a universal language, such as English and usually much more compatible. Some compilers do not support characters that are not standard in the English alphabet, such as æ, ø or å. Also, by having the code written in English it can be understood by a broader range of people.

- Variables are defined with the lower camelcase style.

Lower camelcase means that the first word, if multiple, will always start with a lowercase letter and any following words will start with an uppercase letter.

Example:

```
1 int thisVariable;
```

- Methods, classes and properties are defined with the upper camelcase style.

As opposed to lower camelcase we also have upper camelcase in which every word in the method, class or property name starts with an uppercase letter.

Example:

```
1 class ClassName{}
2 void MethodName{}
3 string PropertyName { get; set; }
```

- Lengthy names are no shame.

Visual Studio, which is the code editor we use, has an auto correct feature and we are therefore not required to write everything out every time we need it. Therefore longer descriptive names are a possibility and helps to make the code more readable.

- Avoid the use of 'var' and implicit casting.

In C# there is a type declaration called 'var' which can be used to get the compiler to 'guess' the variable type. This is useful with long types, but it subtracts from the code readability. The same applies to implicit type casts, it is much easier to read code if the typecasts are visible at all times. An implicit cast is when a type converts to another type automatically.

- Indents are 4 spaces and the tabulator button can only be used if it has been set to replace tabs with spaces.

Different editors respond differently to tabs, but the same way to spaces. This is why we chose to convert all tabs to spaces, to accommodate for different text editors.

- Comments follow the same indention as the rest of the code.

If a comment is linked to a specific line of code, it will be on the line above the code, if it is linked to a block of code, it will be place on the line before the block starts.

- Everything within a loop or statement has to be in a block.

A block is a piece of code within 2 curly brackets. These curly brackets will always be present in all loops and statements. This also applies if the loop or statement only effects one line of code. This does not apply to properties.

- Curly brackets will always be on a line of their own.

This makes it easier to see where a block starts and where it ends.

- Global variables of a class will be defined in the top of the class definition.

Example:

```
1 Class ClassName
2 {
3    private bool globalClassVariable;
4    private int anotherClassVariable;
5
6    void MethodsStartsHere(){}
7 }
```

- Comments starts with a capital letter and is ended with a dot.

Additionally, Every method will have a comment describing its function, parameters and output if any.

Example:

```
1 //This is a comment.
2
3 /// <summary>
4 /// This comment describes the following method.
5 /// </summary>
6 void MethodName(int _parameter){return returnVariable;}
```

- The name of a parameter of a method will start with an underscore.

This will make it easy to distinguish parameters from variables.

For example:

```
1 void SomeMethod(int _parameterName){}
```

12.2 Glossary

AES: Advanced Encryption Standard, using private key cryptosystem.

ASCII: Characters represented by a numeric value, that can be interpreted by a computer.

BMP: Windows Bitmap. Can store digital images of arbitrary width, height and color depth.

Carrier media/file: A media, like text message, picture file or even tattoos, that can be used to hide a message in.

CBC: Cipher Block Chaining. A block cipher mode where each block of plaintext is XORed with the previous cipher text block or an initialization vector.

Cipher text: The encrypted message.

Cryptography: Securing the communication from third party adversaries, with constructing and analysing protocols.

CSA: Common Sample Algorithm.

Graph theory: The study of graphs. Connecting pairwise relations between objects and changing them to make the bit pattern fit the message.

GTA: Graph-Theoretic Algorithm.

GUI: Graphical User Interface.

JPEG: Commonly used compression format for digital images

Lossless compression: Removes redundant information, but can reconstruct the original data perfectly after compressing.

Lossy compression: Removes redundant information. Cannot be reconstructed to the original data.

LSB: Least significant bit. Also refers to the method that changes the least significant bit in every byte to be equal to a message.

mod x: Modulo x. The remainder after division.

Pixel: The smallest addressable element in a picture. Is a contraction of picture element.

Plain text: The message before encryption.

PNG: Portable Network Graphics, a fileformat that supports lossless data compression.

RGBA: Stand for the red, green, blue and alpha colour channels in a pixel, where alpha is the transparency of the pixel.

RSA: An asymmetric cryptosystem, which uses public/private keys.

Salt: A value used to add length and strength to a key.

Sample: A sample is a collection of n bytes, where n is the amount of channels. This is a pixel in an image.

Stego-file: A carrier file in which data has been embedded steganographically.

Steganography: Securing the communication from third party adversaries, by concealing a message, file, image and so on, so the third party does not notice that a message was sent.

Uncompressed: A format which is not compressed by any means.

WAV: Waveform Audio File Format, an uncompressed audio file format.

12.3 Inheritance diagram

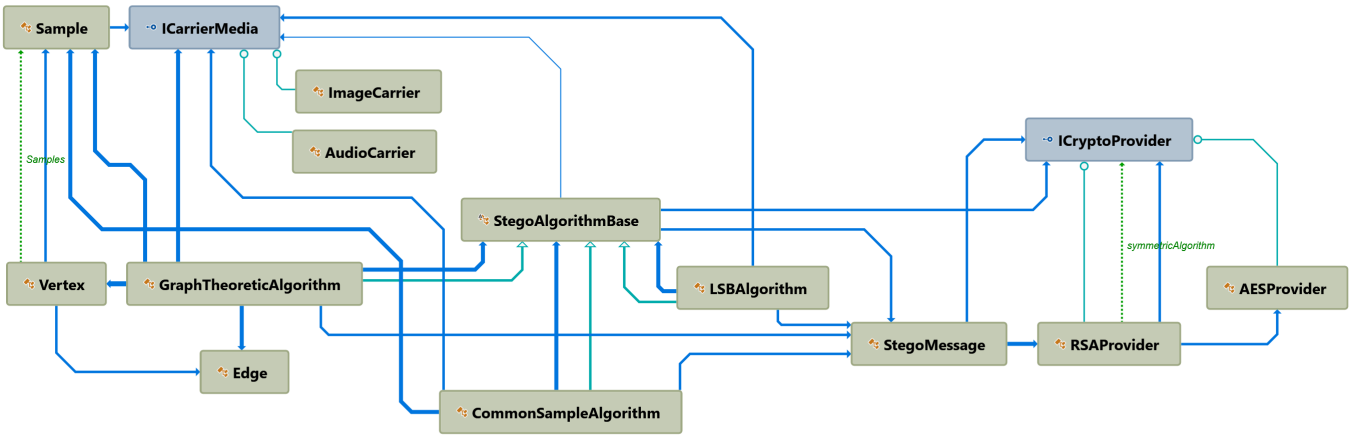


Figure 12.1. Class/inheritance diagram of the program.

Bibliography

[1] M. Michaelis and E. Lippert, *Essential C#*, ch. 5. 16.

[2] *Overvågning*. https://amnesty.dk/emner/frihed/overvaagning. Visited: 29/02/16.

[3] M. Warkentin, E. Bekkering, and M. B. Schmidt, *Steganography: Forensic, Security, and Legal Issues*. http://ojs.jdfsl.org/index.php/jdfsl/article/view/179, 2008. Visited: 29/02/16.

[4] *Steganography and Malware: Concealing Code and C&C Traffic*. http://blog.trendmicro.com/trendlabs-security-intelligence/steganography-and-malware-concealing-code-and-cc-traffic/, 2015. Visited: 20/05/16.

[5] Gyldendal den store danske, "Borgerkigen i syrien," visited: 21-03-16.

[6] *What's Next in Government Surveillance*. http://www.theatlantic.com/international/archive/2015/03/whats-next-in-government-surveillance/385667/, 2015.

[7] S. Jones, "Spioner boltrer sig på twitter," *Politiken*, vol. Artikel id: e5252e10, p. 9, 2015. 01-08-2015.

[8] B. Winegarner, *6 ways journalists can keep their reporting materials private and off-the-record*. http://www.poynter.org/2013/6-tips-for-keeping-your-secret-communications-secret-and-your-anonymous-sou 215242/, 2013. Visited: 29/02/16.

[9] *Case of Goodwin v. the United Kingdom*. http://worldlii.org/eu/cases/ECHR/1996/16.html, 1996. Visited: 29/02/16.

[10] T. H. Susan E. McGregor, Polina Charters and F. Roesner, "Investigating the computer security practices and needs of journalists," tech. rep., http://www.franziroesner.com/pdf/journalism-sec15.pdf, 2015.

[11] *Hackers used data exfiltration based on video steganography*. http://securityaffairs.co/wordpress/30624/cyber-crime/hackers-used-data-exfiltration-based-video-steganography.html, 2014. Visited: 18/02/16.

[12] *Sunsets and cats can be hazardous to your online bank account*. http://blog.trendmicro.com/trendlabs-security-intelligence/sunsets-and-cats-can-be-hazardous-to-your-online-bank-account/, 2014. Visited: 20/05/16.

[13] M. Rouse, *Definition back door*. http://searchsecurity.techtarget.com/definition/back-door, 2007. Visited: 29/02/16.

[14] B.-J. Koops, *Crypto law Survey*. http://www.cryptolaw.org/, 2013. Visited: 09/03/16.

[15] *IT-retten.dk*. http://www.it-retten.dk/bog/04/02/. Visited: 09/03/16.

[16] *List of dual use items and teknology*. http://trade.ec.europa.eu/doclib/docs/2008/september/tradoc_140595.pdf. Visited: 09/03/16.

[17] S. Gupta, G. Gujral, and N. Aggarwal, *Enhanced least significant bit algorithm for image steganography*. http://www.ijcem.org/papers072012/ijcem_072012_08.pdf, 2012. Visited: 29/02/16.

[18] *DEFLATE Compressed Data Format Specification version 1.3*. https://tools.ietf.org/html/rfc1951, 1996. Visited: 18/02/16.

[19] P. Fisher, "The pixel: A snare and a delusion," *http://www.tandfonline.com/*, vol. Volume 18, Issue 3, p. 8, 2010. Visited: 17/02/16.

[20] F. Al-Kalani and M. A. Rabei, "Implementation of image encoding based on rgb and argb," tech. rep., http://computerresearch.org/index.php/computer/article/view/532/532, 2012. Visited: 24/02/16.

[21] S. A. Khayam, "The discrete cosine transform (dct): Theory and application," tech. rep., http://wisnet.seecs.nust.edu.pk/publications/tech_reports/DCT_TR802.pdf, 2003. Visited: 23/02/16.

[22] *Audio File Formats*. http://www.nch.com.au/acm/formats.html, 2015. Visited: 24/02/16.

[23] *Lossless and lossy audio formats for music*. http://www.bobulous.org.uk/misc/audioFormats.html, 2009. Visited: 24/02/16.

[24] *Information hiding using audio steganography*. http://airccse.org/journal/jma/3311ijma08.pdf, 2011. Visited: 24/02/16.

[25] A. Beach, *Video compression*, ch. 2. 2008.

[26] A. S. K. Mennatallah M. Sadek and M. G. M. Mostafa, "Video steganography: a comprehensive review," *Multimed Tools Appl*, vol. 74, pp. 32, 7063–94, 2015.

[27] *Hiding information in retransmissions*. http://arxiv.org/ftp/arxiv/papers/0905/0905.0363.pdf. Visited: 03/03/16.

[28] *Definition of packet.* http://searchnetworking.techtarget.com/definition/packet, 2007. Visited: 03/03/16.

[29] M. El-Nawawy and S. Khamis, "Political activism 2.0: Comparing the role of social media in egypt's "facebook revolution" and iran's "twitter uprising"," *CyberOrient, Online Journal of the Virtual Middle East*, vol. 6, no. 1, 2012.

[30] *Facebook statistics.* http://www.socialbakers.com/statistics/facebook/. Visited: 18/02/16.

[31] *Online Social Media in the Syria Conflict: Encompassing the Extremes and the In-Betweens.* http://arxiv.org/pdf/1401.7535.pdf, 2014. Visited: 18/02/16.

[32] *Reddit Audience And Demographics.* https://reddit.zendesk.com/hc/en-us/articles/205183225-Audience-and-Demographics, 2015. Visited: 24/02/16.

[33] *Whistleblowers.* http://www.whistleblowers.org/index.php?option=com_content&task=view&id=899&Itemid=170, 2014. Visited: 18/02/16.

[34] *RSA encryption - Wolfram.* http://mathworld.wolfram.com/RSAEncryption.html, 2014. Visited: 29/02/16.

[35] *RSA encryption - Burt Kaliski.* http://mathaware.org/mam/06/Kaliski.pdf, 2014. Visited: 29/02/16.

[36] K. Krishnan, "Computer networks and computer security," tech. rep., http://www4.ncsu.edu/~kksivara/sfwr4c03/lectures/lecture9.pdf, 2004. Visited: 29/02/16.

[37] M. D. Ryan, "Symmetric-key cryptography," tech. rep., https://www.cs.bham.ac.uk/~mdr/teaching/modules/security/lectures/symmetric-key.html, 2008. Visited: 29/02/16.

[38] C. Savarese and B. Hart, "The caesar cipher," tech. rep., http://www.cs.trincoll.edu/~crypto/historical/caesar.html, 2010. Visited: 29/02/16.

[39] J. Graham-Cumming, "Why some cryptographic keys are much smaller than others," 2013. Visited: 14/03/16.

[40] Çetin Kaya Koç, "Cryptographic engineering," *https://books.google.dk/books?id=nErZY4vYHIoC&pg=PA321&redir_esc=y#v=onepage&q&f=false*, vol. Issue 1, pp. p321–323, 2009. Visited: 29/02/16.

[41] M. Robshaw, "Stream ciphers," tech. rep., ftp://ftp.rsasecurity.com/pub/pdfs/tr701.pdf, 1995.

[42] P. C. v. O. Alfred J. Menezes and S. A. Vanstone, *Handbook of Applied Cryptography*, ch. 7. 2001. Visited: 29/02/16.

[43] H. Wu, "Cryptanalysis and design of stream ciphers," tech. rep., https://securewww.esat.kuleuven.be/cosic/publications/thesis-167.pdf, 2008. Visited: 29/02/16.

[44] *DissidentX on GitHub.* https://github.com/bramcohen/DissidentX, 2013. Visited: 22/02/16.

[45] *Steghide documentation.* http://steghide.sourceforge.net/documentation/manpage.php, 2002. Visited: 22/02/16.

[46] *PNGDrive on GitHub.* https://github.com/claus/PNGDrive, 2012. Visited: 29/02/16.

[47] J. F. Vahid Sedighi, Rémi Cogranne, "Content-adaptive steganography by minimizing statistical detectability," tech. rep., http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7289422, 16. Visited: 18/02/16.

[48] S. Hetzl and P. Mutzel, "A graph–theoretic approach to steganography," Visited: 22/03/16.

[49] T. G. P. Handbook, *Making and verifying signatures.* https://www.gnupg.org/gph/en/manual/x135.html. Visited: 21/03/16.

[50] *Broadcast Wave Format (BWF) user guide.* https://www.ebu.ch/fr/technical/publications/userguides/bwf_user_guide.php, 2007. Visited: 21/03/16.

[51] *Intro to Audio Programming, Part 2: Demystifying the WAV Format.* https://blogs.msdn.microsoft.com/dawate/2009/06/23/intro-to-audio-programming-part-2-demystifying-the-wav-format/, 2009. Visited: 17/03/16.

[52] *Microsoft Developer Network, .NET Framework library.* https://msdn.microsoft.com/en-us/library/mt472912%28v=vs.110%29.aspx, 2015. Last Visited: 23/05/16.